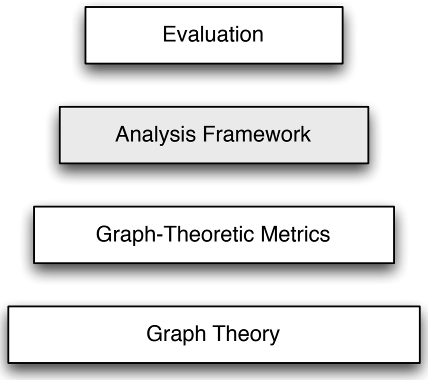
4 Data Acquisition - GTNA

For the purpose of generating and analyzing topology snapshots for well-known network models we created the Graph-Theoretic Network Analyzer (GTNA), an efficient Java-based toolkit for the comprehensive analysis and generation 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 this section we present the design and simple extensibility of GTNA, as well as the network models and metrics that are already implemented. We also describe the different plot types currently supported by GTNA and show examples of its scalability and performance.



GTNA
Graph-Theoretic Network Analyzer

4.1 Framework Components

Due to the variety 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 this subsection 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.
The GTNA framework is divided into four interchangeable modules with distinct tasks. The network module provides a generation mechanism for network topologies. The metrics module offers a set of already implemented metrics and an interface for the creation of new metrics. 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 values and the 95% confidence intervals are calculated automatically. The plotting module wraps the freely available plotting software gnuplot [33].

Network

A new network generator can be created by a class that implements the provided interface *gtna.networks.Network*. A set of parameters for the new network may be set according to it's needs. For example, the content addressable network (CAN) [25] is already implemented. The important attributes for a 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 configuration parameters. E.g. a scalability evaluation is performed by simply scheduling several runs of the same network with growing network size. An analysis 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 topologies and configurations of the same network can be performed.

Series

A series provides aggregation and summaries for several simulation runs of the same network configuration. Averages and confidence intervals for all metrics are calculated automatically. 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 using an implemented network generator.

Metrics

A basic set of metrics has been developed for GTNA and can be found in the package *gtna.metrics*. In order to extend the available metrics the interface *gtna.metrics.Metric* is provided. All basic operations needed for the implementation of this interface are already 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 the characteristic path length and the clustering coefficient. Important multi-scalar metrics are the shortest path length distribution and the local clustering coefficient which compute an array of values for every input graph.

Plotting

One of the well-known tools for plotting is the freely available gnuplot software [33]. It is under active development since 1986 and capable of plotting almost any graph required for the presentation of various data sets. Nevertheless, gnuplot requires a steep learning curve. It is command-line driven and hard to learn for novice users. It is not required to know any gnuplot syntax for plotting a graph with GTNA. All commands needed to run the software 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 a single file.
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.

Implemented Networks

GTNA provides a set of reference network implementations 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 [7] and Kleinberg's Small-World model [16]. Table 4.1 shows the implemented network models and their configuration parameters.

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 4.1: Network models implemented in GTNA

Table 4.2 shows the better known P2P networks, whose topology generation is provided by GTNA.

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

Table 4.2: P2P networks implemented in GTNA

The framework also includes methods to read graphs from files supporting a number of different formats. This includes the Graph Modeling Language (GML) [13] format as well as the internet mappings provided by the Cooperative Association for Internet Data Analysis (CAIDA) [31].

Implemented Metrics

Metrics can easily be implemented with the provided interface. Currently, all metrics that we described in 3 are implemented in GTNA as listed in table 3.6. This includes the different metrics concerning the vertex degrees, the shortest path and routing length metrics as well as network fragmentation and clustering coefficient computations.

