

The Twitter-Testbed

A Social Network Simulation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Andreas Scharf

Matrikelnummer 0225735

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung

Betreuer: Univ.Prof. Dr. Schahram Dustdar

Mitwirkung: Univ.-Ass. Dipl.-Ing. Martin Treiber

Wien, 04.07.2012

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Andreas Scharf
Kerngasse 1
2353 Guntramsdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Juli 2012

(Unterschrift)

Abstract

Social networks have reached more than one hundred million users worldwide. This large amount of data makes it interesting for scientific studies and the development of specific programs for their investigations. In contrast to the numerous developments and the large amounts of data to work on, test data to evaluate the integrity of the code is available only in a very limited amount.

Although it is basically possible to read test data directly from the social networks, the high expenditure of time and the problem with anonymisation of personal data hinder the publication of new datasets and are certainly the main reasons for the lack of available test data.

In this master thesis a program is being developed which uses Forest Fire Model for the generation of artificial networks of any size and number that exhibit structures similar to that of the Twitter network. Together with an integrated simulation of communications provided by the tool, the possibility of a realistic simulation of the overall processes in this social network is established. This way, a basis for extensive testing scenarios can be provided.

The output data of the program shall be an adequate approximation of a real social network without real personal data provided in feasible form to serve as input to other programs. By using the Twitter-Testbed as a source of anonymous test data, the development process is accelerated and the quality of the results can be stabilized. The cost and time involved should be reduced significantly due to the elimination of the problems that come into play when trying to gather data from the real network.

Kurzfassung

Soziale Netzwerke haben heutzutage Dimensionen jenseits der hundert Millionen Benutzer erreicht. Diese große Datenmenge macht sie interessant für wissenschaftliche Untersuchungen und die Entwicklung von spezifischen Programmen, welche sie untersuchen. Im großen Kontrast zu den unzähligen Entwicklungen und den großen Datenmengen auf denen diese arbeiten sollen, stehen Testdaten zur Evaluierung der Integrität nur äußerst beschränkt zur Verfügung.

Obwohl es grundsätzlich möglich ist Testdaten direkt aus den sozialen Netzwerken auszulesen, steht dem ein hoher zeitlicher Aufwand und die Problematik der Anonymisierung von personenbezogenen Daten entgegen, welche den genannten Mangel an verfügbaren Testdaten begründen.

In der vorliegenden Master-Arbeit stellen wir ein Programm vor, welches das Forest Fire Model zur Grafengenerierung nutzt, um künstliche Netzwerke in beliebiger Größe und Anzahl zu erstellen, welche ähnliche Strukturen wie das Twitter Netzwerk aufweisen. Zusammen mit einer integrierten Simulation von Kommunikationsvorgängen, bietet dieses Werkzeug die Möglichkeit, einer realitätsgetreuen Nachbildung der Vorgänge in diesem sozialen Netzwerk und damit eine Grundlage für ausgiebige Testszenarios.

Die Ausgabedaten des Programms stellen eine adäquate Approximation eines realen sozialen Netzwerks dar, beinhalten dabei aber keinerlei personenbezogener Daten. Durch den Einsatz des Twitter-Testbeds als Quelle für anonyme Testdaten, kann der Entwicklungsprozess beschleunigt und die Qualität der Ergebnisse stabilisiert werden. Die Kosten und der Zeitaufwand sollten auf Grund des Wegfalls der beschriebenen Problematiken spürbar reduziert werden.

Acknowledgments

First of all, I want to thank Martin Treiber for his great efforts in keeping me on track and for supporting me throughout the whole process of writing this research-based paper. Additional thanks go to Schahram Dustdar for giving me the possibility to write this thesis and his generosity in providing resources.

Furthermore, I want to thank my parents for their support, love and help throughout my entire life and especially during the process of writing this final paper.

A very special thanks goes out to my children for giving me strength to make it through the tough times and tricky passages of writing this paper.

Finally, I want to thank all my friends and relatives who supported me in their own ways and made writing this thesis a pleasant experience to me.

Contents

1	Introduction.....	1
1.1	Motivation.....	3
1.2	What is Twitter?.....	5
1.3	Organisation.....	7
2	Related Work.....	9
2.1	Graph Generation.....	9
2.2	Properties and Behavioural Patterns of Social Networks.....	10
2.3	Recent Analysis of the Twitter Network.....	11
2.4	Simulation.....	12
2.5	Social Sciences.....	12
3	Methodology.....	14
3.1	Design.....	14
3.1.1	Graph Theory Basics.....	15
	Distinction of Graph Types.....	16
	Graph Characteristics.....	16
3.1.2	Characteristics of the Twitter Network.....	20
3.1.3	The Forest Fire Model.....	22
	Extensions of the Model.....	24
3.1.4	Communication Simulation.....	24
	Scheduler.....	26
	Event Types.....	27
	Network Behaviour.....	31
3.1.5	Input and Output Interfaces.....	37
	Graph Structure.....	37
	Communication Data.....	38
	Statistics and Debug Mode.....	39
3.2	Implementation.....	40
3.2.1	The JUNG Framework.....	40
3.2.2	The GraphML File Format.....	41
3.2.3	Status.Net.....	43
3.2.4	Graph Generation.....	45
	Class graphs.Node.....	46
	Class graphs.TwitterNode.....	46
	Class graphs.Edge.....	46
	Class graphs.TwitterEdge.....	47
	Class generators.ForestFireModelGenerator.....	47
	Class generators.Initializer.....	50
3.2.5	Communication Simulation.....	51
	Class events.Event.....	53
	Class events.EventTweet.....	53
	Class events.EventMention.....	54

Class events.EventDirectedTweet.....	54
Class events.EventRetweet.....	55
Class events.EventReply.....	55
Class simulation.Scheduler.....	56
Class simulation.TwitterScheduler.....	57
3.2.6 Input and Output.....	57
Class io.TwitterGraphMLReader.....	57
Class io.TwitterGraphMLWriter.....	57
Interface ConversationConnector.....	58
Class ConversationConnectorCSVWriter.....	58
Class ConversationConnectorXMLWriter.....	59
Class ConversationConnectorDB.....	60
4 Application Usage.....	63
Parameters for Graph Generation:.....	63
Parameters for Graph Properties Initialisation:.....	64
Parameters for the Simulation:.....	65
Parameters for Input and Output:.....	66
Parameters for Status.Net:.....	66
5 Evaluation.....	67
5.1 Graph Generation Metrics.....	68
5.1.1 Reality Conformance.....	68
5.1.2 Speed.....	69
5.1.3 Memory.....	70
5.2 Communication Evaluation.....	70
5.2.1 Reality Conformance.....	71
5.2.2 Speed.....	73
5.2.3 Memory Consumption.....	74
6 Conclusion.....	75
7 Future Work.....	78
7.1 Graph Generation.....	78
7.2 Communication Simulation.....	79
8 Appendix.....	I
A. References.....	I
B. Twitter-Testbed Properties File.....	III
C. Sample graphML File.....	IV
D. Source Code Samples.....	V
E. Sample XML Output File.....	VII
F. Evaluation Graphs.....	VIII
G. Properties File for the Evaluation Run.....	IX

List of figures

Figure 1: Bidirectional network.....	6
Figure 2: Unidirectional network.....	6
Figure 3: Graph types.....	16
Figure 4: Sample power law degree distribution.....	17
Figure 5: Shortest Path calculation samples.....	17
Figure 6: Calculation of the local clustering coefficient.....	19
Figure 7: Calculation of reciprocity.....	19
Figure 8: Visual power law fit.....	20
Figure 9: Twitter-Testbed schema UML diagram.....	25
Figure 10: Scheduler (UML-diagram).....	27
Figure 11: UML: Scheduler - trigger tweet event.....	30
Figure 12: UML: Scheduler - trigger reply event.....	31
Figure 13: Properties of nodes and edges.....	33
Figure 14: Gaussian distribution ($m=0.8,s=0.1$)	34
Figure 15: Tweet types ratio breakdown.....	36
Figure 16: Tweet types ratio scenario separated breakdown.....	37
Figure 17: Application overview.....	40
Figure 18: graphML XML schema definition.....	42
Figure 19: graphML attributes specification.....	42
Figure 20: graphML graph structure.....	43
Figure 21: graphML end of graphml definitions.....	43
Figure 22: Status.Net home screen.....	44
Figure 23: Twitter home screen.....	45
Figure 24: Implemented events class schema.....	52
Figure 25: Class Event.....	53
Figure 26: Class EventTweet.....	53
Figure 27: Class EventMention.....	54
Figure 28: Class EventDirectedTweet.....	54
Figure 29: Class EventReTweet.....	55
Figure 30: Class EventReply.....	55
Figure 31: Class diagram for output of conversations.....	58
Figure 32: Sample CSV content for the conversation output.....	59
Figure 33: Status.Net affected tables EER diagram.....	62
Figure 34: Graphs creation times.....	69
Figure 35: Memory consumption of graph generation depending on network structure.....	70
Figure 36: Number of updates in a simulation run (10,000 nodes; 10 days simulation).....	72
Figure 37: Duration of the simulation process.....	73
Figure 38: Memory consumption of different graph sizes over time.....	74

List of tables

Table 1: Possible relations in bidirectional and unidirectional networks.....	6
Table 2: Figures of the Twitter network.....	22
Table 3: Figures for the communication simulation.....	36
Table 4: Properties of the initialiser class.....	51
Table 5: Parameters defined in class Scheduler.....	56
Table 6: Status.Net database entries required for table notice.....	61
Table 7: Status.Net database required entries for table conversation.....	61
Table 8: Status.Net database entries required for table reply.....	61
Table 9: Status.Net database entries required for table inbox.....	61
Table 10: Properties for process graph generation	63
Table 11: Parameters for the graph's initialisation.....	64
Table 12: Parameters for the simulation.....	65
Table 13: Parameters for input and output.....	66
Table 14: Parameters for Status.Net.....	66
Table 15: Evaluation of generated graphs (main figures).....	68
Table 16: Quota of message types in Twitter.....	71
Table 17: Ratios of message types in simulation (10,000 nodes; 10 days).....	72
Table 18: Updates per hour in different graph sizes.....	74
Table 19: Evaluation graphs settings.....	VIII
Table 20: Statistical measures of seven evaluation graphs.....	VIII

Listings

Listing 1: Example update text of a mixed message type.....	29
Listing 2: Defining a graph factory.....	45
Listing 3: Function addToRetweetedMessageQueue of TwiterNode.....	46
Listing 4: Function evolveGraph of class ForestFireModelGenerator.....	48
Listing 5: Function burn() of class ForestFireModelGenerator.....	50
Listing 6: Source code of the triggerEvent() function.....	54
Listing 7: Function start() of class Scheduler.....	56
Listing 8: graphReader - node transformer.....	57
Listing 9: graphWriter - node transformer.....	58
Listing 10: XML-syntax for tweets.....	59
Listing 11: XML-syntax for directed tweets.....	59
Listing 12: XML-syntax for retweet.....	59
Listing 13: XML-syntax for replys.....	59
Listing 14: Command-line starting command for Twitter-Testbed application.....	63
Listing 15: Sample properties file.....	III
Listing 16: Sample graphML file.....	IV
Listing 17: Source code of the calcResponseWillingness function.....	V
Listing 18: Source code of the getNextRandomTweet() function.....	V
Listing 19: Source code snippet of Status.Net connector.....	VI
Listing 20: XML output format.....	VII
Listing 21: Properties file used for evaluaton runs.....	IX

1 Introduction

Social network services (SNS) like Facebook¹, Myspace², Twitter³ and various other forms of that kind are a fast-growing business with hundreds of millions of users today. Facebook as the biggest SNS today lately reported over 750 million active users [1] and has still a growing rate of more than 10 million new users per month. Although, according to “Inside Facebook” [2] a slight decrease can be noted.

Today there are numerous SNSs supporting all kinds of interests and practices. While most of them focus on specific groups like authors (LiveJournal⁴), movie enthusiasts (Flixster⁵), photographers (flickr⁶), music fans (Last.fm⁷) or students (studiVZ⁸) and support them in their own specific way, their key technological features are fairly consistent.

SNSs typically support the user in generating a profile where the person can put information about themselves. Further, they provide the possibility to create a community by linking to other users in some way and to send messages and receive updates from others. Usually SNSs are also notifying a person of the activities of other connected users.

In these Social Networks millions of people are sharing information about themselves all of which is accumulated in a central point. Depending on preferences of the user, the available information can be very detailed: personal data like age, gender, education or interests are only a few of which is made available to the public. The relationships of nodes within these SNSs and this accumulation of private data makes their exploration very interesting for researchers from various different fields.

The growing availability of social network data facilitates the work of researchers. Instead of having to ask people directly in interviews, researchers simply tap into the data of social networks. In fact, social network data is often more detailed than data that can be retrieved when asking people directly.

SNSs are also interesting for computer sciences in terms of new technologies and network architecture. Social computing is engaged in ways of human communication as well as human-machine communication to support collaborative work through social networks.

1 <http://www.facebook.com>

2 <http://www.myspace.com>

3 <http://www.twitter.com>

4 <http://www.livejournal.com>

5 <http://www.flixster.com>

6 <http://www.flickr.com>

7 <http://www.lastfm.de>

8 <http://www.studivz.net>

A field that is strongly related to social networks is Graph Theory, which studies mathematical structures used to model the relations in social networks, building the methods to describe and compare them. Common questions in graph theory are questions like “How to find the shortest path between two nodes?” or “How to find clusters of nodes that are more connected to each other than to the rest of the network?”. Graph theory is only one specific research field out of various domains in the field of mathematics showing great scientific interests for social networks.

Today these services are almost all free of charge and financed mostly through advertising. With an advertising revenue of nearly 2 billion dollars for Facebook in the year 2010 [3] it is obvious that there is an incentive for economic sciences as well. There is an interesting potential for advertising focused on target groups because of the knowledge of user's interests and relations. “Viral marketing” is a keyword describing a field of economic sciences that uses existing social network platforms for marketing purpose [4].

In this thesis we will focus on the Twitter network, one of the most popular social networks today. Twitter has currently over 200 million users and a growing rate of about 460.000 users per day [5] [6] [7] [8]. The aim of the work is to artificially generate a graph that has similar properties as the real network and simulate the communication within. This way an infinite number of different yet similar networks can be created serving as test data for the development process of new algorithms in various fields aiming on social networks in any way.

In the next chapter (1.1 Motivation) the motivation of this work will be described in more detail. The questions “What is the goal of the project?” and “Why is this application a benefit for the scientific community?” will be answered in the following chapter. Chapter 1.2 we will describe the Twitter network and explain the differences and characteristics to other SNSs. The organisation of the rest of the work will be summarized in chapter 1.3.

1.1 Motivation

The raise of online social networks facilitated the study of network phenomena for scientific fields like mathematics, sociology [9], economics or computer science [4]. Existing scientific investigations of this phenomenon in the field of computer science focus on the description of Social Network Services [10], [11], their representation [12], [13], impact [14] and evolution [15]. Over the recent years there has been conducted a considerable amount of scientific research on how social network services can be used for others than the intended task of communication. Crowdsourcing, for example, aims to use popular social network services to manage business processes [16], [17], [18]. This chapter will explain our motivation to build a simulation of the Twitter network and the potential benefit such a network has for the scientific community. Further, the decision to choose Twitter as specimen will be outlined.

A common challenge for the development of applications regarding big social networks is gathering test data from the desired network. The extraction of the structure of social networks is a complex task and due to restrictions from the SNS-providers it is usually nearly impossible to obtain the whole graph without possible black holes. There are always nodes that cannot be reached, be it because of time, privacy settings, network structure or the algorithm used. Moreover, because of the impossibility to compare to the “ground truth”, one cannot estimate the influence of the missing data on the overall graph. Still, the extraction process for a representative piece of a popular SNS can take several weeks or even months depending on the size and the complexity.

Kwak et. Al [14] obtained 41.7 million nodes (users) and 1.47 billion edges (relations) in a two month continuous crawling attempt to the Twitter network and wrote the data to one 24GB graph file. Basically, what they did was a breadth-first search along the followers and followees starting at a very popular person. While this process is very time consuming, the result is an enormous amount of data and still does not capture the entire network.

Sharing the gathered information with the scientific community raises again some difficulties. The first difficulty bears the size of the file – To store this vast amount of data on a portable media would take several DVDs, transferring it through the internet even with a fast connection would take even days. The second difficulty lies in the matter of privacy protection. Before a dataset can be shared with the scientific community, all the private data has to be removed, profiles and messages are not allowed to be shared with the public because of privacy restrictions that have been set up by the service providers. Furthermore, the identification numbers, used to reference a profile, have to be anonymised in order to make it impossible to link the data to real user accounts. Although it is possible to reduce the gathered data to an anonymised structure, it has been shown that these efforts do not

provide a solution. By identifying unique subgraphs or using auxiliary graphs datasets can be de-anonymised by malicious parties [16], [17]. Because synthetic graphs do not contain private information they offer an attractive alternative to overcome these problems.

To be sure to represent the structure of the network accordingly even in early stages of the development process, the only way to test an algorithm is to run it against these huge graphs. Nevertheless, it should be taken into account that this process sparks off problems with memory consumption and the execution time of the program. Although, the developer might not be sure where the algorithm he is about to develop leads in the end, considerable problems with the test data have to be faced which make the development process very complicated and time consuming.

In order to assess the functionality of algorithms in a reasonable time frame, the development of algorithms of the huge data sets requires prior testing on similar, albeit smaller versions of social networks. To make the development process faster and more solid, a testbed for social network services is needed. Such a testbed is able to generate a graph according to the properties of the desired network and is scalable in matter of size considering the stage of development and the purpose of the software.

Literature research shows that there were only a small amount of papers published on the simulation of social networks in general. According to our assumption and due to our research results, it is our firm belief that this is the first attempt to simulate a social network including graph generation and communication simulation.

We expect our project and the program we developed to have a considerable impact and a great amount of benefit for the developmental process in all related research areas. The reason why we believe that is the following:

- It is possible to study the whole network structure without possible black holes of protected or unreachable content that bias the network structure.
- The process of getting test data is less time-consuming.
- The testing will get a stronger foundation because an infinite number of network variations can be generated and tested on.
- Scientists will have complete control over the test environment with the possibility to change the network structure and/or behaviour according to the problem.
- Content of communication can be defined to match some experimental syntax.
- A graphical representation of the simulation can help to understand the behaviour.
- By using synthetic graphs one would not have to deal with privacy concerns.

All of the above mentioned reasons lead to a faster and more solid development process for all scientific domains concerning social networks as listed in the Introduction (chapter 1).

The focus of this thesis is a simulation of the well known Twitter network. With over 200 million users worldwide, Twitter is not only one of the most popular social networks today but also has some outstanding differences to most other popular social networks. This makes Twitter an ideal candidate for our first approach of SNS-simulation.

Twitter restricts messages to a maximum of 140 characters in plain text. This simplicity make Twitter perfectly suitable for communication via all sorts of mobile devices. Although it separates the audience into "Followers" and people who are for whatever reason not capable of squeezing their messages they want to convey into 140 characters, this kind of simplicity of communication provides the perfect basis for further intense data analysis. The high information content makes it possible to use this protocol-like writing style for human-machine communication like for example in [18]. It will also ease the development of a test bed because there are no special media types or rating systems to deal with.

Furthermore, one difference to other social networks is the bidirectional relation structure of the Twitter graph. This means that there are actually four types of connections between users instead of two in a usual unidirectional network. The additional possibilities to connect to each other allows for more connections in total which means a wider reception area and a faster spread of information into the network. Because of all these above mentioned factors Twitter is even a more attractive subject for research in computer science.

Due to the given reasons this paper will put its focus on the Twitter network. The goal is to generate a network that has similar properties as those that can be found in the real Twitter graph and make a simple simulation of random communication between the connected nodes. Although this thesis aims to create a simulation of Twitter, this paper is just the first attempt of a social network simulation and could be extended for other social networks too.

1.2 *What is Twitter?*

Starting in march 2006, Twitter has become the most popular microblogging service today. Twitter is reported to have over 200 million registered users [7] and a growing rate of 460,000 new users per day [8], making Twitter one of the biggest social networks of the world. Starting out as a company internal research project for communication improvements, the Twitter service had experienced a vast increase in popularity soon after it won the South by SouthWest (SXSW) conference⁹ Web Awards in march 2007.

⁹ <http://sxsw.com/>

Twitter enables people to share short textual messages - “tweets” - with others within the system. The reason why Twitter has limited the characters of one tweet to 140 is because originally the service was designed for messages to be shared via SMS¹⁰. Though the service evolved to include more uses besides SMS, such as web and desktop clients, this limitation persisted and was therefore rearranged as a feature. “Twitter’s Creative Director Biz Stone argues: ‘creativity comes from constraint.’” [19]

Through the strict restriction of a maximum of 140 characters and no support for multimedia like photos or videos, the users have to express themselves precisely, which results in an average information content that is very high compared to other social networks. Twitter users have learned to make their message as short as possible and to even leave space for others to comment within the 140 characters when replying. Obviously the shortness of the messages influences the speed of information spread as one can read them quickly, download and display them fast and answer or forward within a short time. That made Twitter an important service for instant information sharing during various critical events and disasters around the globe e.g. the earthquake in Haiti 2010 or the forest fires in the United States (California) 2007.

Unlike most online social networking sites such as Facebook or MySpace, the relationship of “following” and being “followed” requires no reciprocation on Twitter. The difference between the more common unidirectional networks and a bidirectional structure is illustrated in Table 2. Figures 1 and 2 show a simple sample of a bidirectional, respectively unidirectional graph. As the graph clearly shows, the bidirectional network allows those kind of relationships where one person pays attention to another but the other person does not in return. A simple example of such a relationship is the one of a celebrity compared to their fans, where the celebrity does not pay the same amount of attention the way their fans pay to them. Such kind of relations not only make sense for famous people. For example, an employee might want to get all notifications from their boss but they might not want to get every notification from each of their workers.

	bidirectional	unidirectional
A not connected to B B not connected to A	●	●
A connected to B B connected to A	●	●
A connected to B B not connected to A	●	
B connected to A A not connected to B	●	

Table 1: Possible relations in bidirectional and unidirectional networks

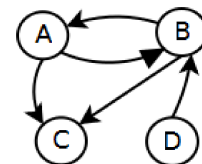


Figure 1: Bidirectional network

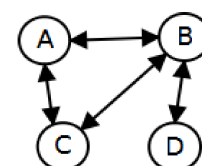


Figure 2: Unidirectional network

¹⁰ SMS (Short message service) is a text messaging service developed for mobile phones.

Twitter users are able to post direct and indirect updates. Direct posts are used if a user dedicated their update to a specific person, whereas indirect updates are used if the update is meant for anyone who cares to read it. Even though direct updates are used to communicate directly with a specific person, they are usually public and anyone can see them.

All tweets can also include so called “hashtags”. Deriving from the conventions in the historical communication channel IRC¹¹, words prefixed by a “#” are used to highlight a keyword or topic of the message. A Twitter user is able to categorize all messages by these tags and to subscribe to hashtags in order to receive tweets from all Twitter users mentioning that keyword. This practice speeds up the information processing even more as it connects people with the same interests even if they are initially not connected.

1.3 Organisation

The remainder of the thesis is structured as follows:

The next chapter (2) presents related scientific works of other associated areas that have inspired the work. They are grouped into five thematic headlines that reflect the effected part in this thesis: Graph Generation (2.1), Properties and Behavioural Patterns of Social Networks (2.2), Recent Analysis of the Twitter Network (2.3), Simulation (2.4), and Social Sciences (2.5).

The main part of this thesis is chapter 3 (Methodology). This chapter includes the whole work beginning from the design, heading towards the implementation and usage of the Twitter-Testbed. The design chapter (3.1) will deal with the theoretical background that can be seen like graph theory (3.1.1), characteristics of a social network graph (3.1.2), the graph generation process (3.1.3) and a description of a communication simulation task (3.1.4). Finally, there has also been included a chapter about the input and output interfaces (3.1.5). Also, this chapter explains how to make the generated content accessible in a way that it can serve as test data for other researchers.

A technical description of the implementation can be found in chapter 3.2. Before we head to the detailed description of the program architecture, we will first characterise used resources and other projects like the JUNG framework, graphML and Status.Net (3.2.1 - 3.2.3). Graph generation and communication simulation are separated in the code to allow for use of the simulation feature using existing graphs and so are treated apart from each other in sections 3.2.4 and 3.2.5 respectively. The final chapter deals with input and output functions, which allow access to the test data (3.2.6).

¹¹ IRC (Internet Relay Chat), a former very popular chat system for text messages.

Chapter 4 explains the usage of our application and summarises all parameters that can be set to control the structure of the network and the communication flow.

An evaluation of our work will be done in chapter 5 where conformance to the real networks metrics and figures of usability concerns like speed and memory consumption will be presented. It will be explained if the applications performance can hold for a replacement of the current methods of gathering test data while fulfilling an accordance of structure similarity.

