GTNA - A Framework for the Graph-Theoretic Network Analysis

Benjamin Schiller Dirk Bradler Immanuel Schweizer
Max Mühlhäuser Thorsten Strufe
TU Darmstadt, Darmstadt, Germany
{lastname[at]cs.tu-darmstadt.de}

Abstract

Concise and reliable graph-theoretic analysis of complex networks today is a cumbersome task, consisting essentially of the adaptation of intricate libraries for each specific problem instance. The growing number of complex metrics that have been proposed in the last years, which mainly gain significance due to the increasing computational capabilities at hand, have led to important new insights in the field. However, they have solely been implemented as single algorithms, each specialized for the purpose of calculating exactly the targeted metric for a selected type of network graph. A comprehensive, extensible tool for the concise evaluation of graphs is currently not available. For this purpose we introduce the Graph-Theoretic Network Analyzer (GTNA), an efficient, Java-based toolkit for the comprehensive analysis of complex network graphs. GTNA, while already including the main metrics that are used to analyze the complex networks in computer science today, is simple to extend through a well defined plugin interface for metrics, network descriptions and network generator models. Throughout the paper we present the design and simple extensibility of GTNA, as well as the network models and metrics that are already implemented and give examples of its scalability and performance.

Keywords: Graph theory, Networks, Analysis, Peer-to-peer

1. INTRODUCTION

Complex networks are omnipresent in our daily life. Peer-to-Peer (P2P) technology, e.g., which leverages on distributed resources and hence creates highly complex networks, is already established in several application domains. The self-organizing capabilities of P2P enable fast deployment without the need for manual configuration of individual peers. Especially in an unreliable and fast changing environment, P2P technology can be used as a basis for communication infrastructures. Wireless sensor networks, to state another example, are deployed in cars, houses and all types of open territories. Each wireless sensor has very limited capabilities, but in the conjunction, creating an instance of a wireless sensor network, even complex tasks can be accomplished. The most prevalent complex network is the Internet. Measurements on the level of autonomous systems show that it is dynamic and still growing. In addition, common laptops, cell-phones and PDAs may join a wireless ad-hoc network and communicate instantaneously. All distributed communication structures have to face several challenges considering routing performance, message loss, resilience against attacks, node failures, scalability and efficiency. In order to analyse and evaluate the behavior of such distributed systems, several network simulation engines have been developed. It seems like simulation is the research tool of choice for a variety of network related research challenges. Nevertheless, with rising popularity of simulations, the credibility of the results has decreased. By surveying MANET simulation studies of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc) [1] significant unresolved challenges were found. One key finding is, that a comparison of simulation studies is hardly possible. Only 15% of the simulation studies of the published ad hoc papers between 2000 and 2005 were repeatable [1]. In order to achieve a fair comparison of different kinds of simulation results, a common simulation engine, a common workload as well as a set of important metrics need to be supported and maintained by the network research community. Nevertheless, the better known simulators support the storage of network snapshots at certain intervals. Even though the dynamic behavior of the network is not captured in a snapshot, a graph theoretic analysis of the underlying topology is feasible. In this paper we present GTNA, a framework for the graph-theoretic network analysis. GTNA is especially developed for the comparison of structural properties found in complex communication networks.

The remainder of the paper is structured as follows: Section 2. discusses related work. In section 3. we present the basic approach of the simulation workflow, currently available network topologies and the implemented metrics. In section 4. the architecture of our proposed framework is presented in detail. Configuration and extension of both, network topologies and metrics are shown in section 5. An example configuration is presented in section 6. Measurements of runtime behavior are presented in section 7., followed by summary and outlook in section 8.

2. RELATED WORK

Making real world testbeds for complex technological networks scalable is a hard and expensive task. Simulation tools are widely accepted to overcome this problem.

Different solutions for simulation have emerged over the

years. Tailor-made solutions for one application are common, like PeerSim [2] and PeerfactSim [3] for Peer-to-Peer networks. Domain-independent solutions like PlanetSim [4] can simulate different networks but are restricted to certain layers. Powerful frameworks like JIST [5], Omnet++ [6] and NS2/NS3 [7, 8] can be extended to serve almost any kind of application. There are even more basic approaches like Parsec [9] that specify programming languages for parallel system simulation.