4.2 Software Architecture

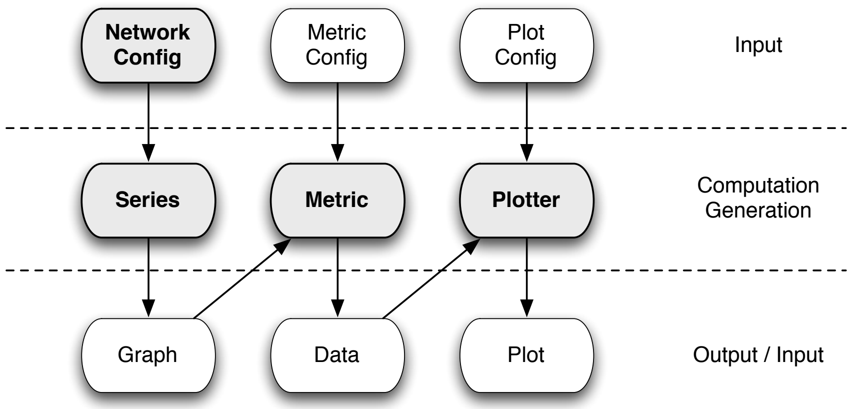


Figure 4.1: Workflow of GTNA

The workflow of GTNA is shown in figure 4.1. Each of the four modules described above is used to produce the desired output that serves as input for another one. We describe this workflow by using two CAN network configurations with different parameters and a Chord network configuration as an example.

Network Configuration

First, a network configuration is created using a custom set of parameters that depends on the type of network and its desired configuration. In listing 4.1 two different instances of CAN network configurations are created. Both contain 500 nodes and use 1 reality. They differ in the number of dimensions used to span the identifier space. The instance of the Chord network configuration also uses 500 nodes but requires three different parameters: the number of bits per identifier, the number of successor links in the identifier space and a distance parameter.

```
Network can1 = new CAN(500, 2, 1);
Network can2 = new CAN(500, 3, 1);
Network chord = new Chord(500, 32, 5, 3);
```

Listing 4.1: GTNA - creating instances of network configurations

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 in the abstract class *gtna.networks.NetworkImpl* that can easily be extended. The most important part of every network implementation is the method *generate()* that returns a network connectivity graph for the given network topology configuration. This graph is stored in an object of the type *gtna.graph.Graph*. Every vertex in the graph is represented by a node object 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 respective configuration parameters given by the network configuration. The generated graph represent possible snapshots of the considered network topology.

Series

A series of the type *gtna.data.Series* contains a certain number of graphs generated using the same network configuration. These graphs share certain properties like the number of nodes, contain roughly the same number of edges and possess a similar degree distribution since they are generated using the same topology model and network-specific configuration. The example in listing 4.2 shows the generation of a series for every network configuration that we instantiated in listing 4.1.

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

Listing 4.2: GTNA - generating series from network configurations

The second parameter in the static method *Series.generate* states that 10 graphs should be created for every series using the exact same given network configuration. The slight differences between the graphs originate from the pseudorandom selection of identifiers and neighbors in accordance with the rules of the overlay generation algorithm and the respective configuration parameters.
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 encapsulated 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 multiple sets of series.

Metric Computation

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 the generation of a new series for every graph. The results from these computations for graph *i* are then written in the folder ***/i/data/* where **** stands for the series-specific path that depends on the encapsulated network configuration. The results from single-scalar metrics are combined and written in the file ***/i/singles.txt* and a readable version of the whole graph is stored under ***/i/graph.txt*. The aggregated values for every series are stored in ***/avg/* and ***/conf/* and the averages of all single-scalar metrics are written in the file ***/average.txt*. Listing 4.3 shows the folder hierarchy of the first series created in example 4.2. In this case **** is replaces by *500/can-2-1*, the network configuration specific output folder of the network configuration *CAN(500, 2, 1)*.

```
500/
    can-2-1/
            0/
              data/
                   dd.txt
                   ddi.txt
                   ddo.txt
                   ...
              graph.txt
              singles.txt
            1/ ...
            2/ ...
            ...
            avg/
                dd.txt
                ddi.txt
                ddo.txt
                ...
            conf/
                 dd.txt
                 ddi.txt
                 ddo.txt
                 ...
            average.txt
            output.txt
```

Listing 4.3: GTNA - file hierarchy and files generated by *Series.generate(CAN(500, 2, 1), 10)*

Plotting

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* and uses *gtna.plot.GNUPlot* as an interface to gnuplot (cf. listing 4.4).

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

Listing 4.4: GTNA - plotting data

Plotting single-scalar values is not very useful since the plots would only contain a single point for every series. It is therefore supported to plot the development of the single-scalar values during the variation 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 in listing 4.5 demonstrates.

```
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/");
```

Listing 4.5: GTNA - plotting single-scalar values

Class Hierarchy

The class hierarchy of GTNA is mostly structured by the different modules and is listed in table 4.3.

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

Table 4.3: Packages of the GTNA implementation

4.3 Configuration and Extension

In this subsection we give a brief overview how GTNA can be extended by creating new metrics and network configurations. We also show how to properly configure them to allow for a seamless integration in the existing workflow.