A conclusion is drawn in chapter 6 and a future outlook for some ideas concerning future work will be presented in chapter 7. This work can be seen as a prototype with the ability for further improvements. Together with the ongoing research in social network services, methods used in this thesis should be regularly revised and adapted to new findings and requirements.

Finally the appendix (chapter 8) provides further resources like source code examples and tables of figures.

2 Related Work

2.1 Graph Generation

A representation of various network generation models was presented by Newman et al. [20] who compared a number of models for unipartite and bipartite random networks with data from existing Social Networks. In their work they show that the model of Erdős and Rényi [21] is not well suited for social networks because their method produces a Poisson degree distribution rather than a power-law distribution which most social networks are stated to have. Along with the comparison to the extension by Molloy and Reed [22], they finally come up with their own model focusing on specifying distribution and that their statistical properties are mathematical solvable. Their strategy is to take a number of N vertices and assign to each number a random number k drawn from the desired distribution. The next step is to randomly choose nodes in pairs and form edges between them where every node takes part in this process k times. This way they derive a random graph that has exactly the desired degree distribution and a clustering tendency. They concluded that the difference between clustering coefficients in their random graphs and real networks indicates lack of randomness in real networks.

Kumar et al. [23] published a model to simulate the “flickr” and “Yahoo! 360”¹² networks. He showed remarkable conformance in degrees of nodes. Their extensive study of these networks is presenting in detail several properties for social networks. A biased preferential attachment approach is being used, extended by a categorization of the nodes in “passive” (simply not active), “inviters” (recruiting new members) and “linkers” (full participants) to generate a realistic model. “Inviters” form links to new nodes only whereas “linkers” are connecting to any other node.

A comparison of characteristics of recent random graph models done by Leskovec et al. can be found in their paper [24] and a more detailed revision in their later research work [25] where the temporal evolution of five different networks was studied. Their results were compared with the Community Guided Attachment model and its extensions. In their work they present a new model called the “Forest Fire Model”. This is a simple, intuitive model that requires few parameters and produces graphs exhibiting the full range of properties observed. The Forest Fire Model, which we use as well in our work, will be discussed in detail in chapter 3.1.3 (The Forest Fire Model). In the study of 70 networks of different domains [26], they compared data with 4 graph generation models namely “Preferential Attachment” [27], “Recursively hierarchical community structure” [28], “geometric preferential

¹² A social networking portal operated by Yahoo! Inc., closed in 2009.

attachment” [29] and their own “Forest Fire Model”. By focusing on the envelopment of these graphs they have found, that instead of a growing diameter, most social networks have an effective decreasing diameter over time.

Recently Leskovec et al. introduced a new model focusing on the microscopic node behaviour which leads to a powerful and surprisingly simple approach in [30] where they use Kronecker graphs to generate a realistic network structure.

Sala et al. did a comparison of six different graph generation models in [31], namely Barabasi-Albert [27], Forest Fire [24], Random Walk [32], Nearest Neighbor [32], Kronecker Graphs [30] and dK-graphs [33]. By an approach to fit these models to real graphs this work shows the strengths and weaknesses of each algorithm. They also pointed out that the metrics used in science to describe graphs may not be feasible to rate the success of a graph generation model.

2.2 Properties and Behavioural Patterns of Social Networks

It is important to know how the networks architecture looks like in detail and what the main characteristics are for the communication to happen inside the network in order to simulate its processes appropriate. Here are some works that are useful in this context:

Karagiannis et al. [34] studied the usage characteristics of email service for the design of new features like those known from social network services. They present “behavioural” profiles for the employees of a large multinational corporation. Although there have been several studies about email communication networks, this one is especially interesting because it gives an insight of the causes that drive this communication. Reading this paper gives a deeper understanding of why and under what circumstances a message will be replied and how long it would take for that response to happen.

Centola [35] carried out an interesting experiment about the effects of network structure on the behaviour of the users and the role of social reinforcement. One limitation the author stated was that he could not vary the topological structure independently to test the predictions. Again, this fact point out the necessity of controllable artificial networks for testing purpose.

2.3 Recent Analysis of the Twitter Network

The retrieval and analysis of large complex graphs like Twitter and its competitors is not trivial and would be out of scope for this work. To model the presented graph it was necessary to rely on prior works that studied the structure of these networks. Literature research has shown remarkable few works focusing on the structure of Twitter compared to the popularity and the outstanding differences to other networks.

With 87,897 nodes the work of Java et al. [36] is covering a rather small part of the whole Twitter network today. Despite of this very fact, they present some important values not presented by others to our collection of Twitter's properties. The focus of this work is on the intention of the users and presents the observation of three major user groups named "Information Source", "Friends" and "Information Seeker" according to their intention.

Another important study is presented by Krishnamurthy et al. [37]. They crawled about 100,000 distinct users merging the data of three different crawl attempts and presenting various statistical parameters. Moreover, they also suggested the categorization of users in three main groups (broadcasters, acquaintances and miscreants) which are comparable to the ones mentioned by Java et al. (see above).

The dataset of Huberman et al. [10] consists of 309,749 users. Although they are not presenting many statistical properties, they come up with the idea of "real friends" as a parameter of behaviour of the network instead of relying on the number of followers and followees. They defined a "real friend" "as a person whom the user has directed at least two posts to." [10]

Kwak et al. [14] have made enormous efforts to crawl the Twitter network resulting in an dataset of 41.7 million users. They focused on the evaluation of ranking algorithms and analysed the top trending topics. Their approach is certainly a good starting point in analyzing the Twitter network. The interested reader is referred to their website¹³ which offers some additional information and the full graph (without tweets) to download for free.

Based on our knowledge we can say for sure that up to this point Cha et al. [38] made the biggest effort to collect Twitter's network. They obtained 54 million nodes and including two billion directed "follow-links". Their work focuses on the influence a particular user has on the network. Taking a closer look on the role of "Retweets" (forwarded messages) and "Mentions" (messages containing references to other users). Unfortunately, they did not present properties of the Twitter network structure that we could use in this paper.

13 <http://an.kaist.ac.kr/traces/WWW2010.html>

Because Twitter messages are restricted to the length of 140 characters, a new and unique syntax has emerged to characterize the purpose of the message. Moreover, the tendency to restrict the messages to the most minimal amount of characters possible makes it not always easy to understand the syntax of tweets. In order to understand the syntax that one might come across in some tweets, it is helpful to start off reading two pieces of studies that deal with this aspect of Twitter:

People who are not familiar with the following symbols and their meaning “RT”, “@”, “#” and “☺” should definitely take a look at the works of Boyd et al. [19] and their research on the controversial aspects of Twitter focusing on retweets. It gives the reader an overview of what kind of syntax conventions the community of the service has built since its start-up. Using a series of case studies and empirical data Boyd et al. present the conventions used by the community and also discuss the background of the special syntax of Twitter's users.

Honeycutt et al. [39] investigated the usability of Twitter for collaborations and therefore focused specially on the @-symbol. Used to mark other users in a message, the “@” stands for an indicator of possible collaborations. In their work they present an overview of the possible usage patterns of the @-symbol in tweets.

2.4 Simulation

One very related paper to this thesis is the one of Harald Psaiet et al. [40]. They did a simulation of a service-oriented collaboration network in order to evaluate the efficiency of similarity-based adaptation in a virtual team of a crowd of task-based services. The work presents some strategies and evaluations of the performance of the overall system when different strategies are applied.

Page and Kreutzer describe in their work “The Java Simulation Handbook” [41] strategies for simulation systems of various kinds. Although our work is certainly not a simulation with regards to the authors' understanding of this topic, we could benefit especially during the design phase from the considerations drawn in this book. It provides insights into the design and implementation perspectives of discrete and continuous event simulation. Moreover, it is coupled with DESMO-J¹⁴, a freely available event simulation tool written in Java and a website offering a tutorial and source code examples.

2.5 Social Sciences

The field of social sciences has gained considerable amount of attention recently. Due to the availability of large data sets (e.g., Twitter, Facebook, MySpace, ...), social scientists were able to study more in detail the relationships and behavioral aspects between people. For

¹⁴ <http://desmoj.sourceforge.net/>

instance, Backstrom et al. [15] provide a detailed discussion on influential factors for group formation and community growth. In their work they emphasize the social influences (e.g., number of friends, relation structure, ...) that have impact on the network. These revelations are important for graph generation that aim to create realistic models of social networks.

3 Methodology

3.1 Design

As mentioned in the previous chapter “Motivation” (1.1), this paper should help the development process in areas related to social networks. The main target is to create a network that is customizable through some input parameters and to store that network in a file for later inspections. Further, it should be possible to generate a simulation of communication and save the output in a separate file.

For our purposes we need first of all to generate a network that is random while matching predefined statistical properties. Because all other parts of the application rely on the right design of the graph, this part has to be taken into special consideration. The first step is to find out what properties are crucial for the Twitter network. A literature research has been conducted with the conclusion that works about graph generation (see chapter 2.1) and Twitter (chapter 2.3) are available but unfortunately no works on graph generation of a network similar to Twitter. This way, we are able to say that our approach is the first one to generate a Twitter-like network structure at random.

We will start off this chapter with some basic graph theory in 3.1.1. We will first have a look at the differences between existing graph types. Next, we will discuss graph metrics that are used to compare different network structures. This chapter also focuses on mathematical definitions of used metrics throughout this paper like reciprocity, clustering coefficient and diameter.

The properties of Twitter that can be found in literature research are presented in section 3.1.2. These parameters will give us a model of how the generated network should look like. The goal is to get as close as possible to those values to represent the network satisfyingly, therefore the network generation model has to feature similar structural properties as the Twitter network. We will take a look at the network structure and Twitter’s characteristics and list all properties we found about the real network. In order to achieve realistic results, each of these properties have to be interpreted and decisions of the importance of each of these values has to be made.

As stated above, it is our firm belief that this is the first attempt to rebuild a Twitter-like network structure. Therefore it is our goal to design this model as flexible as possible, also because we want further improvements on this model to be configured easily. For the generation of a random graph structure we chose the Forest Fire Model by Leskovec et al. [24] which, in comparison with other network generation models, focuses on social networks. Moreover, the Forest Fire Model is more convincing due to its intuitive process of node

attachment and the promised properties of the expected networks it is able to create. Additionally, the burning process from node to node seems to be perfect for adaptations when trying to entirely fit the model to the real Twitter network. A description of the Forest Fire Model and its extensions will be presented in this chapter (3.1.3 The Forest Fire Model).

We will also describe the design process of the simulation system and the decisions we made along the way to get to a satisfying result. While the design of the graph generation process is mostly determined by the Forest Fire Model, we had to develop a social network simulation flow from scratch. In chapter 3.1.4 we will deal with the specification of components that allow for a synthetic message flow, including a scheduler as the heart of the process. We also describe regulation algorithms that are responsible for the network's behaviour.

3.1.1 Graph Theory Basics

In order to better understand the terms used in the following chapters, it is crucial to know some graph theory basics. The intention of this chapter is to give readers who are not familiar with graph theories a basic insight into this matter. It is important to understand the basics in order to understand the concepts that are about to follow in the upcoming chapters. Experienced readers may skip this chapter entirely or jump to the part that revolves around topics they are less familiar with.

In the fields of mathematics and computer science a graph is a set of objects (referred to as nodes) and their relations (referred to as edges). In theory they are often visualised as dots or circles for the nodes and lines for edges. A wide range of relationships in real life can be represented by graphs like public transportation (stations, paths), street maps (junctions, streets), company employees (employees, supervisor relationships) and many more.

The mathematical definition of a graph G is a pair of a set of vertices V and a set of edges E where E consists of 2-element subsets of $V = \{u, v\}$.

$$G=(V, E)$$

The order of a graph is defined as the number of vertices $|V|$ and its size is $|E|$, the number of edges. The number of edges connected to a node is called its "Degree" and can be divided into in- and out-degree if the underlying graph is directed. Otherwise, in an undirected graph, an edge has no starting or finishing point and connects two nodes in both directions.

Distinction of Graph Types

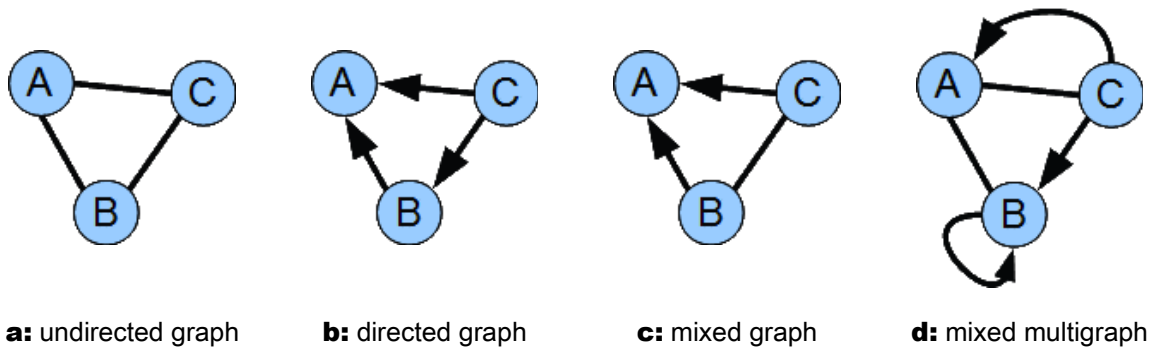


Figure 3: Graph types

Graphs can be differentiated in various ways and the definitions are not always consistent. Figure 3 shows the most prominent criteria that are often found in literature. Without further specifications, a graph is usually meant to be an undirected simple graph (a). In contrast to undirected graphs (a), it is possible to have a relation only going one way (for example from node A to node B) and not vice versa like in a directed graph (b). Mixed graphs (c) allow both edge types to be used and multigraphs (d) even permit more than one relation between two nodes (A,C).

Nodes and edges might have one or more properties. Edges that are holding a property value are often called “weighted” edges. For example sample (d) in the graphs of Figure 3 can also contain self loops. Although the definitions of so called “simple” graphs may vary, “simple graphs” are mostly meant to disallow loops, whereas “multigraphs” might or might not permit them.

Graph Characteristics

There are a number of statistical measures that serve as indices for specific network characteristics in order to compare graphs with each other. Still, it is not clear to what extent these values are able to measure the similarity of structures. Because of the complexity of graphs, one can assume that a graph conforms to one statistical measure while having a completely different structure. We will present some of the popular graph metrics that are also used in this thesis.

Degree Distribution

One of the most basic characteristics of a graph is its degree distribution $P(k)$, defined as the fraction of nodes having degree k . Directed graphs also have the distributions $P(k^{\text{in}})$ for in-degrees and $P(k^{\text{out}})$ for out-degrees that are calculated the same way, but considering only edges that are pointing in the related direction.

A visual comparison can now be drawn by plotting the degree distributions or by calculating the moments (mean, variance, skewness, ...) and their comparison. In most social networks there are a few nodes with a very high degree and a lot with a small degree. Figure 4 shows a typical degree distribution following power law.

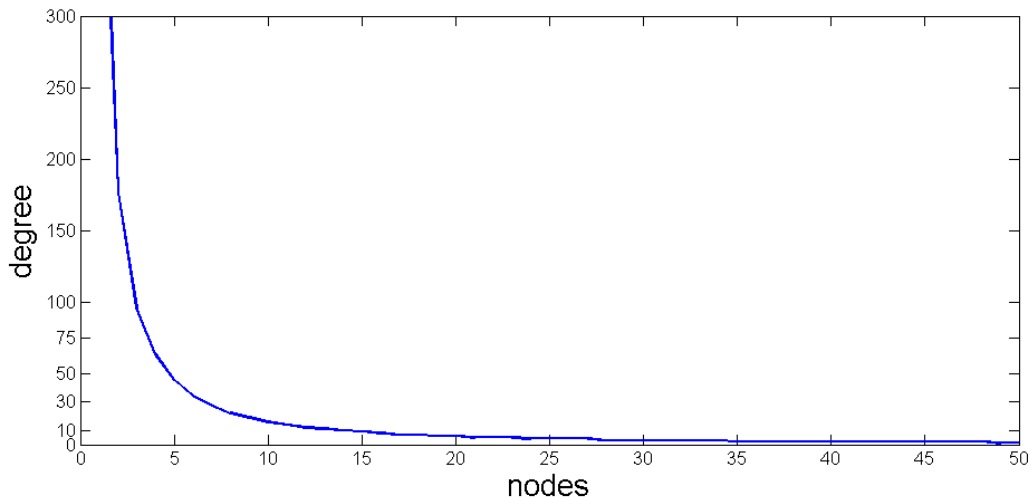


Figure 4: Sample power law degree distribution

Shortest Path

The shortest path is an important metric for information transportation in a network. It is defined as the minimum number of hops on a possible path from node A to node B. All shortest paths of a network can be represented by a matrix $d_{i,j}$ holding the shortest paths from each node i to every other node j .

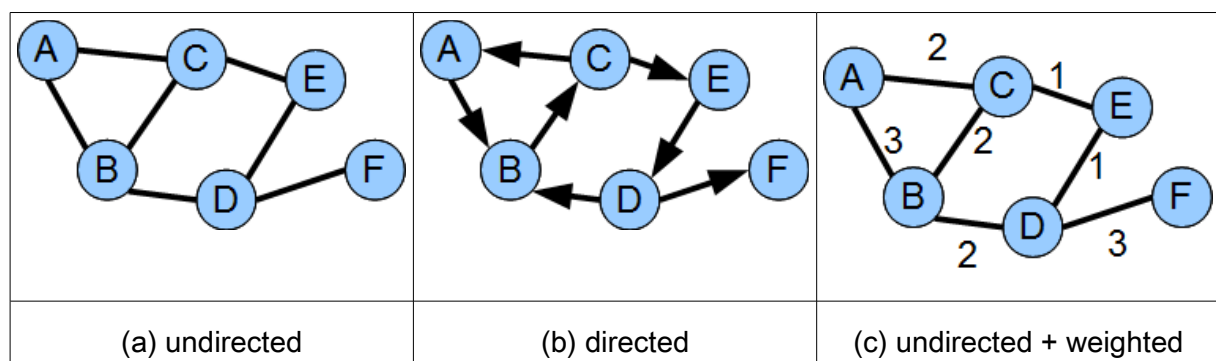


Figure 5: Shortest Path calculation samples

The calculation of a shortest path is depending on the type of graph. We will describe the calculation using the examples in Figure 5. If edges are unweighted (a, b) then every edge has equal costs of 1. To get the shortest path, one has to find a valid connection that delivers the minimum summation of costs. The shortest path from A to F in (a) is $\{A,B,D,F\} = 3$. For the directed graph (b) there is only one possible path from A to F $\{A,B,C,E,D,F\} = 5$. In example (c) we have to consider the weights of the edges to calculate the costs the shortest path is now $\{A,C,E,D,F\} = 7$.

Average Shortest Path

The average shortest path length L , also known as characteristic path or average path length is defined as the mean over all shortest paths (see above) of a network. It is a typical measure for the separation between nodes of the graph and therefore an indicator for the efficiency of information transport on a network.

$$L = \frac{1}{n(n-1)} \cdot \sum_{i,j \in N, i \neq j} d_{i,j}$$

Diameter

The diameter D is a measure of connectivity. It is the greatest distance between any pair of nodes in the graph and calculated as the maximum of all shortest paths. To calculate the diameter, one has to calculate $d_{i,j}$, the shortest paths of all nodes first (see above).

$$D = \max\{d_{i,j}\}$$

Clustering Coefficient

Watts and Strogatz introduced the clustering coefficient in 1998 [42] to compare random networks. It is a measure of “cliquishness of a typical neighborhood”. The clustering coefficient of a node is defined as the proportion of existing edges between the neighbours of a node and all possible edges between them. The overall clustering coefficient of a graph is defined by the average of the clustering coefficients of all nodes.

Let G_i be the subgraph of connected neighbours of node i and e_i the edges in G_i . The local clustering coefficient c_i of node i is the ratio between e_i and $k_i(k_i-1)/2$, the maximum number of possible edges in G_i . Then the clustering coefficient of a graph is given by the average of all local clustering coefficients:

$$C = \langle c \rangle = \frac{1}{N} \cdot \sum_{i \in N} c_i$$

Three sample calculations of a graph's clustering coefficient are visualised in Figure 6, showing three simple graph structures G_i , together with the corresponding clustering coefficient c_i .

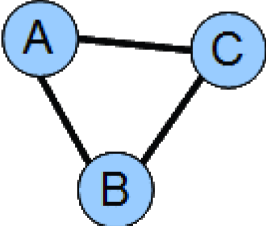
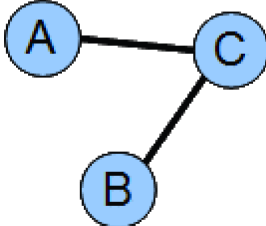
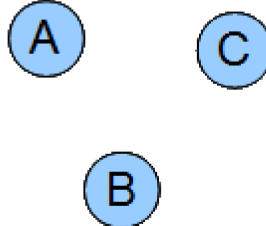
G_i			
c_i	$3/3=1$	$2/3=0.\dot{6}$	$0/3=0$

Figure 6: Calculation of the local clustering coefficient

Reciprocity

A vertex pair $\{A, B\}$ is said to be reciprocal if there are edges $e_{i,j}$ between them in both directions ($\exists(e_{i,j} \wedge e_{j,i})$). The reciprocity of a directed graph is the proportion of all possible $\{A, B\}$ pairs which are reciprocal, provided there is at least one edge between A and B . The reciprocity of an empty graph is undefined. Undirected graphs always have a reciprocity of 1.0 unless they are empty. This measure provides an indicator of the relation between nodes because friends tend to reciprocate where unknown people normally do not link back to every follower. A calculation example can be seen in Figure 7, given 3 sample graphs G_i and the accurate reciprocity value r .

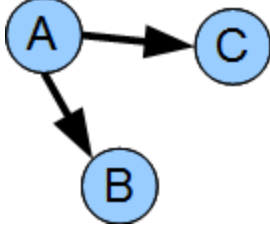
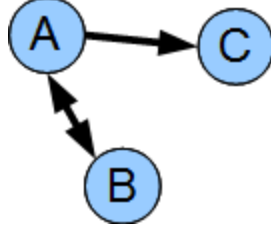
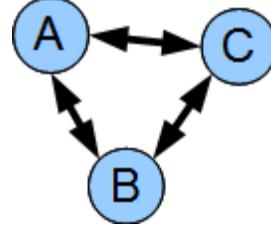
G_i			
r	$0/2=0$	$1/2=0.5$	$3/3=1$

Figure 7: Calculation of reciprocity

Power law

Power law distributions have gained attention over the last years because they can be found in diverse range of natural phenomena. A quantity x obeys a power law if it is drawn from a probability distribution

$$p(x) \propto x^{-\alpha},$$

where α is a constant parameter known as the power law exponent.[43]

If you plot a power law distribution with both axes using a logarithmic scale it, draws a straight line with the slope of the power law exponent. Figure 8 shows a pseudo-randomly-distributed power law distribution (blue circles) and a reference line for a slope of -2.5.

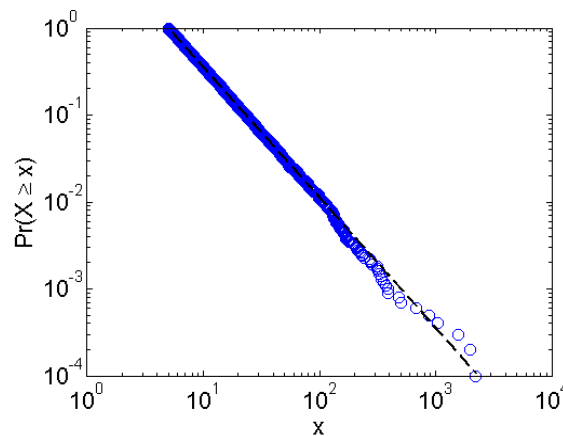


Figure 8: Visual power law fit

Although verifying a power law by fitting the logarithmic plot is a common used method, it is not mathematically accurate. Clauset et al. have published a guide in which they “describe in detail a set of statistical techniques that allow one to reach conclusions like these” [43]. For fitting a power law they calculate the goodness-of-fit and a likelihood ratio test for a statistical probability of the hypothesis. The whole calculations are complex and extensively described in their paper so the interested reader is referred to their work for details. On their website they offer Matlab¹⁵ functions to download that aid in calculating and plotting a power law fit.

3.1.2 Characteristics of the Twitter Network

In order to create a graph similar to a real world network, one has to get the specifications about the underlying structure of the desired network. We had to rely on the small number of works already done to analyse the structure of Twitter due to various reasons:

First of all, Twitter does not provide their data to research institutes due to their privacy policies. Moreover, they do not even offer any official up-to-date statistics about the size of the network or the overall user activities available as it is the case for other social services. The only way to get these properties is to gather the structure by crawling the public available Twitter graph. Due to the size of the network, it is challenging and it would take weeks to get the data. Even trying to crawl the whole graph would not be sufficient because calculations of graph metrics done with millions of nodes would take normal computer systems to the limit of their technological capacities. Crawling and analysing data of a more than 200 million nodes network is outside of the scope of this paper. Therefore, we will only work with published literature that provide us with the necessary information.