The main goal of any simulation is to estimate the outcome of an abitrary system without the need to actually build that system. So a major part of any simulation process involves the need to analyse the results of the simulation. The simulation of complex technological networks can be analysed either by *dynamic* or *structural* properties. Dynamical properties are for example throughput or message delay. Structural properties focus on the analysis of underlying graph properties such as the degree distribution.

The simulation tools mentioned above focus mainly on dynamic properties. They can and have been used in the past to compare algorithms and make good estimations on the real world behavior. But dynamic properties do highly rely on the quality of the model and tools used to evaluate them.

Structural properties have to be evaluated by using external tools for graph analysis. There are several examples like Pajek [10], Leda [11] or the Boost Graph Library [12] that focus on structural analysis. Their disadvantage is that they can only analyse already exisiting network snapshots. The Network Workbench [13] tries to overcome this issue but it focuses mainly on complex networks in biological, social and physical sciences and offers only a limited set of properties.

Structural properties can help to give a deeper understanding of how the underlying complex network works. For example, comparing shortest path lengths to real routing paths can give insight of how good the routing on top of the given network topology is. Analysis of graph snapshots can therefore give bounds on the possible performance of any application build on top of such a graph. Other upcoming metrics like motifs [14] and role-to-role connectivity [15] give an even deeper understanding of the local structure and responsibilities of a single node.

GTNA is the first approach that combines simulating technological networks with an indepth structural analysis into one toolkit.

3. OUR APPROACH

Due to the varietey of network simulators with different input and output formats it is crucial for an analysis approach to support both, a flexible plugin mechanism for input data (e.g. a network topology from another simulation or emulation) and an extendable metrics analysis mechanism. In the next section the features and design considerations for handling communication networks, series of snapshots, metrics and plotting are presented. The following section shows the networks already implemented, followed by the implemented metrics.

3.1. Modules

Our analysis approach called GTNA is divided into four interchangeable modules with distinct tasks. The network module provides a network topology generation mechanism. The metrics module provides a set of already implemented metrics and an interface for new metrics which need access to the network structure. The series module allows the combination of a set of simulation runs as well as import mechanisms to read already available traces or simulation data. Aggregated data, like the average, variance and the 95% confidence interval are calculated automatically. The plotting module wraps the freely available plotting software gnuplot [16], knowledge of the gnuplot script syntax is not necessary.

3.1.1. Network

A new communication network is created by implementing the interface *gtna.networks.Network*. A set of parameters for the new network may be set according to it's needs. E.g. the content addressable network (CAN) [17] is already implemented. The important attributes for CAN are the number of nodes, the dimensions and the number of realities. Our approach is capable of comparing results for any combination of the chosen input parameters. E.g. a scalability evaluation is performed by simply scheduling several runs of the same network with growing network size. An analsysis of the effects of multiple realities and dimensions in CAN is performed in the same way. Of course, other network approaches require a completely different set of parameters, nevertheless, with this tool, cross comparisons for different network configurations of the same network can be performed.

3.1.2. Series

A series provides aggregation and summaries for several simulation runs. All kind of communication network topologies can be arranged in a series. Averages and confidence intervals for all metrics are automatically calculated. A series is created with the wrapper class *gtna.data.Series*. Input data for a series can be provided in two ways, either by importing available traces, emulations and simulations or by creating the network topology within GTNA.

3.1.3. Metrics

A basic set of metrics is already developed for GTNA. In order to extend the available set of metrics the interface *gtna-metrics.Metric* is provided. All basic operations are implemented by the abstract class *gtna.metrics.MetricImpl*.

The provided metrics can be divided into single-scalar and multi-scalar metrics. The single-scalars are wrapped by *gtna.data.Value* and stored in an instance of *gtna.data.Singles*. They calculate one value for each graph. Well-known examples are characteristic path length and clustering coefficient. Important multi-scalars are shortest path length distribution and local clustering coefficient.

3.1.4. Plotting

One of the well-known tools for plotting is the freely available gnuplot software [16]. It is under active development since 1986 and capable of plotting almost any graph required for the network community. Nevertheless, gnuplot requires a steep learning curve. It is command-line driven and hard to learn for novice users. GTNA uses gnuplot for plot creation. It is not required to know any gnuplot syntax for plotting a graph with GTNA. All commands needed to run gnuplot are shielded in the class *gtna.plot.GNUPlot*. All results can be freely combined and plotted in one or several plots. If required, even multiple metrics for multiple series can be condensed in one plot.

The data is plotted using the class *gtna.plot.Plot*. To plot single-scalar values, a coherent graph requires several instances of a distinct network. One typical example would be the development of the characteristic path length with increasing network size.

3.2. Implemented Networks

GTNA provides a set of reference networks that are located in the package *gtna.networks.canonical*. These network topologies, namely ring, star and fully connected, are helpful when implementing new metrics because their properties are very clear and metrics easily computable by hand. The package *gtna.networks.model* contains implementations for many well known network models such as the random network model by Erdös and Rényi [18] and Kleinberg's Small-World model [19]. Table 1 shows the implemented network models and their configuration parameters.

Table 1. Implemented network models

Name	Parameters
BarabasiAlbert	edges per node
DeBruijn	base, dimensions
ErdosRenyi	average degree, direction
Gilbert	edges, direction
GNC	direction, edge back
GNR	redirection prob., direction
Kleinberg	dimensions, alpha
UnitDisc	square size, radius
WattsStrogatz	successors, beta

Table 2 shows the better known P2P networks, whose topology generation is provided by GTNA. The framework also includes methods to read graphs from files supporting a number of different formats. This includes the Geography Markup Language (GML) [20] format as well as the internet mappings provided by the Cooperative Association for Internet Data Analysis (CAIDA) [21].

Table 2. Implemented P2P networks

Name	Parameters
CAN	dimensions, realities
Chord	bits per ID, successors
Gnutella04	-
Gnutella06	-
Kademlia	bits per ID, bucket size, alpha, lookups
ODRI	base, dimensions, walk length
Pastry	bits per ID, base
PathFinder	virtual peers, neighbors, direction
Symphony	long links, successors, retries, direction

3.3. Implemented Metrics

Metrics can easily be implemented with the provided interface. Table 3 shows the currently implemented metrics. The column submetric shows the implemented related metrics. They are calculated by slightly adapting the basic metric algorithm, by altering its input data or filtering the output values.

4. SOFTWARE ARCHITECTURE

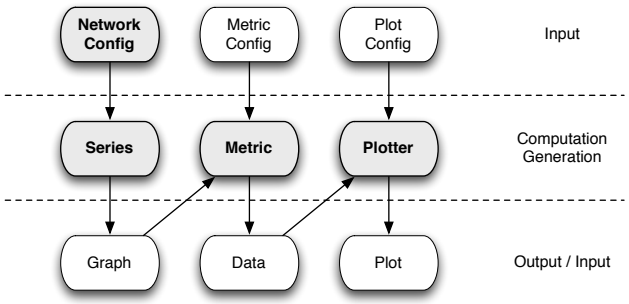

Figure 1. Workflow of GTNA

Figure 1 shows the workflow of GTNA. Each of the four modules is used to produce the desired output that serves as input for another one. We describe this workflow by using two CAN networks with different configurations and a Chord network as an example.

First, a network configuration is created using a custom set of parameters depending on the type of network and its desired configuration (*listing 1*).

Table 3. Implemented Metrics

Name	Submetrics
Average Neighbor Degree	average neighbor (in / out) degree by (in / out) degree
Clustering Coefficient	local clustering coefficient, weighted clustering coefficient, clustering coefficient
Degree Distribution	(int / out) degree (complementary) distribution, average / max / min (in / out) degree
Network Fragmentation	development of average isolated cluster size / number of clusters / max cluster size, average / max number of clusters, point of rupture [node removal by random or (in / out) degree] [computations uni- or bidirectional]
Routing Length	routing length (complementary) distribution, local characteristic routing length, max routing length, characteristic routing length
Shortest Path Length	shortest path length distribution, expansion, local characteristic path length, diameter, characteristic path length, connectivity
Rich Club Connectivity	rich club connectivity

Listing 1. Creating a network configuration

```
Network can1 = new CAN(500, 2, 1);
Network can2 = new CAN(500, 3, 1);
Network chord = new Chord(500, 32, 16);
```

Every network class must implement the interface *gtna.networks.Network* in order to provide all methods that are needed for further computations. The basic methods are already implemented by the abstract class *gtna.networks.NetworkImpl*. Network size, output destination and network name are some of them which are mostly needed for output generation. The most important method of every network implementation provides the generation of a network connectivity graph. It creates an object of the type *gtna.graph.Graph* that contains the underlying network graph. Every network component is represented by a node of type *gtna.graph.Node*. An edge exists between two nodes if they are connected in the network topology. The structure of this graph depends on the specific design of every network and is influenced by the configuration parameters given by the network configuration.