¹⁵ Matlab is a numerical computation environment developed by MathWorks Inc.

To our knowledge the biggest effort to collect Twitter's network up until now has been made by Cha et al. [38]. They obtained 54 million nodes with two billion directed follow-links among them. Albeit, this is only a fourth of the whole 200 million users Twitter has today, still an armada of 58 white listed machines and certainly at least a month continuous crawl attempts where needed to achieve this goal.

In Kwak et al.'s data [14] there are 41.7 million users connected with each other by 1.47 billion relations. They gathered a communication set of 106 million tweets containing 4,262 trending topics¹⁶. To obtain this dataset, Kwak et al. did a breadth-first crawl beginning with a very popular person and extended this connected component with user profiles that referred to trending topics. They used 20 white-listed machines in a two month continuous crawl attempt to extract the data, resulting in a 24 GB graph file. Although this data is available for other researchers, it is not easy to calculate statistics of datasets that large. We have tried to build some metrics for the provided graphs ourselves but did not have success until recently. Unfortunately, although Kwak et al. got one of the biggest copies of the Twitter network, they only present a few metrics about the graphs structure. We believe that this is because they might also have reached their limits of capacities.

As described in chapter 2.3 it was just possible to find three works revolving around listing properties of the Twitter graph. The hard facts about them are listed below:

The dataset of Java et al. [36] consisted of 1,348,543 posts, 829,053 links and 87,897 users in two months. This data was gathered by monitoring status updates on the overall public timeline and fetching their relations to other users with the Twitter API.

Huberman et al. [10] had 309,740 users, leaving the reader in the dark about how they were chosen. A calculated number of 78,983,700 status updates was concluded from the given average number of posts. These users are connected by 51,107,100 links, again, calculated by the average numbers of followers and followees.

Table 2 lists all important features of the Twitter network gathered from the sources above.

¹⁶ Most popular topics that are talked about in a defined time-frame.

Parameter	Value (Source)	Source
Avg. number of Followers	85	[10]
Avg. number of Followees	80	[10]
Average degree	18,86	[36]
In-degree slope	-2,4	[36]
Out-degree slope	-2,4	[36]
Degree correlation	0.59	[36]
Diameter	6	[36]
Clustering coefficient	0.11	[36]
Reciprocity	0.58	[36]
Reciprocity	22.1%	[14]
Average path length	4.1	[14]

Table 2: Figures of the Twitter network

3.1.3 The Forest Fire Model

In order to find a suited network generation model, we did another literature research. Although there are plenty of models in literature, only some of them are focusing on social networks. The models we found are briefly described in chapter 2.1. The Forest Fire Model by Leskovec et al. [24], claims to fulfil all characteristics we have collected about the Twitter network. Therefore, it seems to be best suited for this project. It has an intuitive straightforward way of building the network structure and can be easily extended. Matching the power-law exponent of -2,4 of Twitter, it has a heavy-tailed in and out degree. Communities are built where nodes are more connected to each other than to the rest of the network, corresponding to the clustering coefficient of 0.11 we found. Moreover, the model has a densification power law and a shrinking diameter, meaning that the network becomes denser, and the diameter is decreasing as the network grows.

The Forest Fire Model was first published in [24] and revised in [25], where they observed snapshots of several social networks to inspect their temporal evolution. Comparing their observations with existing models like the “Community Guided Attachment model” and its extension the “Dynamic Community Guided Attachment model”, Leskovec et al. have found out that these models do not capture all the properties they observed. This very fact lead them to the creation of a new model – The Forest Fire Model.

This model is based on having new nodes attach to the network by “burning” through existing edges in epidemic fashion. Nodes arrive one at a time where every new node v gets attached to an existing node w (called ambassador) and starts burning through w 's

neighbours. Node v then links to the neighbours of w with a certain probability (p) and further continues the same burning strategy on those newly discovered nodes and their neighbours. In this way, v burns from one node to another like a forest fire that spreads from tree to tree.

This process is intuitively corresponding to the way new people usually discover a social network: A new person (represented by node v) will most certainly be invited by a friend (the ambassador w), connect to it and look, if it can find other known people in the crowd w is connected to. If some interesting / known person u could be found on ambassador w 's friend's list, the new person v will probably look up the list of u 's friends and so on.

The most basic version of the model can be formalized as follows:

Every time $t > 1$ a new node v gets added to the graph G_t it has to follow the following rules:

- 1) choose an ambassador w uniformly at random and connect to it to v .
- 2) Generate two geometrically distributed random numbers x and y with means

$$\frac{p}{1-p} \text{ and } \frac{rp}{1-rp} \text{ respectively.}$$

- 3) Node v chooses up to x out-links and up to y in-links (not visited yet) of $w(w_1, w_2, \dots, w_{x+y})$ randomly and connects itself to the nodes on the other side.
- 4) Apply steps 2 and 3 recursively to each of w_1, w_2, \dots, w_{x+y}

Note that in the whole process each node can only be visited once, preventing the construction from cycling.

This way the model naturally builds *communities* because nodes closer to the ambassador have a higher probability getting linked than those far away. In the same way the *Densification Power Law* comes in as the number of links a newcomer creates per discovered node will drop out rapidly with the distance from the ambassador. *Heavy tailed out-degrees* arise from the fact that highly linked nodes can easily be reached by a newcomer, no matter where the process starts.

As these properties descriptions are quite intuitive and easy to explain, the Forest Fire Model also provides a shrinking diameter. Although the authors stated that it is hard to provide an explicit explanation, they have shown in simulations that the model is capable of producing sparse or dense graphs with effective diameters that either increase or decrease, while also producing power-law in- and out-degree distributions, by varying the forward and backward burning parameters (p and r).

Extensions of the Model

The basic model features all expected parameters of a general social network but by extending the model it can capture observed data of a given network even more closely. Leskovec et al. already proposed to extend their own model with orphans, nodes that are not connected to any other node. This can be achieved by either initializing the graph with a set of unconnected nodes or providing some probability to which a newcomer will not form any links (not even to its ambassador).

In our model we will create orphans by initialising the graph with a variable number of unconnected nodes that can be set in the properties file and default to one.

We extended the model by adding a back-link-probability as well which gives each discovered node the possibility to form a link back to the newcomer that links to it. In this way the effective in- and out-degrees can be better controlled to fit the desired network.

Leskovec et al. [24] also experimented with the idea of “multiple ambassadors”, where a newcomer can choose more than one starting point. This extension should accentuate the decrease in effective diameter over time as nodes with multiple ambassadors are bringing far-apart parts of the graph together. Although we didn't use more than one ambassador for the presented graphs, it is still an interesting modification worth experimenting with in future works. We have implemented the function in the application. In order to invoke it, one has to make a change in the settings only (see 4 Application Usage).

3.1.4 Communication Simulation

The initial specification of our application was quite simple:

It should be able to automatically generate a random albeit representative network structure (graph) of variable size, store it into a file and “send” some random messages from node to node along the edges of the graph, thereby producing a protocol of “who tweeted what and when”. This protocol should then be the input to any testing algorithm and the outcome could be compared with the real network-structure in the graph file. This schema is illustrated in Figure 9.

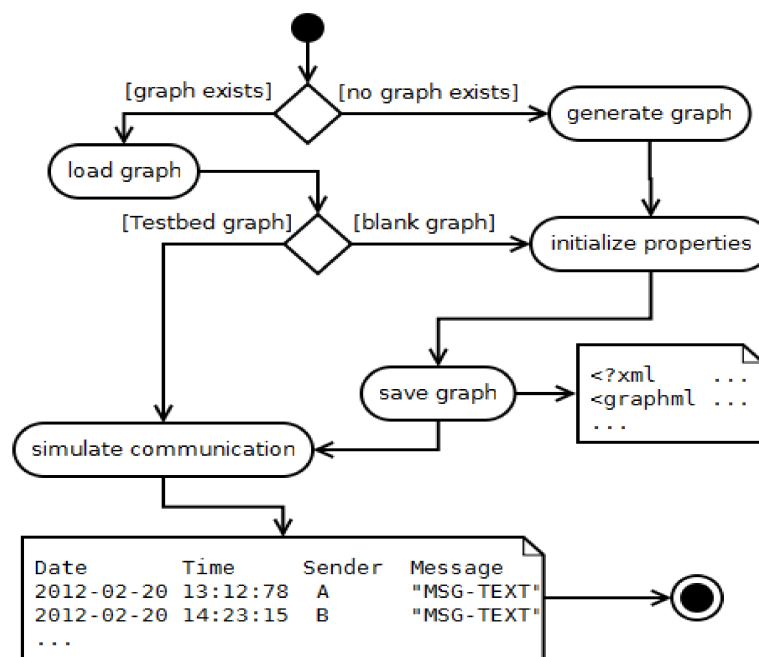


Figure 9: Twitter-Testbed schema UML diagram

Once the network is established it should be able to generate an output similar to Twitter, which is basically the so called “timeline”. The “timeline” basically are messages of the users in a continuous time flow. To generate the timeline, a simulation has to be done where the action (message) from one node triggers a reaction from another node.

At first it seemed feasible to create one thread for each node which then listens to messages and has the ability to react in a certain way. While this would be a realistic representation of these processes in real life, that would require a lot of memory and CPU for thousands of nodes. In fact, only a small number of all nodes will post frequently and a certain number might do nothing at all but consuming memory space. Besides that, one has to face the challenge of multi-threading issues.

We realised that what we needed is a scheduler that can hold events in a timely ordered list and execute one event after the other. If one node would for example send a message, it is actually adding the event “sendMessage” to the scheduler with the time of its expected execution. The scheduler would execute this event after all other events that had stated an earlier execution-time have been executed. This way we will not have any problem with memory consumption and CPU, especially as the communication doesn't need to be “real-time”.

Scheduler

The scheduler is the heart of our simulation process. It is holding all events in a timely ordered list and is responsible for the execution of the event's actions. Start and end of the simulation are controlled by the scheduler and it is also the docking point for output interfaces. Figure 10 shows a schematic view of how the process flow works.

To start a simulation, the scheduler has to be initialised by a number of start events. Therefore, we will iterate over all nodes in the graph and produce a starting event for each of them. In the case of Twitter this is a normal tweet. The crucial element is the time of this starting event that could possibly also be beyond the scope of the scheduler's time limits and would result in a node that is not actively tweeting. Still, this node can react passively on other updates.

After a list of events has been created in the initialisation, the scheduler has to iterate over each element and trigger one event after another, removing it from the queue afterwards. This process will continue until a stop criteria is met, which is either an empty queue or the exceeding of the time limit. To prevent the process from dying out after the first tweet of each node, every tweet recreates the next update of the respective node. Whilst the simulation runs reactions to tweets can also be added to the queue by the receivers.

A similar procedure has also been used in the Java Simulation Handbook [41], where some of the concepts adapted for this project derive from. The introduced scheduler therein and the code examples of DESMO-J, a simulation framework developed by the authors, have been helpful as a boilerplate and were partly used in our code as well.

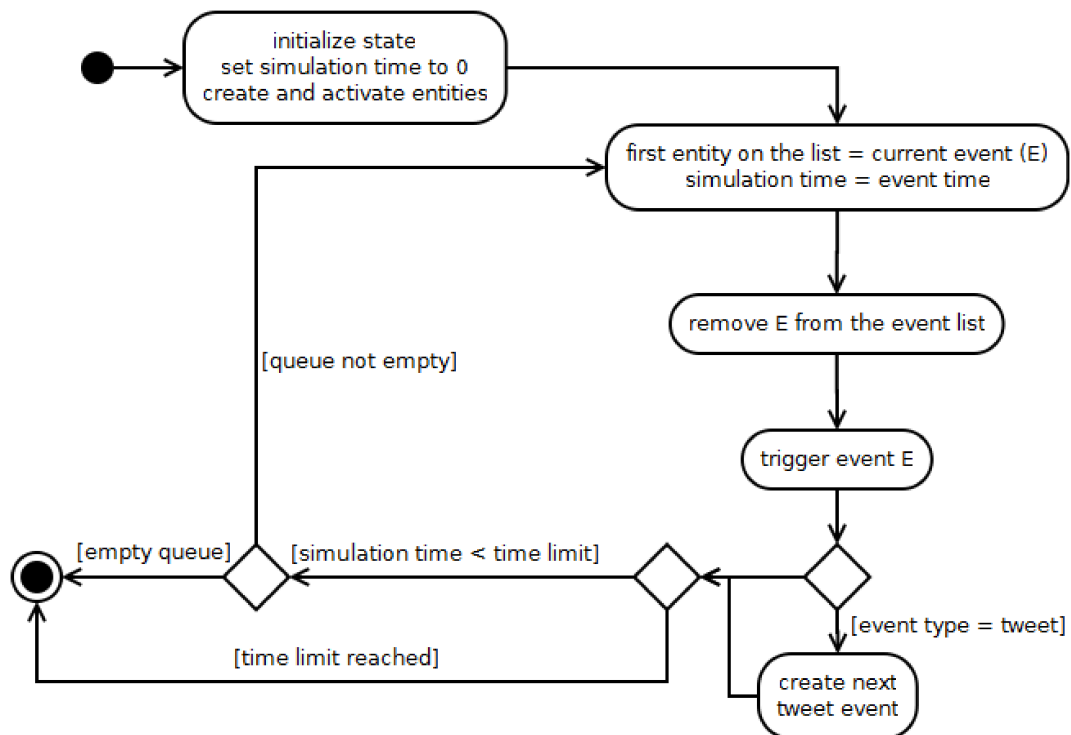


Figure 10: Scheduler (UML-diagram)

Furthermore, every event type has to provide a function that can be called by the scheduler to start the appropriate reactions. For example, considering a normal tweet this method has to call every node with an incoming edge (follower) and calculate if a reaction has to be set and if it does, what type of reaction this should be.

Event Types

After the concept of the operating procedure, one has to figure out the types of events that have to be implemented. Posting an update in Twitter is commonly referred to as “tweeting”. Besides, it is also possible to send private messages to other users. These messages do not exhibit the limitation of 140 characters and will not be part of the time line. As private messages are not visible to the public, no statistics concerning their frequency of occurrence could be found in literature. Because they are addressing one particular user, they do not play a major role in scientific investigations concerning the network structure. Therefore, we decided to skip the topic of private messages entirely and focus on the tweet events.

Based on the intention of a tweet, updates can be categorized into three types: Normal tweets, retweets and replys. All of them have the same format limitations and can only be differentiated by the occurrence of some special syntax.

Tweets are propagated to all followers and visible to any user of the network. They are meant to make status updates. Replies and retweets are in fact tweets including some special syntax to signalize their special meaning. Moreover, the difference between tweets, retweets and replies only emerges at the point where the receiver is notified.

Retweeting is the Twitter-term for what is called “forwarding” in email communication. The retweeting of interesting tweets to one's own followers is a crucial operation for the Twitter system because it is what allows a message to spread into the network and reach a bigger audience. In fact, every tweet that includes the quintessence of another tweet can be referred to as a retweet. To clarify that this information is the creation of another user, a special syntax has emerged. In the same way as forwarded emails in email communication are usually labeled with “FW:” or “Forward:” a retweet starts with “RT @Username” or “Retweet @Username” mentioning the cited user. This way a former received message spreads to all followers of the receiver and the sender gets a notification that one of their messages has been used for a retweet.

Replies start with “@Username” and will have the user “Username” get a special notification. After a change of Twitter's policies in 2009, replies with the mentioned syntax will only get passed along with followers of “Username” and the person replying. To prevent this from happening one can put a “.” (or any other sign) in front of the message and it will pass to all followers while still notifying the mentioned user. Usually this operation is called “**.reply**”. Although this practice is meant to answer to someone's tweet it bears not necessarily the intention of actually replying to something in order to use this syntax. Generally speaking, it is used any time someone wants to directly address a particular user in a public update. For this reason we will rather refer to this type of message with the more general terms “**directed tweet**” or “**directed message**”.

As most of these conventions grew with the network, there are several modified versions of the basic syntax that was mentioned above. Some modifications aim to encourage and to leave enough characters for the message and possible additions of other users. Others target the readability and understanding of messages. Because they are just conventions for the user, everyone is free to use one of the available styles or even invent their own. Boyd et al. have studied the conversational aspects of retweeting on Twitter [19]. They present an insight into their findings of the most popular methods used to edit, comment and relay retweets. Honeycutt and Herring have worked on the usage patterns of the @ sign in Twitter messages, which is an indicator for replies and mentions [39]. For further information about Twitter's common conventions, interested readers are encouraged to take a look at their works.

In order to filter out the individual types from the text of tweets, the following rules have to be obeyed:

- 1) Messages starting with “@username” are considered to be direct tweets or replies, dependent on prior messages.
- 2) A message can be considered a retweet if and only if it is not starting with “@” and contains one of the retweet keywords:
“RT: @”, “retweeting @”, “retweet @”, “(via @)”, “RT (via @)”, “thx @”, “HT @”, “r @”, “🔄 @”, ... [19]
- 3) If the message has no one of the above mentioned indicators but still includes mentionings (“@username”) somewhere in the text it is considered to be a mention.
- 4) Everything else is a normal tweet.

As we are generating all messages by ourselves, it is not necessary to parse each message in order to extract the keywords. Instead, each message type will also have its own class and therefore can easily be distinguished from other types. We will also add additional information to the message types objects, like the ID of the message a reply is an answer of or, in the case of retweets, their original message which they relay.

Actually, all of the message types mentioned are just normal tweets considering their text content. The difference emerges when it comes to the intention of the sender and the reaction of the receiver. As normal tweets are meant for anybody who cares to read them, they will only get a reaction if the message really matters the receiver. It has been found out that retweets have a higher chance to get retweeted again than normal tweets (“Once retweeted, a tweet gets retweeted almost instantly on the 2nd, 3rd, and 4th hops away from the source, ...” [14]). The only difference between the diverse message types is the inclusion of special keywords like for example “Retweet” or “RT” at the beginning of a message identifying a retweet. That also means that the message types can be mixed. Like in Listing 1, it is possible to retweet one's tweet and mention another one to give them a special notice of what might interest them.

```
RT @twitteruser Japanizing English - a book by Johnannes Perling @someuser
certainly a good read :-)
```

Listing 1: Example update text of a mixed message type

In our model we have defined 5 message types to cover all forms of tweets:

- **tweets:** Non of the retweet keywords or “@user” included
 - **mentions:** includes one or more “@user” keywords
 - **directed tweets:** begins with “@user” to address that user
 - **replies:** begins with “@user” and is related to a prior message
 - **retweets:** includes at least one retweet keyword and probably also “@user”

These message types build a hierarchy where each child element is also of the type of its parent. For example: a retweet is a mention and a tweet. If it is a mixed type, children count before parents because they are more precise and directed tweets count more than retweets because they limit the audience.

Obviously, the tweet event is the fundamental messaging event. “Tweeting” is the Twitter name for sending a status update to all followers. This event has to hold a message, the ID of the sender and the time of action. If such an event gets triggered it must initiate a process that alerts every node having an incoming connection from the sender of the message so that they can react and trigger a corresponding event (like a reply) themselves. How this process is achieved will be thoroughly discussed in the upcoming chapter (Network Behaviour). Figure 11 shows the process of message sending that represents a detailed view of the “trigger event E” state in Figure 10.

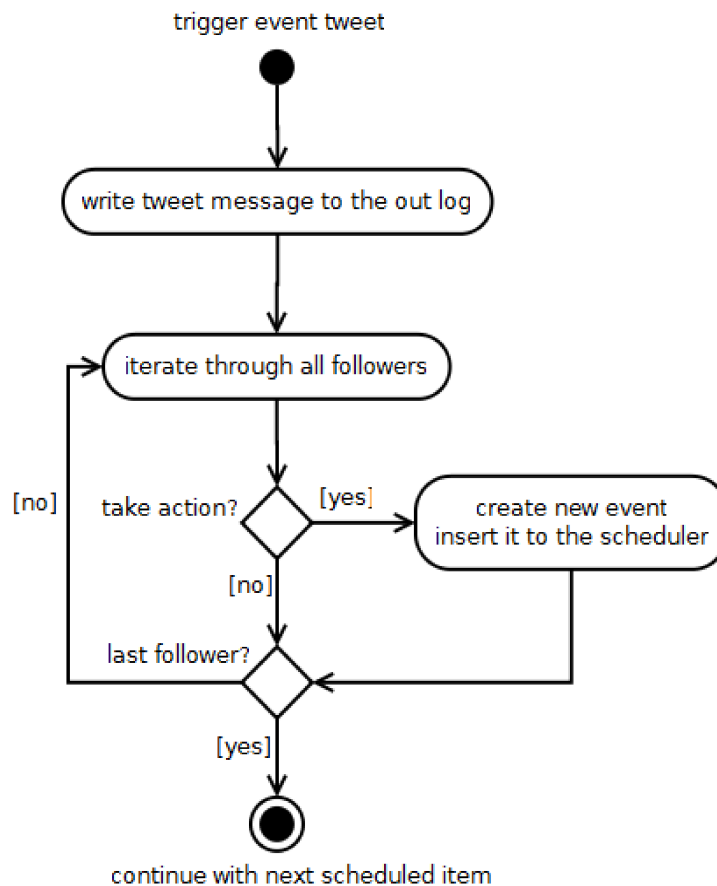


Figure 11: UML: Scheduler - trigger tweet event

Almost the same procedure has to be done for the other message types respectively. It is exactly the same chain for retweets apart from another message text and the inclusion of an original message reference. The only difference is the reaction pattern on the receiver's side.

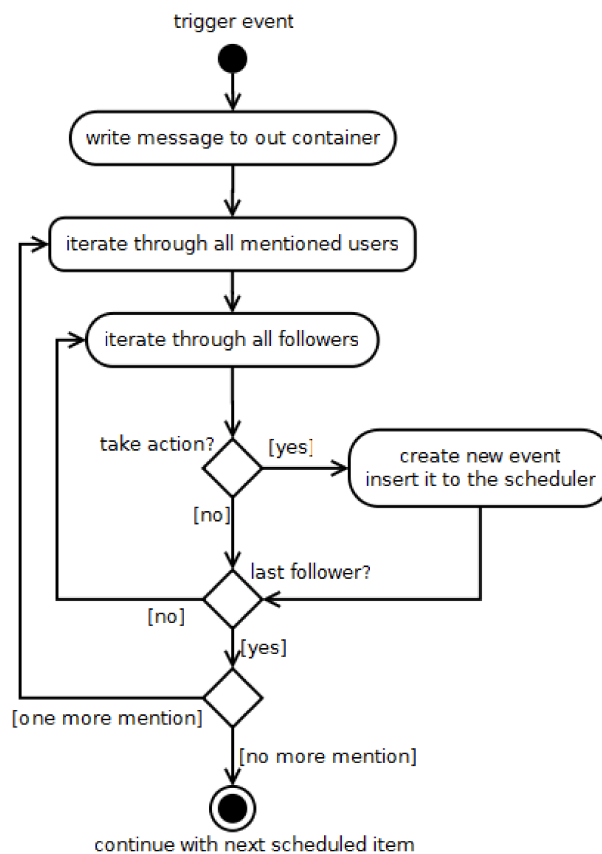


Figure 12: UML: Scheduler - trigger reply event

Other message types vary only in regards of their audience, which changes slightly. Mentions for example have to alert the mentioned user in a special way and get propagated to all other users as normal tweets. Replies and directed tweets are not visible to all of the sender's followers but to all mentioned users in the message and their followers. So actually the "iterate through all followers" step has to be done for each of the receivers instead for the sender (Figure 12).

Network Behaviour

The question we are facing is "When does an event trigger". In the simulation the essential challenge is to determine at what time what kind of event has to be triggered. This seems to be quite obvious at first glance but it gets trickier the more we go into detail. To make a 100% accurate simulation of the communication in real SNS, an almost infinite number of parameters would have to be studied and implemented. Besides, it is fairly impossible to get figures to every potential parameter and trying to pack all that data into the model would result in an infeasible large amount of useless data. The question we therefore asked ourselves is "What parameters are really crucial for the simulation when it comes down to the lowest possible level". Our goal was to find a maximum of 5 parameters to be stored in the graphs properties that define the behaviour of our nodes.

The first parameter we invented was a **closeness value**. All reactions of a person to a message depend to some degree on the level of closeness. For example, it is more likely to respond to the same message if it comes from a close friend rather than if it comes from someone they haven't met yet.

As already mentioned before, we can always go into more detail with the parameters used. For example, the closeness value can be seen as an aggregation of several other parameters like for example: “how well node a knows node b”, “how node a likes node b”, “what type of relationship node a has to node b”. But as we don't want to store all these information and we want to keep the model as simple as possible, we aggregated all these possible parameters in the parameter of closeness ranging from 0 to 1. Zero means there is no relation at all, these nodes don't know each other; possibly connected by mistake. An closeness value of 1 means the opposite, these nodes act like twins, reacting to any message of each other. So the higher the closeness value, the more likely it is to get a response.

Another parameter often seen in literature is the nodes usage intention. In that case, there are three different groups presented. In [36] and [37] they are named [“Information source”, Friends”, “Information Seeker”] and [“broadcasters”, “acquaintance”, “miscreants”] respectively. Whereas the first two groups can be considered as identical, the last ones might not necessarily be identical. Anyway, these groups can be calculated by the degree correlation. For our “**activity type**” property we will follow the terms used by Java et al. but replacing “Friends” with “normal user”.

type	property
Information source	Significant higher in-degree
Information seeker	Significant higher out-degree
Normal user	in- and out-degree almost equal

Table 3: Twitter user's activity types

Theoretically, we could calculate this parameter using in- and out-degree, but as we don't want it to be solely determined by the ratio of degrees (because it could be possible that even a famous person is only an information seeker) and we also want that every parameter can be stored in the graph-file for reproducibility. We will store this property within the node object.

Each node, in either of the user-type groups, can be more or less active. As the activity type defines the ratio between event types evoked by this node, we need a value that stands for the frequency of new updates. This parameter does not depend on the number of followers

though it may be influenced by it. We will add the parameter “**activity level**” again, ranging from 0 to 1. The higher the activity level, the more tweets will be send and the more possible reactions will actually be triggered by this node.

Decision-Making Process

Now we should be able to calculate a conditional probability of IF a node triggers an event. We are doing this by taking the closeness value, the activity type and the activity level into account. To trigger an appropriate action we additionally need to know the exact time WHEN the event should take place. Thus, a parameter is required that allows us to determine the nodes frequency of looking up other's new updates – or in other words: when the user is available / using the social network. Therefore we invented the “**availability level**”.

Availability level is also ranging from 0 to 1 and it tells us how often a node is willing to interact with the network. This way it is possible that a person can react to every tweet (high activity level) but with a long delay (low availability level) and vice versa.

Users can also be classified into time slots of usage. Not every individual uses the service 24/7. Each person has rather their own preferences as to when they want to use a social network. On the one hand there are users who are only looking for new updates in the evening after works, while others may prefer to check the latest news in the morning on their way to work. Some people may only make use of Twitter at weekends and holidays because that's when they have time to do so and there might be also other users who utilise Twitter for their work and do not want to use it in their free time.

The first step is to implement the “**availability type**” on a predefined day-type basis for each user, dividing them into weekend, weekday or all-day users.

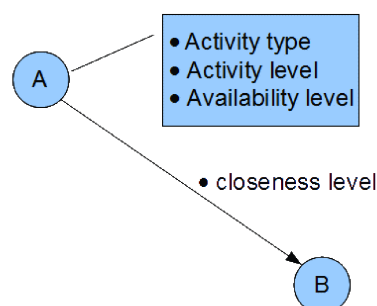


Figure 13: Properties of nodes and edges

To sum up, we have four properties that mainly influence the communication in the system: the closeness value, the activity type, the activity value and the availability value. As can be seen in Figure 13 The closeness value is a property of the edge whereas the other parameters are stored within the node.

To calculate **if** a reaction to a certain event should be done we will consider the event type, closeness value and activity level. Basically, we will build the mean of closeness value (c) and activity level acl and compare it to the threshold t_e for the event type. If the calculated value is higher than the stated threshold for the event type t_e , a reaction should occur.

$$f(x) = \frac{c \cdot acl}{2} \geq t_e$$

This way the reaction of one node to a specific event of another node would always be the same. To prevent that from happening we will randomise the value using a normal distribution.

$$X \sim N(\mu, \sigma^2)$$

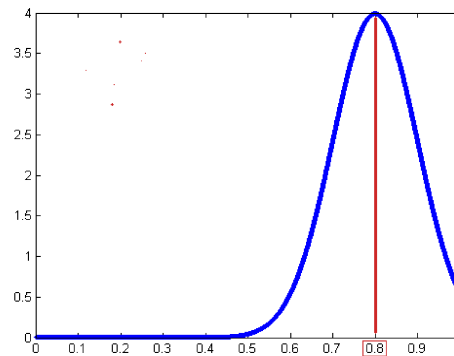


Figure 14: Gaussian distribution
($m=0.8, s=0.1$)

Choosing the mean of the calculated value $f(x)$ and a standard deviation of 0.1 produces a variance where 68% of the values are within mean ± 0.1 , 95% within mean ± 0.2 and 99% within mean ± 0.3 . Because a normal distribution does not have an absolute minimum or maximum we will cut it at 0 and 1 respectively. Figure 14 shows a Gaussian distribution with the mean 0.8 and a standard deviation of 0.1 fitted to the parameter space 0-1.

As the graphic illustrates, a value below 0.5 is highly unlikely.

$$g(x) = 0 \text{ for } N(f(x), 0.1^2) \leq 0$$

$$g(x) = N(f(x), 0.1^2) \text{ for } 0 \leq N(f(x), 0.1^2) \leq 1$$

$$g(x) = 1 \text{ for } N(f(x), 0.1^2) \geq 1$$

The normal distributed number $g(x)$ will now be compared to the threshold t_e instead of $f(x)$.

To determine **when** the reaction will take place in the time-line we will again use activity level acl and closeness value c together with availability level avl , availability type avt and a predefined maximum time interval i for reactions. Because it should also be random, each time a reaction gets triggered and to save calculations, we will use the same normal distributed value $g(x)$ that we have already computed using c and acl . To get a time interval we will multiply avl with the interval i and $1-g(x)$.

$$h(x) = avl \cdot i \cdot (1 - g(x))$$

For normal tweets we will compute the time of action in a similar way but without closeness value:

$$f(x)_{tweet} = \frac{acl + avl + act}{3}$$

Again, we add a variance by using a random Gaussian distributed value:

$$g(x) = 0 \quad \text{wenn} \quad N(f(x), 0.1^2) \leq 0$$

$$g(x) = N(f(x), 0.1^2) \quad \text{wenn} \quad 0 \leq N(f(x), 0.1^2) \leq 1$$

$$g(x) = 1 \quad \text{wenn} \quad N(f(x), 0.1^2) \geq 1$$

In the end, we multiply with the defined maximum interval i :

$$h(x) = i \cdot (1 - g(x))$$

Each time a node sends a message, a decision has to be made in terms of **what** type this message is. In our model there are two main scenarios:

1. "Status update": Node sends a message by its own intention
2. "Reaction": Node reacts on another event (message type)

Possible event types for self-intended "status updates" are a normal tweet, a tweet containing a mentioning or a directed message. For "reactions" the decision is retweet or directed message (reply).

Research in literature revealed several metrics for these event types. Directed messages (including retweets and replies) have a occurrence of 25% [10], 31% [39] and 36% [19] in different datasets averaging to 31%. From these messages 86% are replies according to [19]. Moreover, 3% respectively 6% of all tweets are considered retweets by [19] and [44]. Sysomos¹⁷ reports on their website[44] that 29% of all tweets get a reaction from which 19,3% are retweets and the rest (80,7%) replies [44]. For convenience purposes, all figures are summarised in Table 4.

¹⁷ <http://www.sysomos.com>

Parameter	Value	Source
Directed messages (of all tweets)	25.4%	[10]
Retweets (of all tweets)	3%	[19]
Directed tweets (of all mentions)	86%	[19]
Tweet including hashtag	5%	[19]
RT not followed by @user	5%	[19]
Mentions (of all tweets)	36%	[19]
Mentions (of all tweets)	12,5%	[36]
Mentions (of all tweets)	31%	[39]
Retweeted tweets	6%	[44]
Replied tweets	23%	[44]

Table 4: Figures for the communication simulation

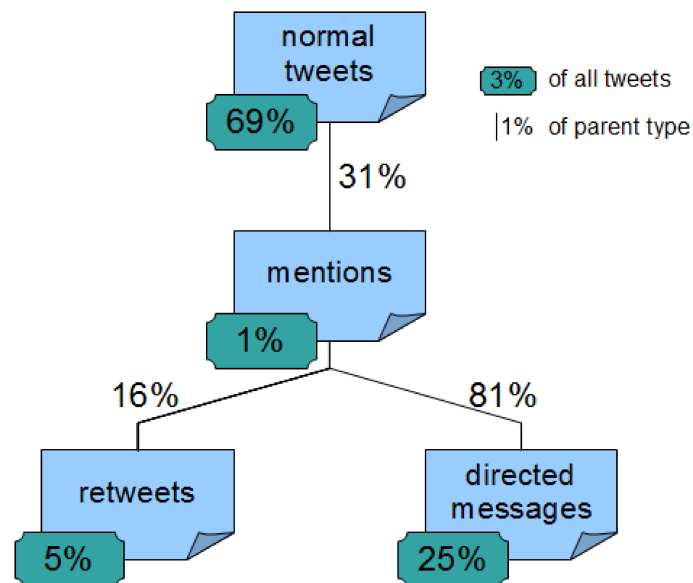


Figure 15: Tweet types ratio breakdown

One can estimate from these numbers that the mass of all tweets gets divided into our event types as follows: normal tweets 69%, mentions 1%, retweets 5%, directed messages 25% (see Figure 15). Considering that 29% of all tweets are a reaction to another tweet (following [44]), we calculate for the former invented scenarios “status update” and “reaction” that the percentages should be 71% and 29% respectively and thus can make a separate breakdown for both settings (Figure 16).

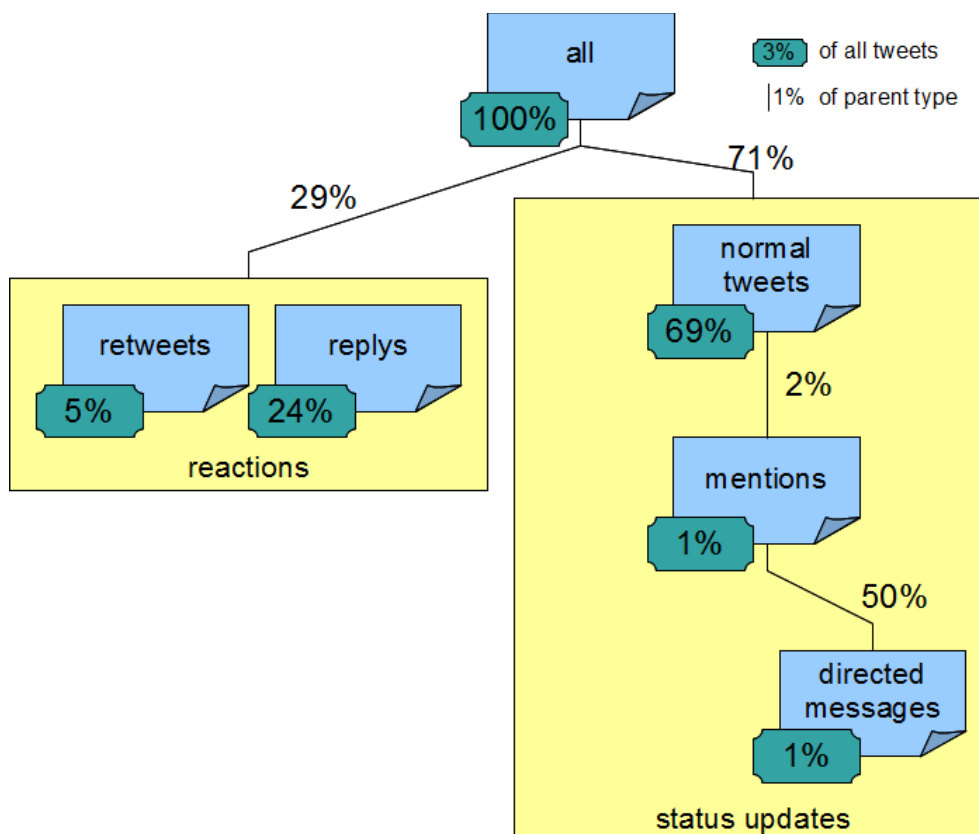


Figure 16: Tweet types ratio scenario separated breakdown

To fit the model to these numbers we have introduced thresholds for each individual message type that control the distribution. Again, there is a randomisation taking place to make the output more realistic while still supporting the desired partition.

3.1.5 Input and Output Interfaces

We have to define interfaces for the applications input and output. The output of the program consists of a graph and the communication in form of messages that need to be stored for further investigations. Nevertheless it should be possible to load a predefined graph into the program serving as basis for the simulation.

Graph Structure

To make it easy to load an existing graph into the simulation we wanted to implement one of the popular file formats already used by other graph related software. The two most popular formats with respect to graph related software are graphML and the Pajek file format. The latter has no official file format definition but is determined by the routines of a Windows based graph analysis tool called Pajek¹⁸. In comparison to graphML, it is much simpler

18 <http://pajek.imfm.si>

because it abandons the use of attributes. This compactness makes it suited for very large graphs where size is a major concern. Nevertheless it is not suited for our purpose because we want to store the nodes attributes within the graph file.

GraphML¹⁹ on the other hand is based on XML²⁰, a hierarchical structured file format that is widely used for data interchange in various domains. It can handle nodes, edges and any type of properties or attributes. Because of the extensibility of XML, it is even possible to save graphical representations within a graphML file. As we will use node specific properties in the simulation process which we want to save as attributes in the graph file, we will therefore have to use graphML. A detailed description of this format being made in chapter 3.2.2 and an additional of a graphML file is attached in appendix C.

Communication Data

Another type of data will be produced by the simulation that produces randomly generated messages to serve as test data for communication analysis applications. As these messages are meant to be the input for other applications analysing social networks, the format in which it will be provided is important for its usability.

Because we expect an amount of thousands of messages, the file format should be light weighted. To make sure it is supported by other applications and, in case it is unsupported, can be implemented without much trouble, it should be a flexible and popular file format. A test environment should always be as close to the real issue as possible to produce reliable information. The file format therefore has to be also akin to what is produced by the real network.

To cut things short, we did not find the perfect format that combines all of these attributes, so we decided to provide more than one format giving the user additional flexibility. It is up to the tester to choose the appropriate format that fits best for their work.

The most simple data format for this use is probably CSV. Comma-separated text files are compact and therefore revealing their strength when it comes to large datasets and the importance of size. They can be viewed in any spreadsheet application for an advanced analysis.

One of the most flexible formats is XML. Because of its well defined hierarchical structure it is used as an exchange format between platforms, especially on the internet. Moreover, XML is supported by a lot of other applications and can easily be implemented in any programming

¹⁹ <http://graphml.graphdrawing.org/>

²⁰ <http://www.w3.org/XML/>

language. One major drawback of this data format is that when it comes to large data it can get difficult to guarantee a valid (closing all branches) XML document if the process gets interrupted or if it does not come to an end.

With regards to authenticity we see that no social network offers a CSV or XML file download for its communication data. Most of them offer some kind of API that allows simple queries to be made. When speaking of Twitter a very akin application is Status.Net²¹, an open source project developing a software for setting up a personal social network service. Status.Net features a Twitter-like API compatible to all major query scenarios. By using Status.Net as a container, it is possible to use the same methods to retrieve communication data as one would use with the original Twitter social service.

Therefore, the Twitter-Testbed will feature a CSV and XML export of all the communication information like message text, sender, message type, mentioned users and referred message. Additionally, we will implement an interface to connect and propagate all messages to a Status.Net instance. The user may either use the API or the web front-end to retrieve the desired information from the Status.Net application.

Statistics and Debug Mode

For convenience purposes we plan to implement some statistical metrics for characterizing the graph after generation has finished. It should be possible to calculate some overall statistics for the network and to extend the class with more statistical properties. Further, debugging information can be printed to follow the processes routines and reconstruct decisions that have been made by the application.

21 <http://status.net/>

3.2 Implementation

In this section the implementation of the network generator and the communication simulation is presented. To give an overview, a diagram containing the whole process is shown first (Figure 17). Next, general implementation issues are being discussed. Later on, this section will take a look at the details of those components.

During the search for a comprehensive framework to support the convenient manipulation of graphs in Java²², we found JUNG, a software library with extensive built-in capabilities. The main features will be described in 3.2.1. The whole implementation was done in Java using the JUNG-framework for graph manipulation.

GraphML, which is already supported by JUNG, will be used to store the network structure in a file. It is an open XML-based file format described in more detail in section 3.2.2.

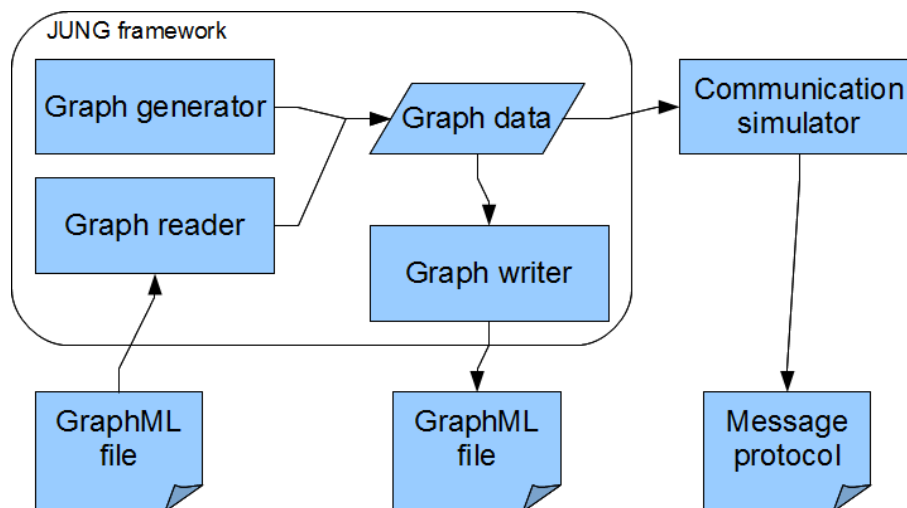


Figure 17: Application overview

The process of graph generation is completely separated from the simulation process. Therefore, it is possible to load any graphML-conform directed graph into the program. The initialisation will generate the required properties for the simulation to the nodes if not available in the graphML file provided.

3.2.1 The JUNG Framework

The Java Universal Network/Graph Framework²³ (short JUNG framework) is an extendible framework for modelling analysis and visualisation of all sorts of graphs in Java. It is available for free as an open source project under the Berkeley Software Distribution (BSD) license²⁴.

²² <http://creativecommons.org/licenses/by/3.0/>

²³ <http://jung.sourceforge.net>

²⁴ <http://www.lininfo.org/bsdlicense.html>

JUNG supports directed graphs, undirected graphs, multi-modal graphs, hypergraphs and nearly any other representation of entities and their relations, thereby making it possible to extend the program using any other social network.

Providing a rich visualisation framework, JUNG makes it easy to implement interactive exploration of graphs. Because it is a very flexible graph framework and is written in Java, which we intend to use for our program as well, we have chosen to use the JUNG framework.

To sum up, JUNG was chosen because it is written in Java, freely available, has a detailed documentation and a strong user base, it supports all kind of graphs and is extensible.

We are using JUNG2 (version 2.0.1) which is a major revision of the former JUNG framework. It depends on three other libraries: Junit²⁵, Colt²⁶ and Common Collections²⁷ included in the downloadable JUNG package.

JUNG2 provides an interface for graphs and sub interfaces for special kinds of graphs, defining the basic operations that can be performed on a graph. Nodes and edges in the graph can be any Java object starting from simple integer values to complex objects.

3.2.2 The GraphML File Format

GraphML is an XML²⁸ based file format for graphs released under the Creative Commons License. It is supported by the JUNG framework and is widely used. As it is based on the well known XML, it is intuitive to work with and compatible with a variety of other programs supporting XML.

Important XML elements to recognize for GraphML:

- **graphml**: the root element of the GraphML document
 - **key**: a type description for graph element properties
 - **graph**: the beginning of the graph representation
 - **node**: the beginning of a vertex representation
 - **edge**: the beginning of an edge representation
 - **data**: the key/value data associated with a graph element

A graphML file has three parts starting with a common XML schema definition Figure 18, including file format (“xml”) and data format (“graphml”) definitions:

25 <http://junit.sourceforge.net/>

26 <http://acs.lbl.gov/software/colt/>

27 <http://sourceforge.net/projects/collections/>

28 <http://www.w3.org/XML/>

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/graphml">
```

Figure 18: graphML XML schema definition

Next, we will look into attributes specifications (Figure 19). A graphML attribute is defined by a “key” element specifying an identifier (`id`) and a domain (`for`). Additionally, a name (`attr.name`) and datatype (`attr.type`) can be specified. It is possible to define a description (`desc`) and a default value (`default`) for a graphML attribute.

```
<key id="al" for="node" attr.name="activityLevel" attr.type="double">
  <desc>Activity Level is ...</desc>
  <default>0</default>
</key>
<key id="t" for="node" attr.name="TweetResponseTime" attr.type="double">
  <desc>TweetResponseTime is ...</desc>
  <default>0</default>
</key>
<key id="at" for="node" attr.name="ActivityType" attr.type="int">
  <desc>Activity Types describe ...</desc>
  <default>0</default>
</key>
<key id="rt" for="node" attr.name="ReTweetResponseTime" attr.type="double">
  <desc>ReTweetResponseTime is ...</desc>
  <default>0</default>
</key>
<key id="w" for="edge" attr.name="weight" attr.type="double">
  <desc>Weight is ...</desc>
  <default>0</default>
</key>
```

Figure 19: graphML attributes specification

The graph structure (Figure 20) is defined in a graph element which can declare an default edge type (`edgedefault`) and a description of the graph (`desc`). The graph element holds a list of nodes and their data values followed by a list of edges and data values.

```

<graph edgedefault="directed">
  <desc>This is a TwitterGraph generated by a ForestFireModel implemented by
  Andreas Scharf at the Distributed System Groups of the Technical University
  Vienna. (Vertices: 1000 Edges: 234996)</desc>
  <node id="5219">
    <data key="al">0.331340844351512</data>
    <data key="t">0.8445161016295285</data>
    <data key="at">2</data>
    <data key="rt">0.6252733485264058</data>
  </node>
  <node id="5949">
    <data key="al">0.8269918223619688</data>
    <data key="t">0.2236424681753011</data>
    <data key="at">3</data>
    <data key="rt">0.29174933234809375</data>
  </node>
  <edge id="1055121" source="5679" target="5286">
    <data key="w">0.7978553809181289</data>
  </edge>
  <edge id="945447" source="5079" target="5012">
    <data key="w">0.7916924680049227</data>
  </edge>
</graph>

```

Figure 20: graphML graph structure

The opened graph tag has to be closed at the end of the file in order to get a valid GraphML file (Figure 21).

```
</graphml>
```

Figure 21: graphML end of graphml definitions

3.2.3 Status.Net

Status.Net²⁹ is a social network service written in PHP³⁰. It allows anyone to create its own private social network aiming to bring the benefits of micro blogging communication into companies. The program is available for free as an open source project and can be downloaded and used under the Create Commons License³¹. It features additional functionality like cross-posting from and to other social network services, availability of address books and much more. Still, the basic usage is very similar to that of Twitter. Like Twitter, the main operations are making a status update, replying to others statuses and forward someone's status. The underlying network is directed and relationships are named "followers" and "following" just like Twitter does. Also, the user interface features known concepts from Twitter and other social networks, like the timeline and an extra "replies"-section. Figure 22 and 23 show the home screen of both applications that make the similarity visible.

29 <http://status.net/>

30 "PHP: Hypertext Preprocessor" is a scripting language especially suited for web development (<http://www.php.net/>)

31 <http://creativecommons.org/licenses/by/3.0/>

The main reason why we use Status.Net as one of the output methods is that it features a Twitter-compatible API which enables people to use the same programs for Twitter and Status.Net by just changing the server address. Thereby, it is possible for developers to test any application designed to read the Twitter API without making supplementary changes. In addition, it enables the user to interact with the test environment, for instance, to log into a specific user's account and view all related messages to explore the communication flow.



Figure 22: Status.Net home screen

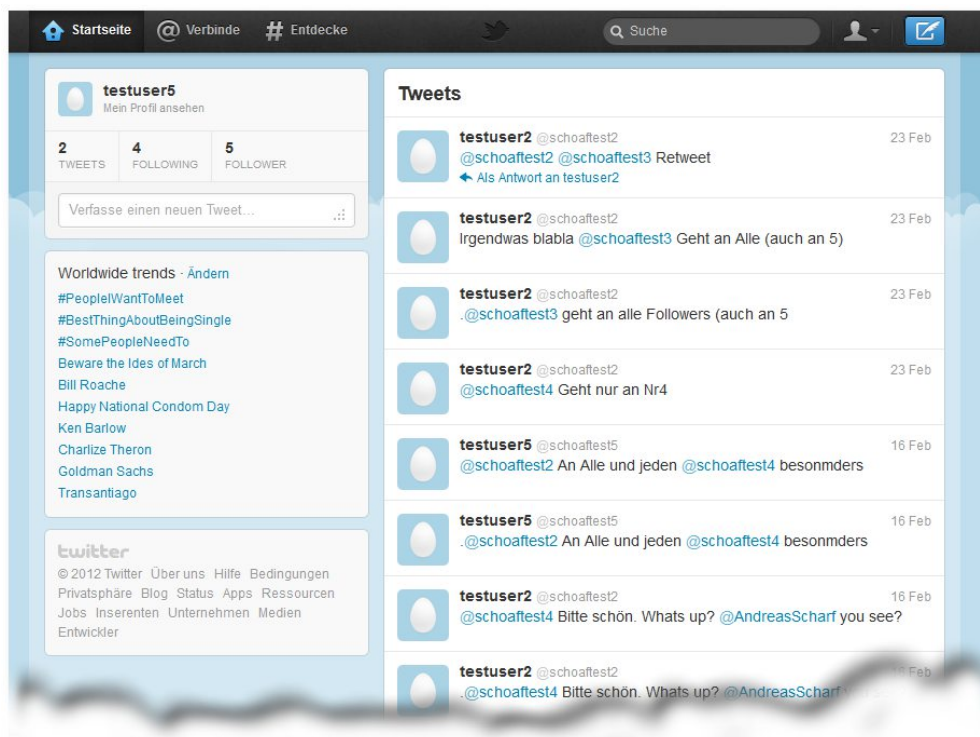


Figure 23: Twitter home screen

3.2.4 Graph Generation

For the implementation of the graph generation described in the design part (3.1.3), we have used JUNG-Framework that supplies all necessary methods to create and manipulate a network structure. As JUNG is already described in chapter 3.2.1, we will now focus on the implementation of our theoretical methods. The framework is already providing all the relevant graph types suited for a variety of graph structures. For our purpose of imitating Twitter we used the class `DirectedSparseGraph` that gives us a graph only allowing directed edges without self loops and permits parallel edges. A graph in JUNG is able to hold any objects as nodes and edges so we created the new classes `TwitterNode` and `TwitterEdge` which get the desired properties and functions needed in the simulation process. We have to pass it as a graph factory to the model generator as can be seen in Listing 2.