A series (*gtna.data.Series*) contains a given number of graphs generated from one network configuration (*listing 2*). These graphs share certain properties, such as the number of nodes, contain roughly the same number of edges and possess a similar degree distribution.

Listing 2. Generating series from network configuration

```
Series can1Series = Series.generate(can1, 10);
Series can2Series = Series.generate(can2, 10);
Series chordSeries = Series.generate(chord, 10);
```

This concept allows not only for the generation and comparison of different network types. It also enables us to observe the impact of changes in the network configuration on the network's structure and key properties. While a series holds information about the generated graphs, it is also used to aggregate data. It provides average values and confidence intervals for the data derived by applying metrics to the graphs. These values are generated by *gtna.data.AverageData* and *gtna.data.ConfidenceData*. They are used in the last step of the workflow to generate plots for a given series or a set of multiple series.

In the next step, the implemented metrics generate the data for every graph contained in the series. Every metric must implement the interface *gtna.metrics.Metric* and may extend the abstract class *gtna.metrics.MetricImpl* which provides basic methods. These methods are called during series creation which generates and writes the data from all metrics (in folder ***/data/*) as well as the single-scalar results (***/singles.txt*) and a readable version of the whole graph (***/graph.txt*).

Listing 3. Plotting data

```
Series[] allSeries = new
   Series[] {can1Series, can2Series, chordSeries};
Plot.multiAvg(allSeries, "./plots/multi-avg/");
Plot.multiConf(allSeries, "./plots/multi-conf");
```

The average data and confidence intervals as well as the single-scalar values are then used as input for the various plots. The plotting functionality is provided by *gtna.plot.Plot* (*listing 3*) and uses *gtna.plot.GNUPlot* as an interface to gnuplot.

Listing 4. Plotting single-scalar values

```
Series[] s1 = Series.generate(new Network[]{
  new CAN(500, 2, 1), new CAN(1000, 2, 1),
  new CAN(1500, 2, 1), new CAN(2000, 2, 1)}, 10);
Series[] s2 = Series.generate(new Network[]{
  new CAN(500, 3, 1), new CAN(1000, 3, 1),
  new CAN(1500, 3, 1), new CAN(2000, 3, 1)}, 10);
Series[][] s = new Series[][]{ s1, s2 };
Plot.singlesAvg(s, "./plots/singles-avg/");
Plot.singlesConf(s, "./plots/singles-conf/");
```

Plotting single-scalar values is not very useful since the plots would only contain single points. It is therefore supported to plot the development of the single-scalar values during the change of one parameter, the most common and important one being the network size. This allows for the comparison of attributes as a network grows and is crucial for

the analysis of network scalability. Multiple sets of different networks can easily be compared as the example shows (*listing 4*).

The class hierarchy of GTNA is mostly structured by the different modules and is shown in table 4.

Table 4. Packages of GTNA

Package	Description
gtna.data	series, data generators
gtna.graph	graph, node, edge
gtna.io	input and output
gtna.metrics	implemented metrics, interface
gtna.networks	implemented networks, interface
gtna.plot	plotter, gnuplot interface
gtna.util	utilities

5. CONFIGURATION AND EXTENSION

Since the network interface requires the implementation of several methods, the abstract class *gtna.networks.NetworkImpl* implements most of them and can be used by extending it. Its constructor only has four parameters: a network-specific key for the identification of the metric, the desired network size, an array containing the parameter keys and an array with the specific configuration parameters. The only method that needs to be implemented for every single network is the generation of a specific network graph using the unique network design and the given parameters. The following example implements the network *NoEdges* (*listing 5*). It requires the examplary parameter *p1*. The Graph is generated without assigning any edges to the nodes.

Listing 5. Creating a new network

```
public class NoEdges extends NetworkImpl
      implements Network {
  private int p1;
  public NoEdges(int nodes, int p1){
    super("NE", nodes, new String[]{"P1"},
      new String[]{p1 + ""});
    this.p1 = p1;
  }
  public Graph generate(){
    Node[] nodes = Node.init(this.nodes());
    return new Graph(this.description(), nodes);
  }
}
```

The unique key *NE* is used to obtain certain values from the configuration file such as the network's name and the output folder (*listing 6*).