```
Factory<DirectedSparseGraph<TwitterNode, TwitterEdge>> graphFactory =
    new Factory<DirectedSparseGraph<TwitterNode, TwitterEdge>>() {
    @Override
    public DirectedSparseGraph<TwitterNode, TwitterEdge> create() {
        return new DirectedSparseGraph<TwitterNode, TwitterEdge>();
    }
};
```

Listing 2: Defining a graph factory

Class graphs.Node

In order to store the properties we have defined for our simulation in 3.1.4. We created our own node objects and added the appropriate variables. This is the basic node class that is intended to ease the implementation of additional node types, for example when implementing another social network type. It provides an `id` (`int`) and a `nextEvent` (`long`) variable. The former is a provides a unique identifier, the latter will be used to store an event time used by the scheduler to cumulate events to a particular point in time.

Further the class defines variables for the properties that have been described before:

```

        int    availabilityType;
        double availabilityLevel;
        double activityLevel;
        LinkedList<Integer> retweetedMessagesQueue;

```

Together with the getter and setter methods this is the blueprint for every node of the application.

Class graphs.TwitterNode

The class `TwitterNode` will instantiate the node objects for the Twitter graph. It extends the `graphs.Node` class and adds methods used specially for Twitter nodes. To provide each node with an ability to remember which message has already been retweeted, it appends a list of retweeted messages (`LinkedList<Integer> retweetedMessages`). To limit this list it will make a simple size check before adding new message IDs:

```

public void addToRetweetedMessageQueue(int msgId) {
    if (retweetedMessages.size() >= RETWEETED_MESSAGES_QUEUE_SIZE) {
        retweetedMessages.removeFirst();
    }
    retweetedMessages.add(msgId);
}

```

Listing 3: Function `addToRetweetedMessageQueue` of `TwitterNode`

The memory of nodes can be adjusted by changing the size of `RETWEETED_MESSAGES_QUEUE_SIZE` allowing a node to remember more or less retweeted tweets.

Class graphs.Edge

The same way we implemented a base class for nodes, there is also a root class for edges to define basic functionality. These objects will be used as edges in the graph. As we do not need a lot of parameters attached to the edge in our simulation, this is a very simple class only providing a distinct `id` (`int`) and a `closenessValue` (`double`) variable that has to be between 0 and 1.

Class graphs. TwitterEdge

The class used to emulate the Twitter network (`TwitterEdge`) does not need any further methods or variables inheriting all properties from `graphs.Edge`. It makes sure that the right edge type is used when more instances of `Edge` are implemented.

Class generators. ForestFireModelGenerator

As the Forest Fire Model is not yet available in Java its code has been reimplemented for this thesis extending the class `EvolvingGraphGenerator` provided by the JUNG framework. To evolve the graph the method `evolveGraph()` (see Listing 4) has to be called which creates a new node, chooses one or more ambassadors and starts the burning process to link this node to others. In the end, all discovered nodes are linked to the new vertex and are given the possibility to link back to the new vertex. To change the number of ambassadors one can set the appropriate property in the settings file. Providing a maximum of ambassadors, a random number up to this maximum will then be chosen as the number of starting nodes for the burning process.

```

public void evolveGraph() {
    int randVertex;
    Vector<Node> toBeLinked = new Vector<Node>();
    // copy the preexisting Vertices
    Node[] preexistingNodesArray = new Node[mGraph.getVertexCount()];
    mGraph.getVertices().toArray(preexistingNodesArray);

    // STEP 1: Create a new Vertex
    Node newVertex = vertexFactory.create();
    // STEP 2: Get a random Vertex as Ambassador
    // adapt the number of ambassadors to preexisting nodes
    int ambassadors = 1;
    if(this.maxAmbassadors>1) {
        if (this.maxAmbassadors >= preexistingNodesArray.length/4) {
            ambassadors = (int)Math.ceil((double)preexistingNodesArray.length/4);
        }
        ambassadors = this.mRandom.nextInt(ambassadors)+1;
    }
    // STEP 3 burn through the ambassadors
    for(int i=0; i<ambassadors; i++) {
        randVertex = mRandom.nextInt(preexistingNodesArray.length);
        // STEP 4: remember this vertex as to be linked later
        toBeLinked.add(preexistingNodesArray[randVertex]);
        // STEPS 5-7 are applied by the function burn()
        toBeLinked = burn(preexistingNodesArray[randVertex], toBeLinked, 1);
    }
    // STEP 8: Add the new vertex and link all the discovered nodes to it
    mGraph.addVertex(newVertex);
    for (Node node : toBeLinked) {
        if (!mGraph.isSuccessor(newVertex, node)) {
            mGraph.addEdge(edgeFactory.create(), newVertex, node);
            // STEP 9: Backlink extension: allow nodes to link back
            if( mRandom.nextDouble() < blp ) {
                if (!mGraph.isPredecessor(newVertex, node)) {
                    mGraph.addEdge(edgeFactory.create(), node, newVertex);
                }
            }
        }
    }
} } }

```

Listing 4: Function `evolveGraph` of class `ForestFireModelGenerator`

`evolveGraph()` will call the method `burn()` that calls itself recursively to link to neighbours of newly discovered nodes. Because already discovered nodes should not be followed anyway to avoid the algorithm from cycling, the extension of back-links is performed after the whole burning process has been finished (see Listing 4).

The method `burn()` takes three input parameters:

Node	node	The currently active node to “burn” from.
Vector	toBeLinked	A simple vector of nodes that holds already “burned” nodes.
int	depth	A regulator to prevent the burning process to dig too deep

Additionally, it uses parameters that are globally specified for the `ForestFireModelGenerator`:

double	pf	Forward burning probability
double	pb	Backwards burning probability

For each edge type (incoming and outgoing) it performs the same procedure only varying the corresponding parameter p (p_f for outgoing and p_b for incoming edges). The corresponding parameter p is generating a geometric distributed random number x with mean $\frac{p}{1-p}$ and choosing randomly x out of all edges of that direction, or all if $x > (\text{number of edges})$. If the chosen nodes are not already part of the `toBeLinked` vector, they get added and the `burn()` method gets evoked for each of them to recursively visit neighbours some hops away. Listing 5 shows the source code of the implementation of function `burn()`.

```

private Vector<Node> burn(Node startVertex, Vector<Node> toBeLinked, int depth) {
    // if the depth limit is reached we stop here
    if (depth >= this.limitDepth) { return toBeLinked; }

    int x; // Stores the random expression of the pf and pb
    double[] p = { pf, pb }; // Forward- and Backwardburning probabilities
    ArrayList<Node>[] neighbours = new ArrayList[2];
    neighbours[0] = new ArrayList<Node>();
    neighbours[1] = new ArrayList<Node>();
    Vector<Integer> chosenNodes = new Vector<Integer>();
    neighbours[0].addAll(mGraph.getSuccessors(startVertex)); // followees
    neighbours[1].addAll(mGraph.getPredecessors(startVertex)); // followers

    for (int i = 0; i < 2; i++) {
        chosenNodes.clear();
        // STEP 5: Generate a geometrically distributed random number
        x = Distributions.nextGeometric(p[i], randGenerator);
        // STEP 6: Randomly choose x nodes linked to V (or all if x>all)
        if (!neighbours[i].isEmpty() && x > 0) {
            if (x < neighbours[i].size()) {
                int numChosen = 0; int actRand;
                while (numChosen < x) {
                    actRand = mRandom.nextInt(neighbours[i].size());
                    while (chosenNodes.contains(actRand)) {
                        actRand = mRandom.nextInt(neighbours[i].size());
                    }
                    chosenNodes.add(actRand);
                    numChosen++;
                }
            } else {
                for (int j = 0; j < neighbours[i].size(); j++) {
                    chosenNodes.add(j);
                }
            }
            // STEP 6b: Add the choosen Nodes from above
            Iterator<Integer> it = chosenNodes.iterator();
            while (it.hasNext()) {
                Node n = neighbours[i].get(it.next());
                if (!toBeLinked.contains(n)) {
                    toBeLinked.add(n);
                    // STEP 7: recursively apply this code to the new vertices
                    Vector<Node> newVecs = burn(n, toBeLinked, depth+1);
                    for (int k = 0; k < newVecs.size(); k++) {
                        if (!toBeLinked.contains(newVecs.get(k))) {
                            toBeLinked.add(newVecs.get(k));
                        }
                    }
                }
            }
        }
    } // close all brackets
    return toBeLinked;
}

```

Listing 5: Function burn() of class ForestFireModelGenerator

Class generators.Initializer

To be able to load other graphs into the system we separated the initialisation of nodes and edges properties from the generation process. This class provides a convenient way to set all needed values by calculation of random numbers in a defined range. Therefore, it loops through all edges and nodes and sets the appropriate value.

The following settings can be made to the initializer changing its behaviour (Table 5):

Property Name	Default	Description
<i>actSourceRatio</i>	0.1	Ratio out to in degree that specifies an information source
<i>actSeekerRatio</i>	0.1	Ratio in to out degree that specifies an information seeker
<i>avtWeekdayRatio</i>	0.6	ratio of weekday out of all availability types
<i>avtWeekendRatio</i>	0.2	ratio of weekend out of all availability types
<i>avtAlldayRatio</i>	0.2	ratio of allday out of all availability types

Table 5: Properties of the initialiser class

The properties *actSourceRatio* and *actSeekerRatio* assign the ratio of out- to in- and in- to out-degree ratio respectively. If a nodes degree ratio is smaller then the provided value it will be set to the according activity type.

Availability type is assigned randomly to each node using the percentage values set in *avtWeekdayRatio*, *avtWeekendRatio* and *avtAlldayRatio*. The total of all three ratios should be equal to 1, meaning 100%.

3.2.5 Communication Simulation

As described in section 3.1.4 (Event Types), there are five main types of messages that evoke different behaviour on the receiver's side. These are tweets, mentions, directed tweets, replies and retweets. To distinguish them properly and to guarantee a convenient implementation of additional event types in the future, there is one class for each message type that provides the required properties. A specific trigger function will specify the appropriate action when this event takes place. Figure 24 shows the hierarchy of the classes of the events. Every event has to extend the class `Event` to implement and override the abstract method `triggerEvent()` to successfully integrate into the simulation system.

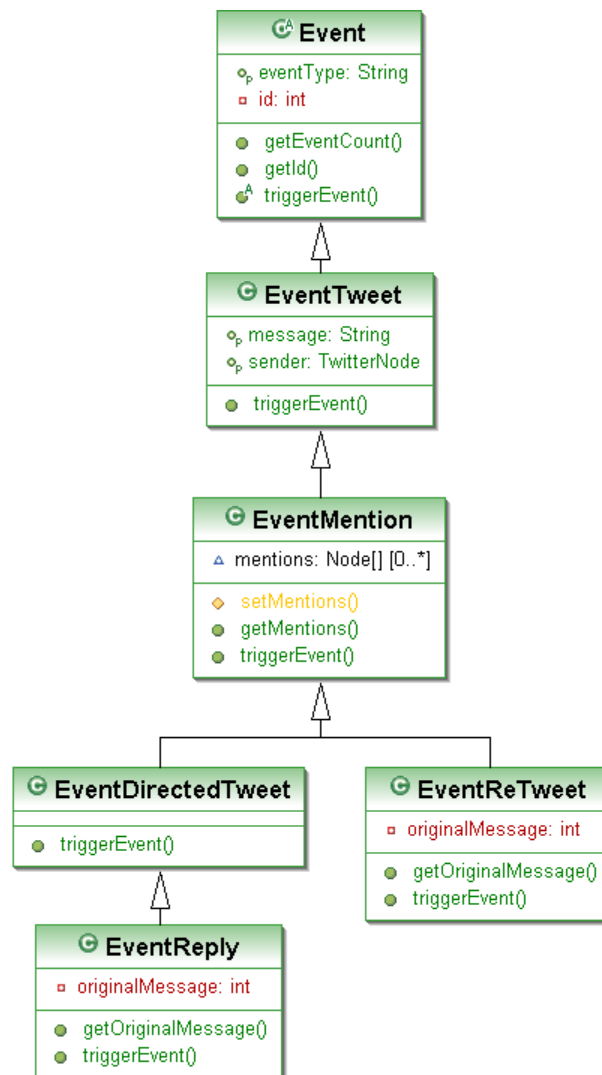


Figure 24: Implemented events class schema

Mentions, directed tweets, retweets and replies are actually messages like normal tweets but they have a different audience and are causing distinct reactions from the receiver. In Twitter these messages get distinguished by identifying key terms. This could possibly also be done whilst simulation but because we already know the intention of a message since its generation, it does not make sense to throw that information away just to parse it in again a few minutes later to get the semantic back. To save that effort, each of these message types get their own class so that they can be easily distinguished and are able to hold specific variables and functions. Because the types have similarities, we will make use of inheritance and declare `EventMention` as subclass of `EventTweet`. `EventDirectedTweet` will be child class of `EventMention` and `EventReply` is subclass of `EventDirectedTweet`.

In the following paragraphs we will describe the event classes and their methods. It turns out that the definition of proper event classes is one of the key factors for the simulation process.

Class `events.Event`

The base class for every event is `events.Event`. An event is every action that can be stored and processed by the scheduler. Therefore, it has to define all variables and methods that are required for the scheduler. Its function is to serve as a blueprint for any possible event.

Basically, an event has to feature at least an unique ID and a trigger function that can be used by the scheduler to perform the appropriate

action when it is next in the queue. To be able to group events according to their intended behaviour, we added an simple event-type string variable. Besides, the following events we have implemented in this version of the application, it is possible to hook any other action in the scheduler to be performed at a certain point in simulation time by extending `events.Event`. We will give some ideas for additional events in the chapter Future Work (7.2).

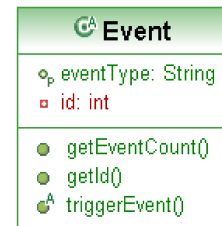


Figure 25: Class `Event`

Class `events.EventTweet`

This is the class for normal tweets. It extends the class `events.Event` that defines the main methods for all events. Because every other message type is actually a modification of the basic tweet, it provides the main properties for all other message types in this program.

In addition to the unique ID inherited from `events.Event`, it needs to

hold the sender of the message and the text itself. It implements the abstract method `triggerEvent()`, which has to go through every node with an connected incoming edge (followers of this node) to activate their reactions. Because this class provides the most basic implementation of the `triggerEvent()` function we will attach its source code (Listing 6) to explain the concept of “message sending” that will also be used by the other types with some additional changes.



Figure 26: Class `EventTweet`

The functionality can be explained as follows: To get all followers of the sender the function calls JUNGs `getPredecessors()` function and then iterates over all of them to calculate for each node the likeliness to respond, depending on sender and receiver. This calculation of the probability is done by `calcResponseWillingness()`. The calculated value is compared to a threshold that can be adjusted in the properties file. If the decision is made in favour of a reaction then a kind of response has to be chosen, which is done by the method `getReactionEvent()`. The result gets added to the scheduler. Both methods `calcResponseWillingness()` and `getReactionEvent()` are described in chapter Network Behaviour (3.1.4). The source code can be found in the appendix (D.).

Whether a reaction is taking place or not, the sender has to schedule it's next tweet-event to stay active.

```
@Override
public void triggerEvent(Scheduler scheduler) {
    // get all followers (in-edges)
    TwitterScheduler sched = (TwitterScheduler)scheduler;
    Collection<TwitterNode> c = sched.getGraph().getPredecessors(sender);
    Iterator<TwitterNode> it = c.iterator();

    // 1) the message is received by all followers, possibly evoking a response
    while (it.hasNext()) {
        TwitterNode reciever = it.next();
        // calculate a probability value for this node to decide a reaction
        double willingness = calcResponseWillingness(sched, sender, reciever);
        if (willingness >= sched.getWillingnessThresholdTweet()) {
            // add a proper reaction (retweet or reply) to the scheduler
            sched.addEvent( this.getReactionEvent( sched, reciever, this.getSender(),
                                                    willingness )
                            );
        }
    }
    // 2) the node itself has to add its next tweet event in the future
    sched.addEvent(sched.getNextRandomTweet(this.sender));
}
}
```

Listing 6: Source code of the `triggerEvent()` function

Class events.EventMention

This class provides the extra capability of holding a list of nodes that are mentioned in the tweet. Moreover, a modified version of the `triggerEvent()`-function is implemented that has to differentiate in its action between those nodes that are explicitly mentioned and the rest of the receivers. All other properties and methods derive from the super class `events.EventTweet`.



Figure 27: Class `EventMention`

The difference in the `triggerEvent()` function is an additional loop that goes through all mentioned nodes to calculate a response willingness with another probability. Because they are mentioned it is more likely for these nodes to produce a reaction.

Class events.EventDirectedTweet

An event of type `EventDirectedTweet` is a subclass of `EventMention` and does not provide any additional properties.

Only the method `triggerEvent()` is different in the sense of that this message type is only visible to the users mentioned and their followers. To achieve this behaviour all to be done is choosing the appropriate nodes to iterate through.



Figure 28: Class `EventDirectedTweet`

Besides the small changes in the `triggerEvent()`-function, its main role is to make a distinction between this event and the other events so that the intention of the event is clearly stated.

Class events.EventRetweet

`EventRetweet` is also a subclass of class `EventMention` and provides an extra property that stores the first message that has been originally retweeted (`int originalMessage`). This way it is ensured that it is always clear if a tweet has already been retweeted even if the message contained changes. In Twitter a change to the message text



Figure 29: Class `EventRetweet`

of a retweet is often caused by edits of users who want to add a comment or have to shorten the text to conform to the 140 character limit when they try to add the name of the source. But because the meaning of a retweet does not change with these edits (at least from the senders opinion), it is important to make sure an already retweeted message can be identified. To make this possible this original message ID is added to a limited stack of the node. It acts as the node's memory and thus can identify all other tweets deriving from the same original message as long as it stays in the stack.

The `triggerEvent()` function is fairly the same as in `EventTweet`. The only difference between the two is another threshold because retweets are more likely to be retweeted than normal tweets.

Class events.EventReply

On the one hand a reply is like a directed tweet in the sense of that it does include a mentioning that it addresses too. On the other hand it is also similar to a retweet because it is a reaction to another message that has to be referenced. We decided that a reply belongs to the group of directed tweets because it covers the intention of the message a bit more, which basically is to direct an update to a specific person rather than relaying some information to another audience.

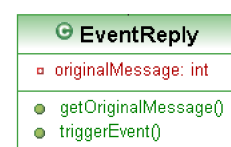


Figure 30: Class `EventReply`

The `EventReply` adds a variable `originalMessage` to its parent class `EventDirectedTweet` through which it identifies the update it answers. A reply's trigger function is exactly the same as the one of `EventDirectedTweet`. Besides, it is using another threshold value to separately control the amount of replies.

Class simulation.Scheduler

The scheduler is the heart of the simulation process. It has to hold all events in a timely order and provide the ability to add new events at any time. At simulation time the scheduler has to call the events in the appropriate order and run their trigger function. Further, it provides an interface to the output methods.

This class defines the following variables:

long	startTime	Start time of the simulation run
long	endTime	End time of the simulation run
long	maxInterval	Maximum timespan between updates.
double	cumulationThreshold	Percentage of cumulated events

Table 6: Parameters defined in class Scheduler

All variables of Table 6 can be set using the properties file. While `startTime` shifts the simulation to a desired starting point, `endTime` acts as a stop criteria to limit simulation. It is set using one of the duration settings described in chapter 4. The value of `maxInterval` influences the frequency of status updates because it limits the time between the current and the next update of a node. `CumulationThreshold` controls the ratio of actual cumulation out of corresponding occurrences.

The main features of this class are first of all to add new events to the right place of the queue, and second of all, to trigger its behaviour. Both are rather simple, the latter (`start()`) is attached in Listing 7. It has to iterate over all events of the event list and trigger its action. Additionally, events are sent to output handlers if a appropriate connector is provided.

```
public void start() {
    // Iterate through the event list until stop criteria met
    while (!eventList.isEmpty() && !(modelCurrentTime >= modelEndTime)) {
        EventNote e = eventList.firstElement();
        this.setModelCurrentTime(e.getTime());
        eventList.remove(0);
        e.triggerEvent(this);

        // Propagate event to the chosen output
        if(useConnector) {
            for(ConversationConnector c:cc) {
                c.writeEvent(e);
            }
        }
    }
}
```

Listing 7: Function `start()` of class Scheduler

Class simulation.TwitterScheduler

This is a specialisation of the scheduler for use for a Twitter network, extending the class `simulation.Scheduler`. It defines the thresholds for message types to control its distribution and provides a function to calculate the next status update time and type according to the description in chapter 3.1.4.

3.2.6 Input and Output***Class io.TwitterGraphMLReader***

We implemented a graph reader in order to be able to run several simulations with the same underlying graph structure and to enable sharing network data. A graphML reader and writer functionality is already implemented in the JUNG framework and must only be fed with the appropriate transformers to translate the properties of the new objects `TwitterNode` and `TwitterEdge` in graphml syntax. The classes `io.TwitterGraphMLReader` and `io.TwitterGraphMLWriter` provide a convenient way to read and write a Twitter graph with all its properties by providing the necessary transformers. Listing 8 shows a sample of a node transformer for the graph reader. The edge transformer follows the same pattern.

```
Transformer<NodeMetadata, TwitterNode> vertexTransformer = new
    Transformer<NodeMetadata, TwitterNode>() {
    @Override
    public TwitterNode transform(NodeMetadata metadata) {
        TwitterNode v = new TwitterNode(
            Integer.parseInt(metadata.getId()),
            Integer.parseInt(metadata.getProperty("avt")),
            Double.parseDouble(metadata.getProperty("avl")),
            Integer.parseInt(metadata.getProperty("act")),
            Double.parseDouble(metadata.getProperty("acl"))
        );
        return v;
    }
};
```

Listing 8: graphReader - node transformer

Class io.TwitterGraphMLWriter

In order to save our graph we use the graphML writer that is already implemented in the JUNG framework and extend it with the required node and edge transformers using the `addVertexData()` and `addEdgeData()` methods. A sample for setting up the availability type property is provided in Listing 9. For each property a transformer has to be added to the writer using `addVertexData()` to inform it of one's existence and define a transformation routine.

```

graphWriter.addVertexData (
    "avt",
    "Description: ...",
    "0",
    new Transformer<TwitterNode, String>() {
        public String transform(TwitterNode v) {
            return Integer.toString(v.getAvailabilityType());
        }
    });
addVertexData(key_identifier, description, default_value, transformer);

```

Listing 9: graphWriter - node transformer

Interface ConversationConnector

As we wanted to support various output methods we implemented a connector class that can be used as an interface between scheduler and output method. Each class that implements this interface can be passed to the scheduler which calls the `writeEvent()` method each time an event gets triggered.

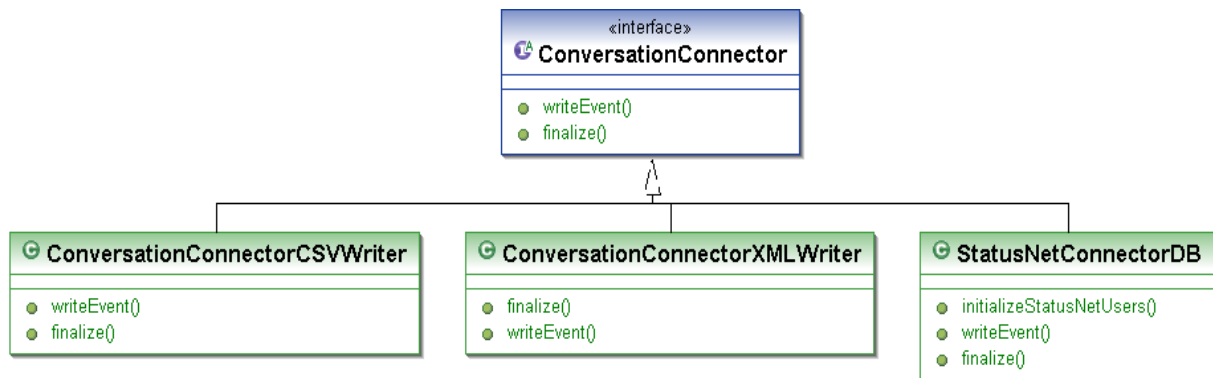


Figure 31: Class diagram for output of conversations

We have currently implemented three ways to save the output of the simulation for further processing, as Figure 31 shows. All messages that get produced can either be stored in a text file or in a Status.Net database. The text file format can be chosen between CSV and XML.

Class ConversationConnectorCSVWriter

This class provides a file writer to save all messages to CSV format. Because it implements the interface `ConversationConnector` it can directly be passed to the scheduler.

Fields are delimited by semicolon (;) and text is enclosed with double quotes (“). To ensure a proper handling all double quotes that appear in messages are replaced by single quotes. The field mentions can contain a list of user IDs separated by comma (,) if the event does not include any mention it is set to string “NULL”. The original message field contains the ID of the referred message or “-1” if none. Figure 32 shows a cut of a sample CSV text file and the parameters included.

```
id;time;eventType;sender;message;mentions;originalMsg
4;5650324;"tweet";2;"1xsp3a7vfvmvq";NULL;-1
8;9858924;"directedtweet";5;"fb@3tf11@12eykwcf";3,12;4
5;12570060;"tweet";11;"4x9upuje9s0o";NULL;-1
...
```

Figure 32: Sample CSV content for the conversation output

Class ConversationConnectorXMLWriter

To give the user more flexibility we implemented a XML-writer to store all communication in a XML-file. For this purpose we used XOM³² an open source API for XML processing in Java.

Every event has an attribute *type* that defines the event type, for example tweet, retweet, etc. and holds at minimum the child elements *id*, *msg* and *sender*, including the appropriate values. Repls, mentions and directed tweets get an additional element *mentions* with one or more child elements *mention*. A *origmsg* element is appended to retweets and repls that holds the message ID of the retweeted/replied update. Listings 10-13 show the resulting XML-syntax for tweets, directed tweets, retweets and repls. A complete sample XML output file is attached in appendix E.

```
<cxml:event type="tweet">
  <cxml:id>1117</cxml:id>
  <cxml:msg>1esgn4bfaug5c</cxml:msg>
  <cxml:sender>691</cxml:sender>
</cxml:event>
```

Listing 10: XML-syntax for tweets

```
<cxml:event type="directedTweet">
  <cxml:id>370</cxml:id>
  <cxml:msg>@user613 02ht7bt0dhy</cxml:msg>
  <cxml:sender>554</cxml:sender>
  <cxml:mentions>
    <cxml:mention>613</cxml:mention>
  </cxml:mentions>
</cxml:event>
```

Listing 11: XML-syntax for directed tweets

```
<cxml:event type="retweet">
  <cxml:id>1141</cxml:id>
  <cxml:msg>RT: 1esgn4bfaug5c</cxml:msg>
  <cxml:sender>14</cxml:sender>
  <cxml:origmsg>1117</cxml:origmsg>
  <cxml:mentions>
    <cxml:mention>691</cxml:mention>
  </cxml:mentions>
</cxml:event>
```

Listing 12: XML-syntax for retweet

```
<cxml:event type="reply">
  <cxml:id>1143</cxml:id>
  <cxml:msg>@user14
90mkym4v170</cxml:msg>
  <cxml:sender>694</cxml:sender>
  <cxml:origmsg>1141</cxml:origmsg>
  <cxml:mentions>
    <cxml:mention>14</cxml:mention>
  </cxml:mentions>
</cxml:event>
```

Listing 13: XML-syntax for repls

32 <http://www.xom.nu/>

Class ConversationConnectorDB

Connecting the scheduler to a database follows the same rules as any other output method. It is done by extending class `ConversationConnector` and implementing a `writeEvent()` method to send event data to a database.

Status.Net provides an API to handle updates in a simple way. Although its usage would probably be beneficial to supporting several Status.Net versions, it is not capable of all manipulations we need for the simulation. Because we are generating totally new networks, we have to ensure that its structure is the same within Status.Net. Therefore, it is necessary to delete and create users and messages.

The API makes use of PHP- and HTTP- URL parameters to receive a request. This may be a feasible method considering posting and receiving updates from time to time, but in our case we are producing hundreds, thousands or even millions of updates in a short time. Passing them through HTTP- requests would take much more time and would in most systems probably result in a denial of service due to overloading of the PHP-service.

Therefore, we decided to write directly to the underlying database which can be considered the fastest way. We had to do some reverse engineering by looking for changes in the database when using the Status.Net web interface. This step was vital to get information about the changes that would have to be made in order to propagate an update correctly to the database. Affected tables are shown in Figure 33 as ER diagram³³.

First, we need to initialise the database tables. The method `initializeStatusNetUsers()`, called at creation time of the connector, truncates all affected tables and creates all users, profiles and the appropriate relations matching the given graph. Each user has to get an entry in the “users”, “profile” and the “inbox” table. For each outgoing edge in our graph an entry to the subscriptions table has to be done to set up the follower / followee structure.

After initialisation, the connector is ready to write updates to the database translating Twitter-Testbed's messages to the database schema. Because the appropriate entries depend on the message type and do not equal their representation, this step cannot be considered trivial. All major entries for a proper integration in Status.Net are listed in Tables 7-10.

For each new message we have to create an entry in table `notice` with the appropriate values. Table 7 lists the important columns for individual updates. If the message does not belong to a conversation (for example normal tweets or directed tweets), a new dataset has to be added to the “conversation” table (Table 8 and the resulting ID has to be written in the `conversation` field of table “notice”. Each new message has to make an update to the “inbox” table (Table 10) adding the ID of the notice to the inbox of all receivers of that update.

³³ The entity-relationship model is a conceptual representation of database tables and their relations.

For replies the field *reply_to* of table “notice” has to hold the user's ID it is a reply to. Further, a new entry in the table “reply” (Table 9) has to be done. These messages are part of an ongoing conversation which has to be referenced in the *conversation* field of table “notice”. Retweets have to put the ID of the retweeting message in field *repeat_of* of table “notice”.

Table	Column	Description
notice	id	Unique identifier taken from the Testbed application
notice	content	Message text
notice	rendered	Message text again, rendered for html output
notice	url	URL of this update (basis url + notice id)
notice	created	Timestamp of the updates creation in simulation time
notice	conversation	Id of conversation it belongs to
notice	reply_to	Id of the referenced user, if any
notice	repeat_of	Id of referenced message, if any
notice	object_type	Activity stream identifier for message type
notice	verb	Activity stream identifier for action type

Table 7: Status.Net database entries required for table notice

Table	Column	Description
conversation	id	Auto incremented identifier
conversation	uri	reference url to the conversation thread
conversation	created	Timestamp of the conversations creation

Table 8: Status.Net database required entries for table conversation

Table	Column	Description
reply	notice_id	Id of the current notice
reply	profile_id	Id of the person this is a reply to

Table 9: Status.Net database entries required for table reply

Table	Column	Description
inbox	user_id	Id of the related user
inbox	notice_ids	List of notices received

Table 10: Status.Net database entries required for table inbox

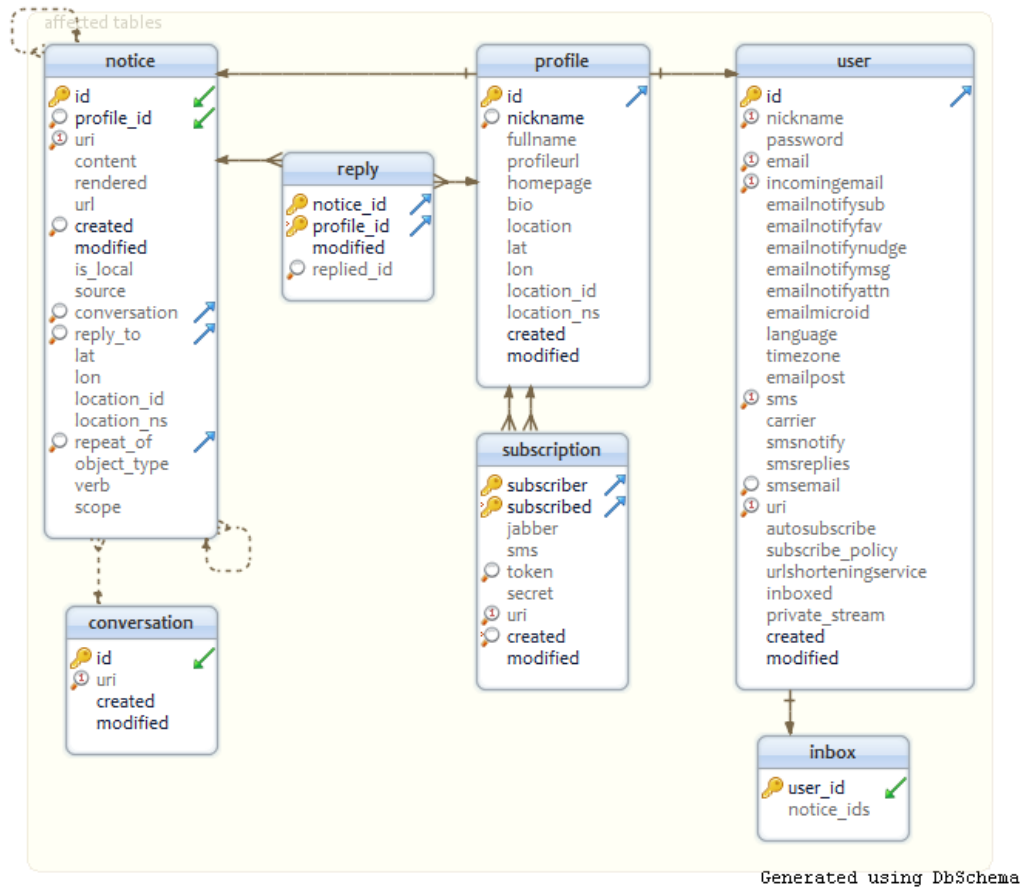


Figure 33: Status.Net affected tables EER diagram

4 Application Usage

The Twitter-Testbed is a simple command-line tool that takes a properties file as input and produces the requested output. The decision to use a properties file as input in favour of other possibilities has been made due to its convenience in usage. It would be confusing to set a large number of parameters through command-line parameters. A graphical user interface is scheduled for the next version due to the large amount of changes concerning available parameters during the development. A properties file can easily be edited and copied. Furthermore, it is possible to create several property files with different settings and save them to disk for later use. It does also advance the ease of sharing graph data by passing this file to interested users.

To run Twitter-Testbed, just call the provided jar-file from within Java and provide a path to a valid properties file. The command-line expression would look like the following (Listing 14):