Listing 6. Basic network configuration

```
NE_NAME = No Edges
NE_FOLDER = noEdges
NE_P1_NAME = Parameter 1
```

All contents of the configuration file can be accessed using the class *gtna.util.Config*. By default, the configuration file is read from *./conf.properties* but can be read from another source at any time using the static method *Config.init(String filename)*.

Listing 7. Creating a new metric

```
public class NodesOfDegree extends MetricImpl
      implements Metric {
  private int[] nod;
  private int domn;
  public NodesOfDegree(){
    super("NOD");
  }
  public void computeData(Graph g) {
    // compute nod and domn
  }
  public void writeData(String folder) {
    DataWriter.write("NOD_NOD",folder,this.nod);
  }
  public Value[] getValues() {
    Value domn = new Value("NOD_DOMN",this.domn);
    return new Value[]{ domn };
  }
}
```

Creating a new metric and including it in the framework also requires the implementation of an interface: *gtna.metrics.Metric*. As in the case of networks, there exists an abstract class that implements most of these methods in a standardized way: *gtna.metrics.MetricImpl*. The only method besides writing data and return single-scalar values that needs to be implemented is the computation of the metric-specific data. This is shown by the implementation of a simple metric called *NodesOfDegree* (*listing 7*). It computes the simple metric *Nodes of Degree* (NOD) that counts the number of nodes for every degree as well as the single-scalar metric *Degree of Most Nodes* (DOMN) which is the degree of the largest group of nodes.

Listing 8. Configuring a metric

```
NOD_CLASS = NodesOfDegree
NOD_NAME = Nodes of Degree
NOD_DATA_KEYS = NOD_NOD
NOD_SINGLES_KEYS = NOD_DOMN
NOD_DATA_PLOTS = NOD_NOD
NOD_SINGLES_PLOTS = NOD_DOMN

NOD_NOD_DATA_NAME = Nodes of Degree
NOD_NOD_DATA_FILENAME = nod

NOD_DOMN_SINGLE_NAME = Degree of Most Nodes
```

To allow the super-class *MetricImpl* as well as the *Series* to access the new metric, it needs to be configured properly (*listing 8*). Single- and multi-scalar metrics also need to be included in the configuration file.

Listing 9. Including a metric in the workflow

```
## METRICS = CC, DD, RCC, RL, SPL
METRICS = CC, DD, RCC, RL, SPL, NOD
```

To include an implemented and configured metric into the overall workflow, it needs to be added to the set of metrics.

Again, the unique key is used and simply added to the entry *METRICS* in the configuration file (*listing 9*).

Every plot requires a number of field such as the filename for the plot, title and names for x- and y-axis. The unique keys assigned to each metric result are used to configure the plot of metric data or their combination. The field *DATA* contains a list of all data keys that should be contained in the plot. If only a single metric is to be plotted, this list contains only this one key as in the case of the plot of the nodes of degree metric (*listing 10*).

Listing 10. Configuring a plot

```
1 NOD_NOD_PLOT_DATA = NOD_NOD
2 NOD_NOD_PLOT_FILENAME = nod
3 NOD_NOD_PLOT_TITLE = Nodes of Degree
4 NOD_NOD_PLOT_X = Degree d
5 NOD_NOD_PLOT_Y = #(nodes(d))
```

To combine multiple metric data in one plot they all need to be added to the data list. The following example shows the configuration for a single-scalar plot that combines the degree of most nodes metric with the maximum and minimum node degree (*listing 11*). The latter ones are computed by the degree distribution metric.

Listing 11. Combining multiple metrics in one plot

```
1 DOMN_MIN_MAX_PLOT_DATA = NOD_DOMN, DD_MIN, DD_MAX
2 DOMN_MIN_MAX_PLOT_FILENAME = domn-min-max
3 DOMN_MIN_MAX_PLOT_TITLE = DOMN / D-min / D-max
4 DOMN_MIN_MAX_PLOT_Y = domn / d-min / d-max
```

6. EXAMPLE

The most common use of our framework so far has been the analysis of single networks as well as their comparison to other networks, common network models and different configurations of the same network. Figure 2 shows the development of the shortest path length distribution of a CAN network when increasing the number of dimensions. Since routing in a network does not always follow the shortest path possible, the routing length of the CAN routing algorithm is also plotted.

Single-scalar values may be as a function of the network size. Thereby they illustrate the network properties under network growth. As an example, the development of the diameter of different CAN networks is shown in figure 3.

7. EVALUATION

In section 5. we have shown how to implement new metrics in GTNA. Evaluations for complex networks like current P2P networks, have to be performed with network sizes of several thousand nodes. E.g. the P2P network Tapestry [22] was simulated with 4096 nodes, Chord [23] was evaluated with 10.000 simulated nodes, Symphony [24] was evaluated with up to 16.384 nodes. Newer complex networks might even require more nodes for a significant analysis. In order to provide a network tool for daily usage, one requirement is to

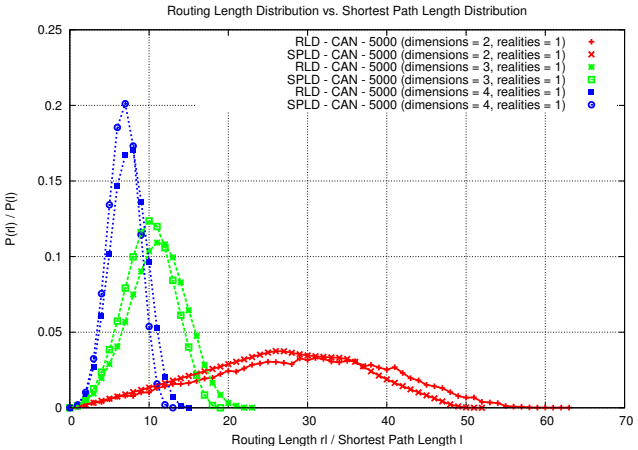

Figure 2. CAN: routing length vs. shortest path length

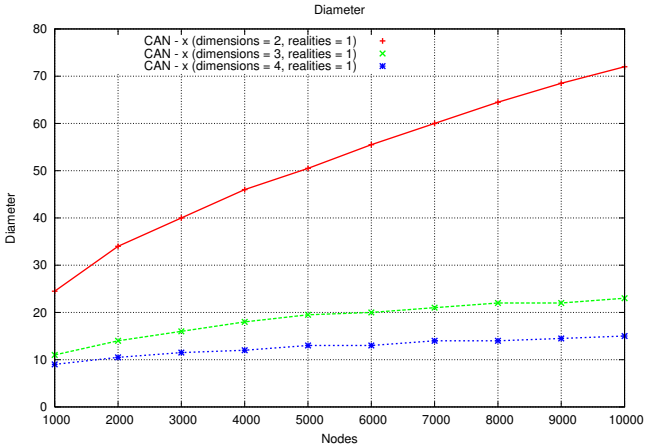

Figure 3. CAN: development of the diameter

have GTNA running on regular desktop computers. For evaluation we have chosen a 2.4 GHz Intel Core 2 Duo processor running MAC OS 10.5.8 with 2 GB of memory. Besides the time needed for calculating the desired metrics, the required amount of memory limits the scalability of GTNA. Our goal was to evaluate network topologies with more than 20.000 nodes, requiring less than 2 GB of memory. We have used an ErdösRényi random graph with an average degree of 20, 30, 40 and 50 for evaluation. Currently GTNA is implemented in JAVA, we used version 1.5.0_22 for all experiments.

We have chosen the shortest-path-length (SPL) metric for evaluation, since it is the most CPU consuming metric currently implemented and therefore determining the runtime of the whole analysis. Figure 4 shows the runtime in seconds needed for network sizes between 1.000 and 20.000 nodes calculating the SPL metric. It does highly depend on the number of nodes, but even a network of 20.000 nodes with average degree of 50 is analyzed in about 3 minutes.

Figure 5 shows the required memory for the conducted

Table 5. Runtime for ErdösRényi with average degree 50

Metric	1.000	2.500	5.000	7.500	10.000	12.500	15.000	17.500	20.000
Average Neighbor Degree	0.500	0.300	0.500	0.900	3.500	3.100	3.600	4.100	4.600
Clustering Coefficient	0.356	0.972	1.991	3.139	4.322	5.785	7.158	8.074	9.542
Degree Distribution	0.000	0.001	0.001	0.001	0.001	0.002	0.002	0.003	0.003
Network Fragmentation (U)	0.173	0.489	1.078	1.896	3.060	4.015	5.207	6.867	7.931
Network Fragmentation (B)	0.154	0.361	0.738	1.151	1.690	2.067	2.569	3.023	3.628
Rich Club Connectivity	0.115	0.722	2.997	6.931	12.868	21.816	30.205	43.285	56.119
Shortest Path Length	0.307	2.126	9.511	22.543	42.311	69.591	102.951	146.539	190.755
Memory Usage in MB	7.491	16.704	31.91	47.041	34.228	42.552	46.281	63.093	66.57