```
> java -jar twittertestbed.jar -f /path/to/properties.prop
```

Listing 14: Command-line starting command for Twitter-Testbed application

Below we will describe each property that can be set. An example properties file is attached in the appendix (B. Twitter-Testbed Properties File). It is important to only put the name of the parameter and the value in one line separated by a single space character.

Parameters for Graph Generation:

Parameter	Example value	Description
<i>pf</i>	0.8	Forward burning probability
<i>pb</i>	0.7	Backwards burning probability
<i>blp</i>	0.42	Back link probability
<i>initVertices</i>	1	Number of vertices to start with
<i>maxAmbassadors</i>	1	Max number of ambassadors a node can start with
<i>limitDepth</i>	100	Depth limit of burning process for more performance
<i>maxNodes</i>	1000	Number of nodes to be created

Table 11: Properties for process graph generation

All parameters listed in Table 11 are described in detail in chapter 3.1.3 (The Forest Fire Model). Besides the essential parameters *pf*, *pb*, *blp* and *maxNodes* we provide the possibility to start with a number of unconnected nodes (*initVertices*) to produce several connected components. With *maxAmbassadors* it is able to connect far-apart nodes by letting a new node start with more than one random entry point. Raising *initVertices* without the increase of *maxAmbassadors* causes the graph to split into *initVertices* parts because there will be no possibility to link to an unconnected node. One might limit the depth of the burning process to *limitDepth* levels in order to speed up graph generation.

If a graphML file is provided to read the contained network all of the parameters for the graph generation process are obsolete and will be ignored. How to specify input and output parameters is being discussed below. In order to run the graph generation process at least *pf*, *pb*, *blp* and *maxNodes* have to be provided. The rest is optional and will be initialised with the default values.

Parameters for Graph Properties Initialisation:

Parameter	Example value	Description
<i>graphForceInitialise</i>	false	Override graph properties of a loaded graph
<i>actSourceRatio</i>	0.1	Out to in degree ratio specifying a information source
<i>actSeekerRatio</i>	0.1	In to out degree ratio specifying a information seeker
<i>avtWeekdayRatio</i>	0.6	Ratio for type weekday out of all availability types
<i>avtWeekendRatio</i>	0.2	Ratio for type weekend out of all availability types
<i>avtAlldayRatio</i>	0.2	Ratio for type allday out of all availability types

Table 12: Parameters for the graph's initialisation

The initialisation of graph properties is intended to run only if a graph does not already provide this data. Nevertheless, the application can be configured to reinitialise a network by setting *graphForceInitialise* to true. This way it is not necessary to run the graph generation again and again to try out different initialisation settings.

actSourceRatio and *actSeekerRatio* define at what ratio a node is considered to be of activity type information source or information seeker, considering out-to-in-degree and in-to-out-degree ratio respectively.

All properties for the initialisation of graphs are summarised in Table 12.

Parameters for the Simulation:

Parameter	Example value	Description
<i>startTime</i>	0	Model start time as UNIX timestamp
<i>duration</i>	1000000	Maximum simulation duration in milliseconds
<i>durationHours</i>	10	Maximum simulation duration in hours
<i>durationDays</i>	1	Maximum simulation duration in days
<i>randomness</i>	0.1	Standard deviation of random values
<i>maxInterval</i>	400000000	Max. interval between updates in milliseconds
<i>cumulation</i>	0.3	Probability of cumulating events to the same time
<i>updateMentionRatio</i>	0.027	Ratio of updates containing mentions
<i>updateDirectedRatio</i>	0.51	Ration of mentions being directed Tweets
<i>responseTweet</i>	0.33	Probability of a response on tweets
<i>responseDirectedTweet</i>	0.2	Probability of a response on directed tweets
<i>responseMention</i>	0.3	Probability of a response on mentions
<i>responseRetweet</i>	0.3	Probability of a response on retweets
<i>responseReply</i>	0.23	Probability of a response on replies
<i>responseRetweetRatio</i>	0.4	Retweet / reply ratio of responses

Table 13: Parameters for the simulation

To run a simulation a duration has to be provided in either milliseconds (*duration*), hours (*durationHours*) or days (*durationDays*). If more than one of them is found they will be totalled, if none of them are given the simulation will be skipped. Providing a *startTime* (UNIX timestamp) will move the simulation to run from that starting point on. By changing *maxInterval* (milliseconds) it is possible to change the update frequency. The lower *maxInterval* is set the more updates will be sent per hour. If a node wants to place an event into the scheduler while there is already one taking place with this sender, these actions can be cumulated to happen at the time of the first event. Set *cumulation* to specify how many of these events will be cumulated. *updateMentionRatio* is telling the scheduler how many of new updates should contain mentions. *updateDirectedRatio* says how many of these mentions will be directed tweets. The probabilities for reactions can be set for each event type separately (*responseTweet*, *responseDirectedTweet*, *responseMention*, *responseRetweet*, *responseReply*). Finally, *responseRetweetRatio* defines the fraction of retweets out of all reactions, the rest are replies.

Parameters for Input and Output:

Parameter	Example value	Description
<i>graphLoadPath</i>	/path/g-in.graphml	Absolute path to a valid graphML file
<i>graphSavePath</i>	/path/g-out.graphml	Absolute path to save the graph data
<i>convCSVSavePath</i>	/path/conv.csv	Absolute path to save the conversation as csv
<i>convXMLSavePath</i>	/path/conv.xml	Absolute path to save the conversation as xml

Table 14: Parameters for input and output

Table 14 shows all possible input and output methods that can be specified. If no graph should be loaded but a new one generated, one has to remove or out-comment the *graphLoadPath* property by putting a “#” in front of the line. All paths are optional and the appropriate action will simply be skipped if the parameter is not provided or the path value is missing. Definitions of the output methods can be found in section 3.1.5.

Parameters for Status.Net:

Parameter	Example value	Description
<i>host</i>	localhost/statusnet	Host address including database name
<i>username</i>	user1	Database username
<i>password</i>	password	Corresponding password
<i>url</i>	http://localhost/statusnet/	Base url to the status net front-end

Table 15: Parameters for Status.Net

Valid credentials to an existing and properly set-up database has to be provided to be able to output the simulation to a Status.Net installation. How to set-up a Status.Net instance can be found on the website of Status.Net ⁽³⁴⁾. It is advisable to create a database user that has only rights for the Status.Net database tables. If a host is provided in the properties file, the application will try to login with the provided credentials and propagate all messages to the Status.Net database if a connection was successful. The value of *host* has to include host address and database name in one string as the provided example shows (Table 15).

To turn this feature off, simply do not provide these key value pairs, for example by commenting the affected rows out (in fact out-commenting the host line is sufficient).

34 <http://status.net/open-source>

5 Evaluation

The goal of this thesis was to build a Twitter-like network structure and simulate communication on the generated graph. We will evaluate the output of our application by comparing it to metrics of the Twitter network presented in 3.1.2 (Characteristics of the Twitter Network) and 3.1.4 (Network Behaviour). In our evaluation we will also consider speed and memory consumption to provide a perspective of the usability of the application.

At first, we will address the graph generation considering structure properties, flexibility and capability. We will then get to the communication simulation looking at the adaptability and accuracy of the output.

For calculations of graph metrics we made use of built-in capabilities of the JUNG-framework as well as two freely available graph analysis tools, Gephi and nodeXL.

Gephi³⁵ [45] is a free and open source tool intended for rich graph visualisation. It is running on Windows, Linux and Mac OS X operating systems and supports different source formats including GraphML. The program's main focus lies on real time visualisation and interactive exploration of large graphs. Besides a large variety of different visualisations, Gephi comes with a repertoire of metrics that can be calculated and plotted, like degree, diameter, centrality and clustering coefficient.

Using the SNAP³⁶ (Stanford Network Analysis Package) library by Jure Leskovec, developed at Stanford University, NodeXL³⁷ [46] builds an easy-to-use graphical front-end integrated in Microsoft Excel. It is an open source project of the Social Media Research Foundation³⁸ supported by Microsoft Research³⁹ and has a interface to their graph gallery that enables the researchers to collectively share and gather data sets within the community. Importing and exporting graphs in formats like GraphML, Pajek, UCINET and others and the convenient usability of the result spreadsheet made it a useful tool for our evaluation. NodeXL does also support a variety of graph visualisations for graphical exploration.

For the evaluation we used a normal PC with a dual core 2.66 GHz CPU and 4GB memory. The operating system was a 32bit Microsoft Windows 7 Home Premium.

35 <http://gephi.org/>

36 <http://snap.stanford.edu/snap>

37 <http://nodexl.codeplex.com/>

38 <http://www.smrfoundation.org>

39 <http://research.microsoft.com/en-us/collaboration/>

5.1 Graph Generation Metrics

In this chapter we will evaluate the graph generation process and the properties of the generated networks structures. The basis of our decision-making as to whether this procedure is a feasible replacement for crawling the Twitter network to gather test data relies on the metrics. Those metrics include factors like the execution time, memory consumption and the properties of the outputted graphs.

For this purpose we generated seven graphs ranging from 1,000 to 100,000 nodes in size. We fitted the parameters of the generation process to produce networks that exhibit similar structural properties as those we have found in literature for the original Twitter network.

5.1.1 Reality Conformance

We focused on the clustering coefficient and the reciprocity because most of the metrics are biased by the overall size of the network. For both metrics we considered a deviation of less than 0.01 as satisfying. Then we generated graphs varying the models parameters until they had, according to the metrics of Twitter presented in 3.1.2, a clustering coefficient between 0.10 and 0.12 and a reciprocity between 0.57 and 0.59. Table 10 shows the evaluation graphs including clustering coefficient, reciprocity and creation time. A list of all calculated metrics for these networks and the configuration parameters can be found in the appendix (F. Evaluation Graphs).

Nodes	1,000	5,000	10,000	15,000	20,000	50,000	100,000
Edges	1,700	8,543	17,168	25,818	34,312	85,550	171,435
Clustering coefficient	0.1	0.115	0.121	0.116	0.117	0.112	0.116
Reciprocity	0.598	0.583	0.585	0.589	0.582	0.585	0.585
Creation time	< 1s	2s	11s	15s	20s	2m 46s	12m 49s

Table 16: Evaluation of generated graphs (main figures)

By varying the parameters of the Forest Fire Model we fitted the desired graphs to match clustering coefficient and reciprocity in four attempts at the most. The values forward burning probability (pf) and backwards burning probability (pb) that are affecting the clustering coefficient in the way that a rise in either of the parameters brings a reduction of the coefficient. Reciprocity is best controlled by the back-link-probability (blp). The higher the blp the higher the reciprocity will be.

5.1.2 Speed

Our approach of generating test-data for social network applications should be less time-consuming compared to the process of gathering graph data from the original network. Nevertheless, generating big graphs with the Forest Fire Model may also take up much time. Generating graph data can even take up to days when a big amount of data has to be produced, e.g. millions of nodes and edges.

To be able to compare the graph generation process to the process of crawling Twitter's data the resulting network has to be similar in terms of nodes to edge ratio. We have shown that the clustering coefficients and the reciprocity match Twitter's figures in all of our evaluation sets and recorded the duration from the initialisation to the termination of the creation process. Measured times are included in Table 16.

The creation time is increasing exponentially to the graph's size because of the burning process of the Forest Fire Model. The shape of the line plot in Figure 34 indicates an exponent of 1.5 visualised by the green reference curve.

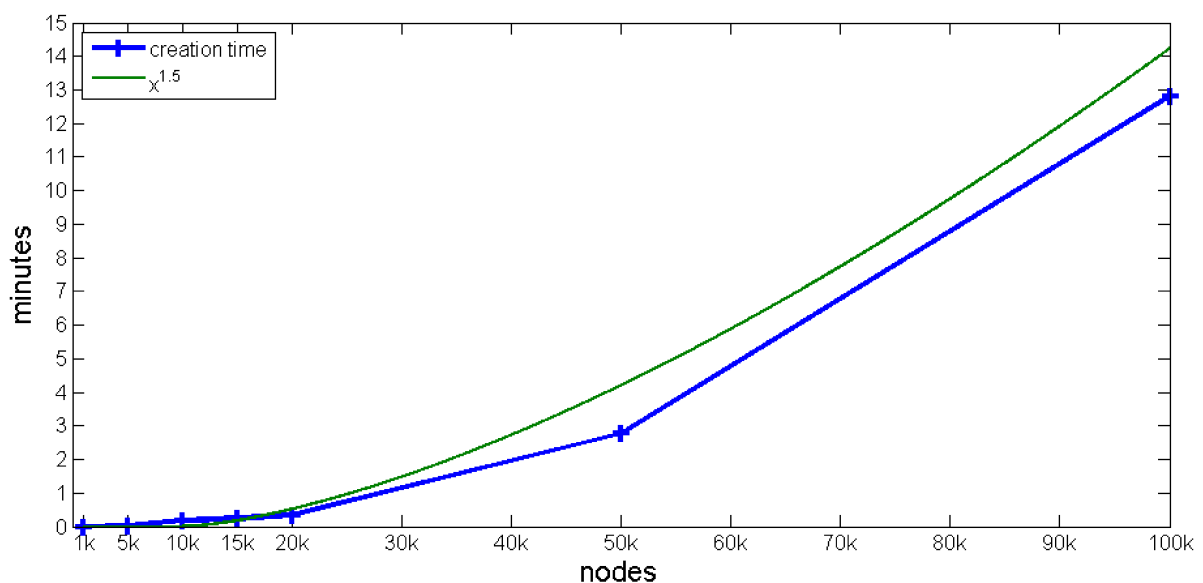


Figure 34: Graphs creation times

The recursive burning process can be tweaked to be faster by reducing the recursion depth. Still, it should be pointed out that if speed is the main concern, for example to generate large graphs in short time, the Forest Fire Model might not be the right choice and should probably be replaced by faster algorithms. Other algorithms like the nearest neighbour model could bring a speed boost. Again, see the works of Sala et al. [31] for a comparison of graph generation models.

5.1.3 Memory

When it comes to memory consumption, both, the edges and nodes have to be taken into account. Because node objects have more properties than edges, they weigh heavier in memory consumption. This can also be seen in Figure 35, showing three curves with different node to edge ratio. For the blue line we generated a graph with very few edges where the top value of 141 MB has a ratio of 101,000 nodes to 100,999 edges (almost 1:1). The opposite is true for the green line reaching 57 MB with 541 nodes and 292,140 edges (1:540). A realistic ratio setting has been chosen for the red curve taking up 148 MB with 99,991 nodes to 171,230 edges (1:1.7). By the form of the curves it can be concluded that there is a linear correlation between memory consumption and objects in the graph depending heavily on the number of nodes.

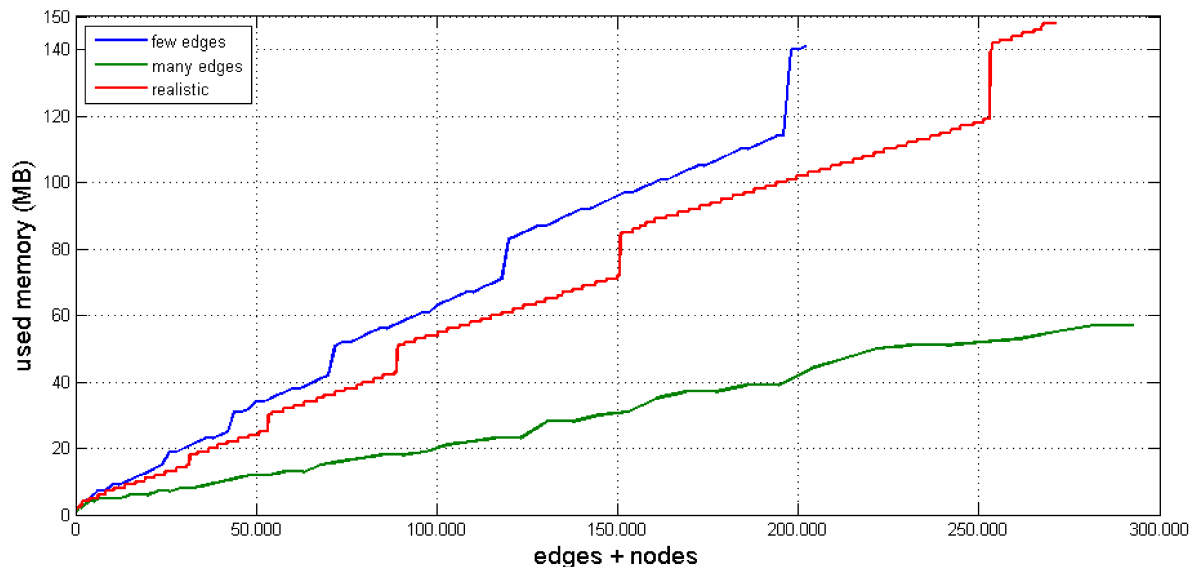


Figure 35: Memory consumption of graph generation depending on network structure

Either way, for the purpose of aiding the development, graphs up to 100,000 nodes should be sufficient and do not raise problems with memory consumption.

5.2 Communication Evaluation

Communication in social networks is a complex process that we wanted to reproduce only at a very basic stage. Still, we will answer the question of to what amount these randomly created messages meet observations in the real graph. To evaluate the usability in practice we will provide some benchmarks like memory consumption and duration of simulation.

5.2.1 Reality Conformance

In chapter 3.1.4 (Network Behaviour) we have described all metrics found in research and presented calculations for the distribution of different message types. The resulting figures are shown in Table 10. A description of message types and a picture of the brake-down can also be reviewed in this chapter. Additionally, we calculated the frequency of updates per user from [6], [10] and [36] to adapt the interval of messages. [10] provides us with average posts per day and a timespan for the dataset, [36] presents start and end of data collection, number of users in the set and total updates. From these figures we got diverse results of 0.30 and 1.24 respectively, suggesting the truth lies somewhere in between. The official figures presented by Twitter at a developers conference in 2010 [6] give us registered users and updates per day to calculate a number of 0.52 updates per day and user.

Message type	quota
Tweets	69%
Mentions	1%
DirectedTweets	1%
Retweets	5%
Replys	24%

Table 17: Quota of message types in Twitter

Now that we have an arrangement of metrics to compare with the output of our application, we tried to tweak simulation parameters to produce an output with similar conformation. It takes some time until the distribution of overall messages and message types reaches a stable point because the simulation process is initialized with updates of type "normal" tweet. We plotted a simulation of a 10,000 nodes network generated with the Forest Fire Model, which is included in the testbed in Figure 36. This run lasted for ten days and ten hours in simulation time which were seven seconds in real life. It can be seen that it took approximately 60 hours of continuous increase to get a stable level of updates per hour. Between 200 and 300 updates per hour in a 10,000 nodes-network equals to an update-frequency between 0.48 and 0.72 messages per day and user. The exact value for the whole simulation was 0.56 updates per person and day. This perfectly fits the specifications. Ratios of message types are presented in Table 18. Considering that we use a Gaussian normal distribution with a standard deviation of 0.1 to randomise all reactions, the data fits the specifications of Table 17 very well. Although one might find slight deviations here, they are justified by the randomisation steps implemented.

Message type	quota	benchmark
Tweets	72.9%	69%
Mentions	0.9%	1%
DirectedTweets	1.0%	1%
Retweets	5.4%	5%
Replys	19.6%	24%

Table 18: Ratios of message types in simulation (10,000 nodes; 10 days)

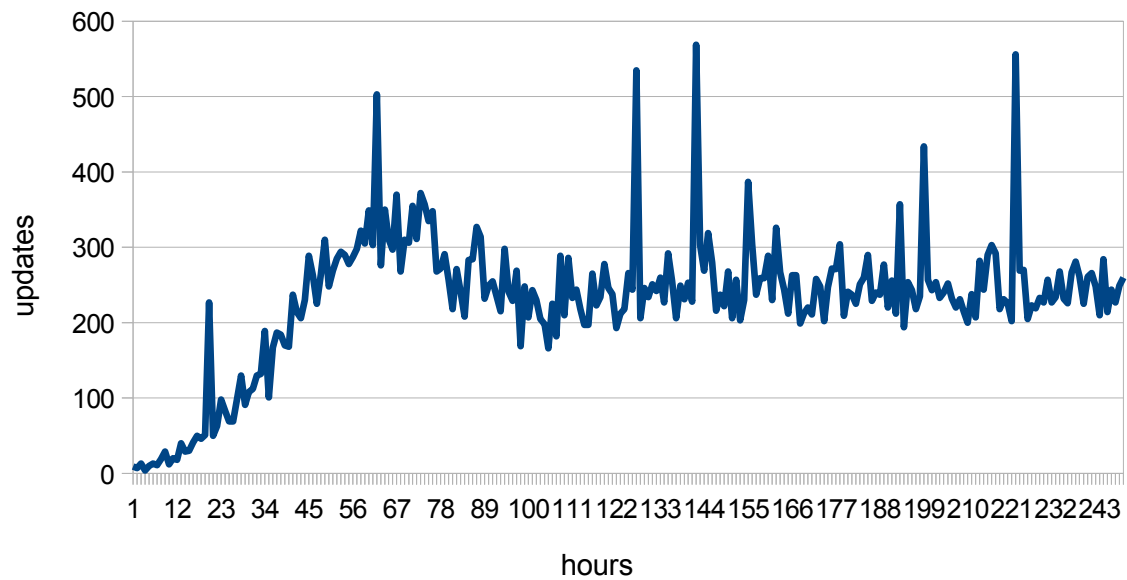


Figure 36: Number of updates in a simulation run (10,000 nodes; 10 days simulation)

It has to be said that the communication depends on many parameters and the graph structure it is working with. If the simulation runs on a very different network structure, even with the same parameters, the results may be completely different. The settings we used for this evaluations result are listed in the appendix (G.).

We created several graphs of different sizes in order to see how fragile the communication process is. Therefore we run simulations on these graphs with the same settings. We got almost the same figures and did not have to change settings to get satisfying results. To continue our thought process, this would mean that once a properties file is adjusted to the needs it can easily produce an unlimited number of different social network simulations with akin metrics. Moreover, one does not have to change settings to test on a bigger network. It is possible to expand the test data set in conformance to the development stage by only changing the size of the network through the number of nodes.

5.2.2 Speed

Speed is an important factor for the usability of our software in terms of generating multiple large graphs. For the development and evaluation of our software we had to create a large amount of graphs. Thereby we realised that a graph generation process has to be finished within a certain time limit, otherwise the process is getting infeasible for intense investigations.

The speed of communication does not only depend on the size of the graph but also on its structure and of course on the parameters of the simulation process itself. Still, we want to present a reference point of the time consumption. For this matter we first tested the simulation on the seven datasets we already used for evaluation of the graph generation process (5.1). As can be seen in Figure 37, the time it takes to run a simulation increases exponentially. The blue line shows the time it took to simulate ten days with different network sizes and the red line is a fitted exponential function. While 20 minutes is a justifiable amount of time to simulate a network with 100,000 nodes and to produce a number of 537,081 messages, the exponential form predicts a long waiting time when the size increases to the one of a real network with over 300,000 millions of nodes.

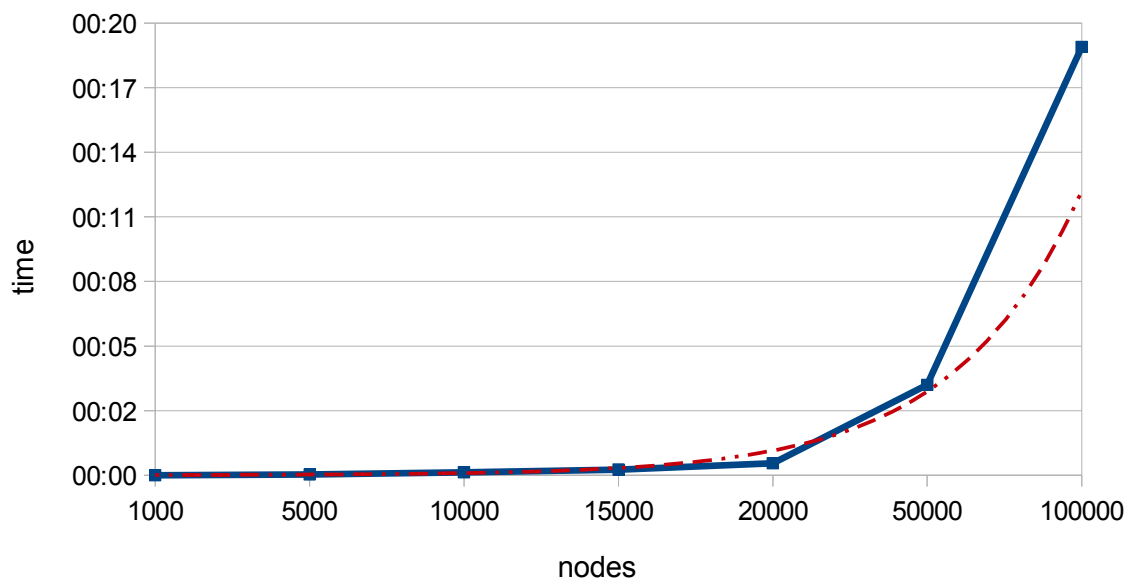


Figure 37: Duration of the simulation process

5.2.3 Memory Consumption

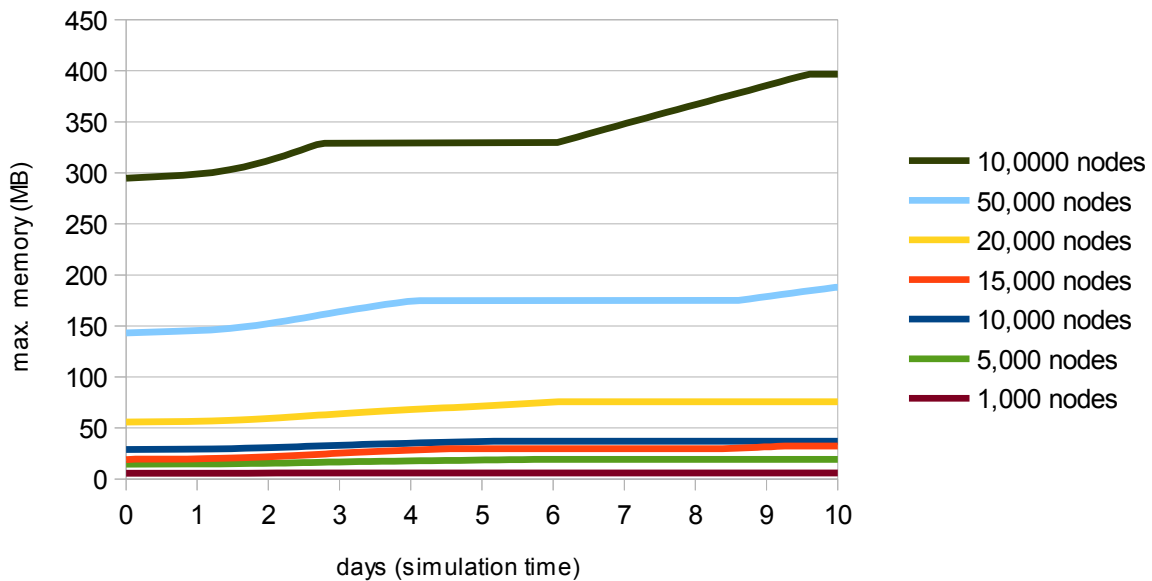


Figure 38: Memory consumption of different graph sizes over time

Memory consumption of the communication process is slightly increasing over time but should reach a stable limit when the update frequency gets to its normal range. Figure 38 shows the maximum memory used in a ten-days simulation run (simulation time). Again, the seven graphs and the simulation settings in prior evaluations are used. Initially, starting with memory used to load the graph and initialising the scheduler shows a linear increase until it reaches a stable maximum. For the biggest graphs (10,000 and 50,000 nodes) this limit is not reached within ten days of simulation time. All of these simulation runs had an average of 0,53 to 0,55 updates per node and day. One can see the average updates per hour in Table 19. Memory consumption of the simulation process is depending on the updates frequency. By comparing Table 19 with Figure 38, it can be seen that the gap between the two biggest networks is present in terms of memory consumption and in terms of updates per hour.

nodes	1,000	5,000	10,000	15,000	20,000	50,000	100,000
updates / node / day	0,5483	0,5358	0,5404	0,5304	0,5367	0,5426	0,5384
updates / hour	22,8	111,6	225,2	331,5	447,2	1130,4	2243,3

Table 19: Updates per hour in different graph sizes

6 Conclusion

The importance of social networks for the research community has grown in recent years also because of the increasing amount of people using these services. The expansion of social networks has led to the development of new software to access and interpret the structure of these networks. Thus, the creation, investigation and analysis of new software of that kind has become a major subject of interest within the scientific research community. Although many teams around the world are working on projects concerning the same popular social networks, there are only a few datasets available for testing new hypotheses and applications in the development stage. A literature research revealed that there are only two datasets for the Twitter network available for the public. By looking for reasons of this lack of provided training sets, we found out that it is a difficult and time-consuming task to gather data from social networks and that sharing this data raises some problems with the networks privacy policies. We came to the conclusion that another way has to be found to provide the scientific community with an appropriate amount of test data. Therefore, we came up with a solution that is applied in many other areas with similar problems. The project idea was to build a social network testbed to overcome this gap and speed up development.

In this thesis we introduced a solution for the widely-spread problem of test data distribution. The Twitter-Testbed is able to reproduce graph structures similar to the real Twitter network and runs a simulation of communication that meets the observations of the original service. This way we were able to produce an infinite amount of test data without having to experience limitations due to privacy concerns. The produced data can be accessed in the same way as the original one and keeping additional adaptations marginal.

Generally, choosing Twitter as the first network to simulate turned out to be a clever idea. Nevertheless, we had to face several drawbacks. First of all, we only found few studies about its structure and its behavioural aspects which made it difficult to adapt our results to the real network. One clear advantage of having chosen Twitter to simulate is the simplicity of the message types. This benefit was especially noticeable in the design process. Although we were able to adapt our system to the data we had, it is possible that they are not properly reflecting the real network and moreover bias our results. Considering the fact that, rather than replacing the original network, the purpose of the software is to produce a test bed, we are certain that the requirements are being met in most of the cases.

The Twitter-Testbed is able to produce random graph structures. Nevertheless, it can be parametrised to fit characteristics of real social networks. It is possible to produce all sizes of networks and save data for a graphML file for further analysis. Additionally, the opportunity to load graphs into the testbed enables the user to make use of already prepared networks or

even import data gathered from the real network instead of creating a random graph. Time and memory consumption are feasible to produce graphs up to 100,000 nodes instantly without the need of extra resources.

With the presented evaluation results we confirmed the similarities between the original network and the artificially created graphs. We have also proven the possibility of generating huge graphs consisting of millions of nodes and edges. Although there are difficulties in telling how similar the structures of original and artificial network really are, we are certain that our evaluated graphs exhibit a feasible amount of similarity that serves as a foundation for application testing.

Communication simulation has been added to the testbed and successfully used to produce realistic interaction patterns. We have implemented different message types and are able to produce a similar message flow to the one expected from the original network in terms of number of different messages and update frequency. With the possibility to access the resulting data in different ways it is ensured that the application can widely be used without the need to make changes to either our application nor the users development project. In addition to storing the data in a CSV- and XML-file, we connected the whole process to a Status.Net instance. Running the simulation combined with a Status.Net instance gives the opportunity to access the data with the same tools that can be applied on the original Twitter network. Thus, communication can be retrieved in real-time while the simulation process is running.

We showed that our simulation design is adjustable to produce the desired amount of messages and the given ratios of message types. The whole process from graph generation to output of simulated communication data can be controlled by a few parameters that are stored in a properties file. This way it is possible for people around the world to work on the same datasets without facing the problems of interchanging huge amounts of data by sharing the properties file used. The produced data can be shared and published without restrictions because no private data is being used.

We provided an overview of the characteristics of Twitter and justified our decisions as to why we used this network as the first network to simulate in our testbed. Then, we presented the design and implementation of a random graph generation process based on the Forest Fire Model to rebuilt Twitter's network structures. We came up with a simple simulation strategy and we have given an insight into the design and the implementation of our simulation procedure and showed the influencing properties. In the end, we provided some figures to show the conformance of our output of the observations in prior works and tested the application for usability factors.

To sum up, we have designed and implemented a comprehensive simulation of the Twitter network that can be adjusted to fit to requested parameters and is capable to produce an unlimited amount of test data. The accessibility of large amounts of test data and the ability to grow sizes of test-graphs with development progress has the potential to raise the quality of affected software and at the same time reduces deployment costs.

7 Future Work

We have shown that it is possible to generate a graph with the properties of a given real network in a reasonable amount of time concerning a test environment. We have also shown that the simulation of the communication can be done to produce an almost infinite number of test data. This project can be regarded as a prototype because there are some issues that could be improved. The following chapter is dedicated to the ideas of possible further enhancements.

7.1 Graph Generation

In the beginning, the lack of available statistical measures about the Twitter network surprised us authors. From the few works we found, only a very small portion investigated the overall network. Therefore, the calculated metrics were not all consistent. For most scientific investigations dealing with Twitter the structural properties are a crucial point that has to be taken with huge consideration. Nevertheless, the few available sources that we managed to find gave enough input to produce a prototype application and to evaluate if it is able to reproduce the data provided. One other important thing is that it has to be clear that the produced output is only a representation of the input we had. For further improvement or productive usage of this software we strongly recommend studying the structure of the desired network in detail. For other social network services there may be more scientific works available to base on. Nevertheless, SNSs are growing fast and are constantly changing their structure. It is therefore necessary to make sure the properties one retrieves from these works are up-to-date and calculated from a large enough source to draw a conclusion for the overall graph.

The graph generation process is open for further improvements concerning different aspects. It is possible that other models can produce an even more realistic representation of the desired network or have other advantages that favour their use for a specific goal. Some other models have been introduced in this paper (see 2.1 Graph Generation) and even more material can be found in literature. It would be interesting to test the influence that other graph generation models have on our application.

It is also possible to improve the Forest Fire Model that we used in this thesis the same way we did with our extension to the model (3.1.3 Extensions of the Model) and to further experiment with the parameter settings to achieve better results. In the end, the fact remains that all of these improvements are limited due to the quality of upcoming studies of the network structure and the conclusions that are drawn within these works.

Connecting the graph-building process to a database as a storage engine would not only offer an increase in the dimension limits by bypassing the size restrictions of the main memory, but also facilitate the calculation of many large-scale metrics. Moreover, it would offer an easy way to access the data for further investigations. Pre-fetch strategies could be used to increase the speed, depending on the intended model.

7.2 *Communication Simulation*

In this paper we tried to keep the simulation as simple as possible. Therefore, we only included parameters in the model that we decided to be indispensable to produce reasonable message flow. Depending on the requirements there always has to be found a balance between simplicity and speed on the one hand and customisability and authenticity on the other hand. Either way, here are some possible extensions to the simulation that we would like to add in future works:

Hashtags are a common way in Twitter to denote the topics of a message. Implementing a hashtag repertoire that populates tags into random messages would not only enhance the message body with a new feature to be analysed, but also could provide a decision-making aid for the reaction part. By adding some kind of profile information to the node including topics of interest, it would be possible to impact the reaction strategy by the accordance of hashtags and topics of interest.

Concerning networks that should represent a group of people who are spread all over the world, an important aspect is time variation. There are people located on one side of the globe who are active because it is day, others are passive because it is night and they might be asleep. The simulation would be more realistic if a day- and night-cycle were added to the message flow.

Including some kind of location position into the user's profile could influence the activity time of that node depending on time zones. Additionally, the geographic closeness of two nodes could influence the probability of a reaction. This geographic location information could be set by clustering the node space in strongly connected components, so called communities [47].

The timing of messages is what really defines a viral spread from within the network. An improvement in the calculation of the point-in-time when a reaction is taking place could make it possible to reflect this interesting phenomena in the simulation data.

In the current state simulation and graph generation are completely separated from each other. The reason is because it makes it possible to save and load graphs independent from the simulation process. For small networks, let us assume that one network only has 1,000 nodes, it is feasible to make a simulation on a non-growing snapshot of a graph. Depending

on the question that is being raised, it might not make a difference even with larger graphs. In fact, one aspect of social networks that is often mentioned, not only in scientific papers, is that they are growing fast. Thinking of a social graph containing over ten thousands of nodes, it is very unlikely that there is no change in network structure even in a small time-frame that only includes some hours. Although the study of dynamics graphs of social network graphs has only recently begun, they would certainly depict reality more appropriately if the structure changes during the simulation process. This could be achieved by implementing an additional event type that adds a new node to the graph when triggered.

8 Appendix

A. References

- [1] Facebook Ireland Limited, „Facebook Statistik“, *Facebook*, 23-Aug-2011. [Online]. Available: <http://www.facebook.com/press/info.php?statistics>. [Accessed: 23-Aug-2011].
- [2] WebMediaBrands Inc., „Facebook Sees Big Traffic Drops in US and Canada as It Nears 700 Million Users Worldwide“, *Inside Facebook*. [Online]. Available: <http://www.insidefacebook.com/2011/06/12/facebook-sees-big-traffic-drops-in-us-and-canada-as-it-nears-700-million-users-worldwide/>. [Accessed: 23-Aug-2011].
- [3] „Facebook’s Ad Revenue Hit \$1.86B for 2010“. [Online]. Available: <http://mashable.com/2011/01/17/facebook-ad-revenue-hit-1-86b-for-2010/>. [Accessed: 10-Nov-2011].
- [4] R. T. Kreuzer und J. Hinz, *Möglichkeiten und Grenzen von Social Media Marketing*. December, 2010.
- [5] „Twitter Facts & Figures (history & statistics) | Website Monitoring Blog“. [Online]. Available: <http://www.website-monitoring.com/blog/2010/05/04/twitter-facts-and-figures-history-statistics/>. [Accessed: 10-Nov-2011].
- [6] „Twitter Finally Reveals All Its Secret Stats“. [Online]. Available: <http://www.businessinsider.com/twitter-stats-2010-4#twitter-now-has-106-million-users-1>. [Accessed: 10-Nov-2011].
- [7] T. Johansmeyer, „200 Million Twitter Accounts... But How Many Are Active? - SocialTimes.com“, *Social Times*. [Online]. Available: http://socialtimes.com/200-million-twitter-accounts-but-how-many-are-active_b36952. [Accessed: 28-Nov-2011].
- [8] „Twitter Blog: #numbers“. [Online]. Available: <http://blog.twitter.com/2011/03/numbers.html>. [Accessed: 28-Nov-2011].
- [9] G. Kossinets und D. J. Watts, „Empirical analysis of an evolving social network“, *Science*, Bd. 311, Nr. 5757, S. 88, 2006.
- [10] B. A. Huberman, D. M. Romero, und F. Wu, „Social networks that matter: Twitter under the microscope“, *First Monday*, Bd. 14, Nr. 1, S. 8, 2009.
- [11] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, und B. Bhattacharjee, „Measurement and analysis of online social networks“, in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, S. 29–42.
- [12] P. Mika, „Flink: Semantic web technology for the extraction and analysis of social networks“, *Web Semantics: Science, Services and Agents on the World Wide Web*, Bd. 3, Nr. 2?3, S. 211–223, 2005.
- [13] J. Heer und D. Boyd, „Vizster: Visualizing online social networks“, 2005.
- [14] H. Kwak, C. Lee, H. Park, und S. Moon, „What is Twitter, a social network or a news media?“, in *Proceedings of the 19th international conference on World wide web*, 2010, S. 591–600.
- [15] L. Backstrom, D. Huttenlocher, J. Kleinberg, und X. Lan, „Group formation in large social networks: membership, growth, and evolution“, in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, S. 44–54.
- [16] L. Backstrom, C. Dwork, und J. Kleinberg, „Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography“, in *Proceedings of the 16th international conference on World Wide Web*, 2007, S. 181–190.
- [17] A. Narayanan und V. Shmatikov, „How to break anonymity of the netflix prize dataset“, *Arxiv preprint cs/0610105*, 2006.
- [18] M. Treiber, D. Schall, S. Dustdar, und C. Scherling, „Tweetflows-Flexible Workflows with Twitter“, 2011.
- [19] D. Boyd, S. Golder, und G. Lotan, „Tweet, Tweet, Retweet: Conversational Aspects of Retweeting on Twitter“, *Hawaii International Conference on System Sciences*, Bd. 0, S. 1–10, 2010.
- [20] M. E. . Newman, D. J. Watts, und S. H. Strogatz, „Random graph models of social networks“, *Proceedings of the National Academy of Sciences of the United States of America*, Bd. 99, Nr. Suppl 1, S. 2566, 2002.
- [21] P. Erdős, A. Rényi, „On random graphs“, in *Publ. Math. Debrecen* 6, 1959, S. 290–297.
- [22] M. Molloy und B. Reed, „A Critical Point for Random Graphs with a Given Degree Sequence“, in *Random Structures and Algorithms* 6, 1995, S. 161–180.
- [23] R. Kumar, J. Novak, und A. Tomkins, „Structure and evolution of online social networks“, *Link Mining: Models, Algorithms, and Applications*, S. 337–357, 2010.

Appendix

- [24] J. Leskovec, J. Kleinberg, und C. Faloutsos, „Graphs over time: densification laws, shrinking diameters and possible explanations“, in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, S. 177–187.
- [25] J. Leskovec, J. Kleinberg, und C. Faloutsos, „Graph evolution: Densification and shrinking diameters“, *ACM Transactions on Knowledge Discovery from Data (TKDD)*, Bd. 1, Nr. 1, S. 2, 2007.
- [26] J. Leskovec, K. J. Lang, A. Dasgupta, und M. W. Mahoney, „Statistical properties of community structure in large social and information networks“, in *Proceeding of the 17th international conference on World Wide Web*, 2008, S. 695–704.
- [27] A. L. Barabási und R. Albert, „Emergence of scaling in random networks“, *Science*, Bd. 286, Nr. 5439, S. 509, 1999.
- [28] E. Ravasz und A. L. Barabási, „Hierarchical organization in complex networks“, *Physical Review E*, Bd. 67, Nr. 2, S. 026112, 2003.
- [29] A. Flaxman, A. Frieze, und J. Vera, „A geometric preferential attachment model of networks“, *Algorithms and Models for the Web-Graph*, S. 44–55, 2004.
- [30] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, und Z. Ghahramani, „Kronecker graphs: An approach to modeling networks“, *The Journal of Machine Learning Research*, Bd. 11, S. 985–1042, 2010.
- [31] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, und B. Y. Zhao, „Measurement-calibrated graph models for social network experiments“, in *Proceedings of the 19th international conference on World wide web*, 2010, S. 861–870.
- [32] A. Vázquez, „Growing network with local rules: Preferential attachment, clustering hierarchy, and degree correlations“, *Physical Review E*, Bd. 67, Nr. 5, S. 056104, 2003.
- [33] P. Mahadevan, D. Krioukov, K. Fall, und A. Vahdat, „Systematic topology analysis and generation using degree correlations“, in *ACM SIGCOMM Computer Communication Review*, 2006, Bd. 36, S. 135–146.
- [34] T. Karagiannis und M. Vojnovic, „Behavioral profiles for advanced email features“, in *Proceedings of the 18th international conference on World wide web*, 2009, S. 711–720.
- [35] D. Centola, „The spread of behavior in an online social network experiment“, *science*, Bd. 329, Nr. 5996, S. 1194, 2010.
- [36] A. Java, X. Song, T. Finin, und B. Tseng, „Why we twitter: understanding microblogging usage and communities“, in *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, 2007, S. 56–65.
- [37] B. Krishnamurthy, P. Gill, und M. Arlitt, „A few chirps about twitter“, in *Proceedings of the first workshop on Online social networks*, 2008, S. 19–24.
- [38] M. Cha, H. Haddadi, F. Benevenuto, und K. P. Gummadi, „Measuring User Influence in Twitter: The Million Follower Fallacy“, in *In Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*, 2010.
- [39] C. Honey und S. C. Herring, „Beyond microblogging: Conversation and collaboration via Twitter“, in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, 2009, S. 1–10.
- [40] H. Psaiar, L. Juszczyk, F. Skopik, D. Schall, und S. Dustdar, „Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems“, in *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, 2010, S. 164 –173.
- [41] B. Page und W. Kreutzer, *The Java simulation handbook*. Shaker, 2005.
- [42] D. J. Watts und S. H. Strogatz, „Collective dynamics of ‘small-world’ networks“, *Nature*, Bd. 393, Nr. 6684, S. 440–442, Juni 1998.
- [43] A. Clauset, C. R. Shalizi, und M. E. . Newman, „Power-law distributions in empirical data“, *Arxiv preprint arXiv:0706.1062*, 2007.
- [44] „Twitter Conversation Statistics - Power of Replies and Retweets“. [Online]. Available: <http://www.sysomos.com/insidetwitter/engagement/>. [Accessed: 27-Feb-2012].
- [45] Mathieu Bastian, Sebastien Heymann, und Mathieu Jacomy, „Gephi: An Open Source Software for Exploring and Manipulating Networks“, 2009.
- [46] M. A. Smith, B. Shneiderman, N. Milic-Frayling, E. Mendes Rodrigues, V. Barash, C. Dunne, T. Capone, A. Perer, und E. Gleave, „Analyzing (social media) networks with NodeXL“, in *Proceedings of the fourth international conference on Communities and technologies*, New York, NY, USA, 2009, S. 255–264.
- [47] J. Leskovec, K. J. Lang, A. Dasgupta, und M. W. Mahoney, „Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters“, *Internet Mathematics*, Bd. 6, Nr. 1, S. 29–123, 2009.

B. Twitter-Testbed Properties File

##### General settings ##### seed = 325987520 debug = false printStatistics = true	##### (random generator seed) (Print debugging information) (Print basic metrics)
#### Forest Fire model settings #### pf = 0.8 pb = 0.7 blp = 0.42 initVertices = 1 maxNodes = 100 maxAmbassadors = 1 limitDepth = 100	##### (forward burning probability) (backward burning probability) (back link probability) (number of vertices to start with) (number of nodes to create) (max Ambassadors to be linked to) (Limits depth of burning process)
##### Initialiser settings ##### graphForceInitialise = false; actSourceRatio = 0.1 actSeekerRatio = 0.1 avtWeekdayRatio = 0.6 avtWeekendRatio = 0.2 avtAlldayRatio = 0.2	##### (override graph properties) (ratio specifying a Infor-Source) (ratio specifying a Info-Seeker) (ratio, weekday of all availability types) (ratio, weekend of all availability types) (ratio, allday of all availability types)
##### Scheduler settings ##### startTime = 0 duration = 1000000000 durationHours = 0 durationDays = 10	##### (model start time) (max duration of the simulation) (duration in hours) (duration in days)
randomness = 0.1 maxInterval = 400000000 cumulation = 0.3	(standard deviation for randomisation) (max time between events in milliseconds) (probability of cumulation)
updateMentionRatio = 0.027 updateDirectedRatio = 0.51	(ratio of updates including mentions) (ratio of mentions being directed)
responseTweet = 0.33 responseDirectedTweet = 0.2 responseMention = 0.3 responseRetweet = 0.3 responseReply = 0.23 responseRetweetRatio = 0.4	(response threshold for tweets) (response threshold for directed tweets) (response threshold for mentions) (response threshold for retweets) (response threshold for replies) (ratio of reactions being retweets)
##### Input / Output settings ##### graphLoadPath = C:\\test.graphml graphSavePath = C:\\test.graphml convCSVSavePath = C:\\testCSV.csv convXMLSavePath = C:\\testCSV.csv	##### (Load graph from file) (Save GraphML file) (Save CSV Message list) (Save conversation to XML file)
##### Status.Net Settings ##### host = localhost/statusnet username = twittertestbed password = twittertestbed url = http://localhost/StatusNet/	##### (MySQL host address incl db name) (DB username) (DB password) (Base URL of Status.Net instance)

Listing 15: Sample properties file

C. Sample graphML File

Sample graphML file produced by the Twitter-Testbed application.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/graphml">

  <key id="avt" for="node">
    <desc>Availability Types describe when a user is usually using the service</desc>
    <default>0</default>
  </key>
  <key id="acl" for="node">
    <desc>Activity Level is an indicator for how often a user gets active (tweets)</desc>
    <default>0</default>
  </key>
  <key id="avl" for="node">
    <desc>The Availability level is an indicator of how often a user checks the latest messages.</desc>
    <default>0</default>
  </key>
  <key id="act" for="node">
    <desc>Activity Type is used to mark the users intention in using the network</desc>
    <default>0</default>
  </key>
  <key id="c" for="edge">
    <desc>The closeness value is a synonym for friendship tightness</desc>
    <default>0</default>
  </key>

  <graph edgedefault="directed">
    <desc>This is a Graph generated by a model implemented by Andreas Scharf
      at the Distributed System Groups of the Technical University Vienna.
      (Vertices: 5 Edges: 10)</desc>

    <node id="2">
      <data key="avt">1</data>
      <data key="acl">0.4589039681992828</data>
      <data key="avl">0.5140197147718047</data>
      <data key="act">0</data>
    </node>
    <node id="1">
      <data key="avt">0</data>
      <data key="acl">0.8534815655725531</data>
      <data key="avl">0.02407301788226368</data>
      <data key="act">1</data>
    </node>
    <node id="3">
      <data key="avt">0</data>
      <data key="acl">0.45322491883345195</data>
      <data key="avl">0.9407195036895842</data>
      <data key="act">1</data>
    </node>
    <node id="4">
      <data key="avt">2</data>
      <data key="acl">0.7673385391215849</data>
      <data key="avl">0.8818601759865199</data>
      <data key="act">2</data>
    </node>
    <node id="5">
      <data key="avt">0</data>
      <data key="acl">0.7117114942639918</data>
      <data key="avl">0.42228491860858364</data>
      <data key="act">1</data>
    </node>

    <edge id="5" source="1" target="4"><data key="c">0.05788996773995292</data></edge>
    <edge id="8" source="4" target="5"><data key="c">0.693609748639321</data></edge>
    <edge id="0" source="2" target="1"><data key="c">0.6975718207087579</data></edge>
    <edge id="9" source="5" target="1"><data key="c">0.3953000172947976</data></edge>
    <edge id="6" source="4" target="3"><data key="c">0.7219677364814028</data></edge>
    <edge id="2" source="2" target="3"><data key="c">0.38869866008561116</data></edge>
    <edge id="4" source="4" target="1"><data key="c">0.80594617523336</data></edge>
    <edge id="3" source="4" target="2"><data key="c">0.330019988437524</data></edge>
    <edge id="7" source="5" target="4"><data key="c">0.8390112035849809</data></edge>
  </graph>
</graphml>
```