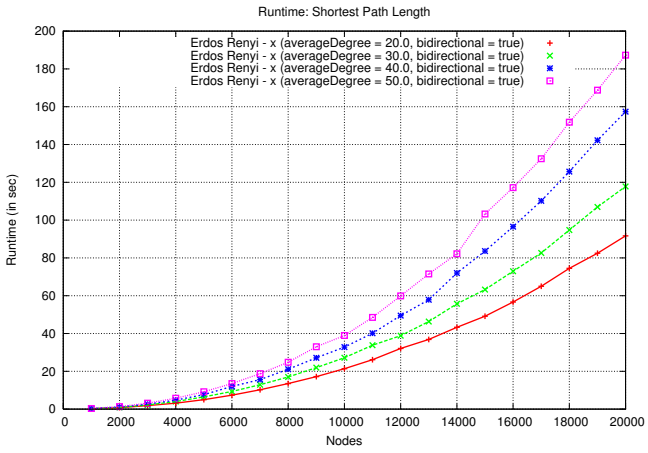

Figure 4. Runtime of the shortest path length metric (in sec)

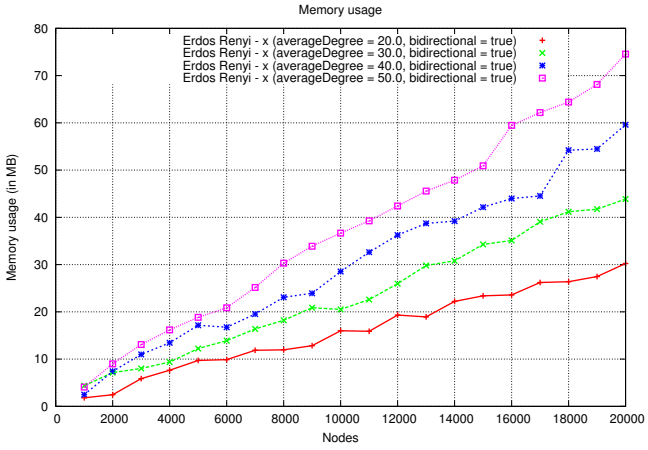

Figure 5. Memory usage (in MB)

SPL experiment. Network with 20.000 nodes and more can easily be analyzed with a regular desktop PC. The measurements of the runtime behavior and the memory usage clearly show the usability of GTNA on regular desktop PCs. Table 5 shows the required CPU time for each metric and the used memory. Given the 2.4 GHz processor with 2 GB of memory, we performed a scalability test with all metrics currently implemented. The limiting factor was not the CPU, but the required memory. The available 2 GB were completely consumed by a simulation with about 600.000 nodes which needed 1.9 GB of RAM.

8. SUMMARY AND OUTLOOK

Graph-theoretic analysis of large networks greatly aids the understanding of complex properties. Regional functions and local behavior, which have a significant impact on the overall performance of the networks, in consequence can be identified. Analysing complex networks so far has been the cumbersome task of adapting specific algorithms for the purpose of measuring certain single metrics in especially formatted network graphs, and in consequence has only been viable for specialists.

This paper introduces GTNA, the Graph-Theoretic Network Analyzer. GTNA is an efficient, Java-based toolkit for the concise analysis of complex network graphs. It already includes graph generators for selected models, like the ErdösRényi and Gilbert Random Graphs, the Barabasi-Albert and Kleinberg scale free graphs, and a multitude of further graphs that are commonly used for comparison. Modules to create parametrized, realistic topologies of all predominant Peer-to-Peer approaches, like Chord, CAN, Pastry, Kademlia, and Gnutella, are already provided. GTNA additionally implements all major metrics used for the analysis of complex graphs in sociology, biology, and computer science. These include all conventional networking metrics (e.g., diameter, average path length, degree distribution) as well as the common robustness- and resilience metrics (fragmentation, consecutive node removal by highest degree or random choice), and more recent metrics for the analysis of complex networks, such as the Rich-Club-Connectivity. The toolkit allows for the generation and analysis of single graphs or batch-generation analyses of multiple graphs. All results are calculated with their confidence intervals using a normal probability distribution. GTNA is easily extensible offering a simple plugin architecture with well defined interfaces and a comprehensive, self-explaining parametrization using configuration files.

GTNA already contains modules for the main graph generator models and the common graph metrics. However, it is still in the process of being extended, and to this end we are currently implementing modules for the measurement of network motifs [14] and the role-to-role connectivity profiles [15]. Additionally, we are currently providing generators for further network models, which include the PFP [25] and PARG [26] models for the generation of realistic Internet topologies, among others. Analysis of network dynamics is another problem we are currently aiming at. Understanding the topological changes throughout the life time of systems can indicate special properties and may give deeper insight into underlying principles of their design and function. We are hence working towards including modules to trace and analyse changing topologies with respect to arbitrary metrics into GTNA.

REFERENCES

[1] S. Kurkowski *et al.*, "Manet simulation studies: the incredibles," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 4, p. 61, 2005.

[2] M. Jelasity *et al.*, "The Peersim Simulator," (last accessed: February 23, 2010) http://peersim.sf.net.

[3] A. Kovacevic *et al.*, "Benchmarking Platform for Peer-to-Peer Systems," *Information Technology*, vol. 49, no. 5, pp. 312–319, 2007.

[4] P. García *et al.*, "Planetsim: A new overlay network simulation framework," *Software Engineering and Middleware, SEM 2004, Linz, Austria*, pp. 123–137, 2005.

[5] R. Barr *et al.*, "JiST: An efficient approach to simulation using virtual machines," *Software Practice and Experience*, vol. 35, no. 6, pp. 539–576, 2005.

[6] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference*, 2001, pp. 319–324.

[7] S. Mccanne, S. Floyd, and K. Fall, "ns2 (network simulator 2)," (last accessed: February 23, 2010) http://www-nrg.ee.lbl.gov/ns/.

[8] "The ns-3 network simulator," (last accessed: February 23, 2010) http://www.nsnam.org.

[9] R. Bagrodia *et al.*, "Parsec: A parallel simulation environment for complex systems," *Computer*, pp. 77–85, 1998.

[10] V. Batagelj and A. Mrvar, "Pajek - Analysis and Visualization of Large Networks," in *Graph Drawing*, vol. 21, no. 2. Springer Verlag, 2002, p. 477.

[11] K. Mehlhorn and C. Uhrig, "LEDA: a platform for combinatorial and geometric computing," *Communications of the ACM*, vol. 38, no. 1, pp. 96–102, 1995.

[12] J. Siek *et al.*, "The Boost Graph Library: User Guide and Reference Manual," in *Proceedings*, vol. 243. Addison-Wesley, 2002, pp. 112–121.

[13] N. Team, "Network Workbench Tool," (last accessed: February 23, 2010) http://nwb.slis.indiana.edu.

[14] R. Milo *et al.*, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, p. 824, 2002.

[15] R. Guimerà *et al.*, "Classes of complex networks defined by role-to-role connectivity profiles," *Nature physics*, vol. 3, no. 1, p. 63, 2007.

[16] T. Williams *et al.*, "GNUplot: an interactive plotting program," 1993.

[17] S. Ratnasamy *et al.*, "A scalable content-addressable network," in *SIGCOMM*. ACM, 2001, p. 172.

[18] P. Erdös and A. Rényi, "On the evolution of random graphs," *Publication*, 1960.

[19] J. Kleinberg, "The small-world phenomenon: an algorithm perspective," in *ACM symposium on Theory of computing*. ACM, 2000, pp. 163–170.

[20] D. Burggraf, "Geography markup language," *Data Science Journal*, vol. 5, no. 0, pp. 178–204, 2006.

[21] The Cooperative Association for Internet Data Analysis, "Ipv4 routed /24 as links dataset," 2007, (last accessed: February 23, 2010) http://www.caida.org/data/active/ipv4_routed_topology_aslinks_dataset.xml.

[22] B. Zhao *et al.*, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," *IEEE Journal*, vol. 74, pp. 11–20, 2001.

[23] I. Stoica *et al.*, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*. ACM, 2001, p. 160.

[24] G. Manku *et al.*, "Symphony: Distributed Hashing in a Small World," in *USENIX*. Stanford InfoLab, 2003.

[25] S. Zhou, "Characterising and modelling the internet topology - The rich-club phenomenon and the PFP model," *BT Technology Journal*, vol. 24, no. 3, pp. 108–115, 2006.

[26] M. Piraveenan *et al.*, "Local assortativity and growth of Internet," 2009.