Listing 16: Sample graphML file

D. Source Code Samples

```
protected double calcResponseWillingness( TwitterScheduler sched,
                                         TwitterNode sender,
                                         TwitterNode reciever ) {
    Graph<TwitterNode, TwitterEdge> g = sched.getGraph();
    Edge e;
    if((e = g.findEdge(reciever, sender)) != null) {
        double c = e.getClosenessValue();
        double acl = reciever.getActivityLevel();
        double fx = (c*acl/2);
        double gx = fx + sched.rand.nextGaussian()*sched.getRandomness();

        return Math.abs(gx);
    } else {
        return 0;
    }
}
```

Listing 17: Source code of the `calcResponseWillingness` function

```
public EventNote getNextRandomTweet(TwitterNode sender) {
    double acl = sender.getActivityLevel();
    double avl = sender.getAvailabilityLevel();
    double act = sender.getActivityTypeValue();
    double rand = this.rand.nextDouble();
    int numFollowees = this.getGraph().getSuccessorCount(sender);
    Event e;
    // Decide whether this message should include a mentioning
    if(rand<this.mentionsRatio && this.getGraph().getSuccessorCount(sender)>0) {
        // Choose one of the followees randomly as mentioning
        TwitterNode[] nodes = { this.getGraph().getSuccessors(sender)
                               .toArray()[this.rand.nextInt(numFollowees)]};
        // Decide if this mention is also a directed tweet
        if (rand < (this.mentionsRatio*this.directedRatio)) {
            e = new EventDirectedTweet(sender, this.getRandomDirectedMsg(
                                         nodes[0].getId()), nodes );
        } else {
            e = new EventMention( sender, this.getRandomMentionMsg(
                                         nodes[0].getId()), nodes);
        }
    } else {
        e = new EventTweet(sender, this.getRandomMsg());
    }
    // calculate time of the event
    double fx = ((acl+avl+act)/3);
    double gx = fx + this.rand.nextGaussian()*this.randomness;
    long eventTime = Math.abs(Math.round((1-gx) * this.getMaxIntervall()));
    EventNote n = new EventNote(eventTime, e);
    return n;
}
```

Listing 18: Source code of the `getNextRandomTweet()` function

Appendix

```
insertNotice = conn.prepareStatement("INSERT INTO notice VALUES" +
    "(?, ?, ?, ?, ?, NULL, FROM_UNIXTIME(?), NOW(), ?, 1, '"+source+"', ?, " +
    "NULL, NULL, NULL, NULL, ?, ?, ?, ?);");
insertReply = conn.prepareStatement("INSERT INTO reply VALUES" +
    "(?, ?, NOW(), NULL);");
insertConversation = conn.prepareStatement("INSERT INTO conversation VALUES" +
    "(NULL,CONCAT('"+apiURL+"index.php/conversation/', " +
    "(SELECT MAX(c2.id)+1 FROM conversation c2)),NOW(),NOW());" +
    ,Statement.RETURN_GENERATED_KEYS);

int conv;
insertNotice.setInt(1, id); // id
insertNotice.setInt(2, sender); // profile_id
insertNotice.setString(3, apiURL+"index.php/notice/"+id); // uri
insertNotice.setString(4, msg); // content
insertNotice.setString(5, msg); // rendered
insertNotice.setLong(6, time/1000); // time of the post (only till seconds part)
if(type=="reply"){
    insertReply.setInt(1, id); // Notice ID
    insertReply.setInt(2, mentions[0]); // ID of referenced message
    insertReply.executeUpdate();
}
// if this message refers to another message use its conversation
if(origMsg>-1) {
    queryConversation.setInt(1, origMsg);
    ResultSet rs = queryConversation.executeQuery();
    conv = rs.getInt(1);
    insertNotice.setInt(7, mentions[0]); // reply_to
    insertNotice.setInt(9, origMsg); // repeat_of
} else {
    insertConversation.executeUpdate();
    ResultSet rs = insertConversation.getGeneratedKeys();
    conv = rs.getInt(1);
    insertNotice.setNull(7,1); // reply_to
    insertNotice.setNull(9,1); //repeat_of
}
insertNotice.setInt(8, conv); // conversation
insertNotice.setString(10, objType); // object_type
insertNotice.setString(11, verb); // verb
insertNotice.setInt(12, 1); // scope
insertNotice.executeUpdate();
insertNotice.clearParameters();
// If the message includes mentions
if(mentions!=null) {
    for(int m:mentions) {
        insertReply.setInt(1, id); // Notice ID
        insertReply.setInt(2, m); // ID of the mentioned user
        insertReply.addBatch();
    }
    insertReply.executeBatch(); // Execute insert query
}
//put the notice in the followers inbox
if(scope != 2) {
    updateInbox.setBytes(1,ByteBuffer.allocate(4).putInt(id).array());
    updateInbox.setInt(2, sender);
    updateInbox.execute();
}
```

Listing 19: Source code snippet of Status.Net connector

E. Sample XML Output File

```

<?xml version="1.0" encoding="UTF-8"?>
<cxml:conv xmlns:cxml="http://www.tuwien.ac.at/infosys/TwitterTestbed/ConvXML">
  <cxml:event type="tweet">
    <cxml:id>1</cxml:id>
    <cxml:msg>p5sbtd92fhhy</cxml:msg>
    <cxml:sender>1</cxml:sender>
  </cxml:event>
  <cxml:event type="directedTweet">
    <cxml:id>2</cxml:id>
    <cxml:msg>@user1 3xjy7jsjimn</cxml:msg>
    <cxml:sender>2</cxml:sender>
    <cxml:mentions>
      <cxml:mention>1</cxml:mention>
    </cxml:mentions>
  </cxml:event>
  <cxml:event type="tweet">
    <cxml:id>3</cxml:id>
    <cxml:msg>rh19s4o14y1r</cxml:msg>
    <cxml:sender>3</cxml:sender>
  </cxml:event>
  <cxml:event type="retweet">
    <cxml:id>4</cxml:id>
    <cxml:msg>RT: p5sbtd92fhhy</cxml:msg>
    <cxml:sender>3</cxml:sender>
    <cxml:origmsg>1</cxml:origmsg>
    <cxml:mentions>
      <cxml:mention>1</cxml:mention>
    </cxml:mentions>
  </cxml:event>
  <cxml:event type="reply">
    <cxml:id>5</cxml:id>
    <cxml:msg>@user2 3jtu8ahveqw</cxml:msg>
    <cxml:sender>1</cxml:sender>
    <cxml:origmsg>2</cxml:origmsg>
    <cxml:mentions>
      <cxml:mention>2</cxml:mention>
    </cxml:mentions>
  </cxml:event>
  <cxml:event type="reply">
    <cxml:id>6</cxml:id>
    <cxml:msg>@user1 3mx1j6och0</cxml:msg>
    <cxml:sender>2</cxml:sender>
    <cxml:origmsg>2</cxml:origmsg>
    <cxml:mentions>
      <cxml:mention>1</cxml:mention>
    </cxml:mentions>
  </cxml:event>
  <cxml:event type="tweet">
    <cxml:id>7</cxml:id>
    <cxml:msg>9jpx0kqojk94</cxml:msg>
    <cxml:sender>3</cxml:sender>
  </cxml:event>
  <cxml:event type="mention">
    <cxml:id>8</cxml:id>
    <cxml:msg>17b47xyw @user1 4itz</cxml:msg>
    <cxml:sender>3</cxml:sender>
    <cxml:mentions>
      <cxml:mention>1</cxml:mention>
    </cxml:mentions>
  </cxml:event>
</cxml:conv>

```

Listing 20: XML output format

F. Evaluation Graphs

Parameters used for graph generation using the Forest Fire Model of Twitter-Testbed:

Graph:	1,000	5,000	10,000	15,000	20,000	50,000	100,000
seed	325987520	325987520	325987520	325987520	325987520	325987520	325987520
pf	0.95	0.95	0.95	0.95	0.95	0.95	0.95
pb	0.9	0.89	0.89	0.89	0.89	0.89	0.89
blp	0.49	0.49	0.49	0.49	0.485	0.485	0.485
init Vertices	1	1	1	1	1	1	1
maxAmbassadors	1	1	1	1	1	1	1
maxNodes	1,000	5,000	10,000	15,000	20,000	50,000	100,000
limitDepth	100	100	100	100	100	100	100

Table 20: Evaluation graphs settings

With these properties applied to the application it is possible to recreate the exact same network structures referred to in the evaluations chapter (chapter 5).

	Graph:	1,000	5,000	10,000	15,000	20,000	50,000	100,000
	Nodes	1,000	5,000	10,000	15,000	20,000	50,000	10,000
	Edges	1,700	8,543	17,168	25,818	34,312	85,550	171,435
	Reciprocity	1	1	1	1	1	1	1
	ExecutionTime	00:00:00	00:00:02	00:00:11	00:00:15	00:00:20	00:02:46	0:12.49
nodeXL	Diameter (undirected)	25	36	33	38	35	40	43
nodeXL	Avg geodesic dist	10.364	12.794	13.747	15.505	15.762	17.138	18.714
nodeXL	Density	0.001698	0.000342	0.000172	0.000115	0.000086	0.000034	0.000017
nodeXL	Min in-Degree	0	0	0	0	0	0	0
nodeXL	Max in-Degree	10	16	16	18	22	19	23
nodeXL	Avg in-Degree	1.698	1.708	1.717	1.721	1.716	1.711	1.714
nodeXL	Med in-Degree	1	1	1	1	1	1	1
nodeXL	Min out-Degree	1	1	1	1	1	1	1
nodeXL	Max out-Degree	9	11	12	10	12	13	17
nodeXL	Avg out-Degree	1.698	1.708	1.717	1.721	1.716	1.711	1.714
nodeXL	Med out-Degree	1	1	1	1	1	1	1
nodeXL	Min clustering coef	0	0	0	0	0	0	0
nodeXL	Max clustering coef	1	1	1	1	1	1	1
nodeXL	Avg clustering coef	0.100	0.115	0.121	0.116	0.117	0.112	0.116
nodeXL	Med clustering coef	0	0	0	0	0	0	0
Gephi	Avg degree	1.698	1.708	1.717	1.721	1.716	1.711	1.714
Gephi	Network diameter (directed)	19	33	26	31	31	35	38
Gephi	Avg path length	7.369	10.179	10.818	12.305	13.009	13.939	15.789

Table 21: Statistical measures of seven evaluation graphs

The provided figures in Table 21 are calculated from within the Java application, nodeXL and Gephi as labelled in the first column.

G. Properties File for the Evaluation Run

```
#####  
# Sample Properties File #  
# for TwitterTestbed v0.1 #  
#####  
  
##### General settings #####  
seed = 325987520  
debug = false  
printStatistics = true  
  
##### ForestFire model settings #####  
pf = 0.95  
pb = 0.89  
blp = 0.485  
initVertices = 1  
maxAmbassadors = 1  
maxNodes = 10000  
limitDepth = 100  
  
##### Initialiser settings #####  
graphForceInitialise = false;  
actSourceRatio = 0.1  
actSeekerRatio = 0.1  
avtWeekdayRatio = 0.6  
avtWeekendRatio = 0.2  
avtAlldayRatio = 0.2  
  
##### Scheduler settings #####  
startTime = 0  
duration = 0  
durationHours = 10  
durationDays = 10  
randomness = 0.1  
maxInterval = 400000000  
cumulation = 0.3  
updateMentionRatio = 0.027  
updateDirectedRatio = 0.51  
  
responseTweet = 0.33  
responseDirectedTweet = 0.2  
responseMention = 0.3  
responseRetweet = 0.3  
responseReply = 0.23  
responseRetweetRatio = 0.4  
  
##### Input / Output settings #####  
graphLoadPath = D:\\TwitterTestbed\\finalRevision\\testgraph_100000Nodes.graphml  
#graphSavePath = D:\\TwitterTestbed\\finalRevision\\testgraph_random.graphml  
#convCSVSavePath = D:\\TwitterTestbed\\finalRevision\\testgraph_random_com.csv  
  
##### Status.Net Settings #####  
# Mysql host adress including db name  
#host = localhost/statusnet  
username = twittertestbed  
password = twittertestbed  
url = http://localhost/StatusNet/
```

Listing 21: Properties file used for evaluation runs