Network configurations

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 topology graph by applying the respective network design while considering the given parameters. The following example in listing 4.6 implements the network *OneEdge* that contains the given number of nodes and creates only one edge between two specified nodes. It requires the standard parameter *nodes* and the indices of the two nodes that should be connected by an edge.

```
public class OneEdge extends NetworkImpl implements Network {
  private int n1;
  private int n2;

  public NoEdges(int nodes, int n1, int n2){
    super("OE", nodes,
          new String[]{"N1", "N2"},
          new String[]{n1 + "", n2 + ""});
    this.n1 = n1;
    this.n2 = n2;
  }

  public Graph generate(){
    Node[] nodes = Node.init(this.nodes());
    if(n1 != n2 && this.n1 < this.nodes() && n2 < this.nodes()) {
      Edges edges = new Edges(nodes, 1);
      Edge edge = new Edge(nodes[this.n1], nodes[this.n2]);
      edges.add(edge);
      edges.fill();
    }
    return new Graph(this.description(), nodes);
  }
}
```

Listing 4.6: GTNA - implementation of a new network generator

The unique key *OE* is used to obtain certain values from the configuration file such as the network's name, its output folder and names for every parameter. The associated part of a configuration file is depicted in listing 4.7.

```
OE_NAME = One Edge
OE_FOLDER = oneEdge
OE_N1_NAME = Source Node
OE_N2_NAME = Destination Node
```

Listing 4.7: GTNA - configuration of a network generator

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)*. During runtime, single configuration items can be overwritten using the method *Config.overwrite(String key, String value)*. To reset them, the method *Config.reset(String key)* is provided.

Metrics

Creating a new metric and including it in the framework requires the implementation of the interface *gtna.metrics.Metric* as well as the addition of certain parameters to the configuration file. Analog to the network configurations, abstract class *gtna.metrics.MetricImpl* implements most of these methods. The only method besides writing the multi-scalar data into files and returning an array of single-scalar values that needs to be implemented for every metric is the computation of the metric-specific data for a given graph. An example is given by the implementation of the simple metric called *NodesOfDegree* in listing 4.8. 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 defined asa the degree of the largest group of nodes.

```
public class NodesOfDegree extends MetricImpl implements Metric {
  private int[] nod;
  private int domn;

  public NodesOfDegree(){
    super("NOD");
  }

  public void computeData(Graph g) {
    // compute nodesOfDegree metrics
    this.nod = new int[g.maxDegree + 1];
    for(int i=0; i<g.nodes.length; i++) {
      Node n = g.nodes[i];
      int degree = n.in.length + n.out.length;
      this.nod[degree]++;
    }
    // compute degreeOfMostNodes metric
    int max = this.nod[0];
    this.domn = 0;
    for(int i=0; i<this.nod.length; i++) {
      if(this.nod[i] > max) {
        max = this.nod[i];
        this.domn = i;
      }
    }
  }

  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 };
  }
}
```

Listing 4.8: GTNA - implementation of a new metric

To allow the super-class *MetricImpl* as well as the *Series* instances to access the new metric, it needs to be configured properly as shown in listing 4.9. Besides the class and the metric's name, lists for all single- and multi-scalar metrics and plots as well as names need to be included. In addition, multi-scalar metrics also require a filename.

```
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
```

Listing 4.9: GTNA - configuration of a metric

To include an implemented and properly configured metric into the workflow of GTNA, 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 as depicted in listing 4.10.

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

Listing 4.10: GTNA - including a metric in the workflow

Plots

Every plot requires a number of entries in the configuration file such as filename, title and labels for x- and y-axis. The *DATA* entry contains a list of all metrics that should be contained in the plot. If only a single metric is to be plotted, this list contains only this one key. A configuration for the nodes of degree metric is given in listing 4.11.

```
NOD_NOD_PLOT_DATA = NOD_NOD
NOD_NOD_PLOT_FILENAME = nod
NOD_NOD_PLOT_TITLE = Nodes of Degree
NOD_NOD_PLOT_X = Degree d
NOD_NOD_PLOT_Y = #(nodes(d))
```

Listing 4.11: GTNA - configuration of a plot

To combine multiple metrics in one plot they all need to be added to the data list. The following example in listing 4.12 shows the configuration for a single-scalar plot that combines the degree of most nodes with the maximum and minimum node degree. The latter ones are computed by the degree distribution metric.

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

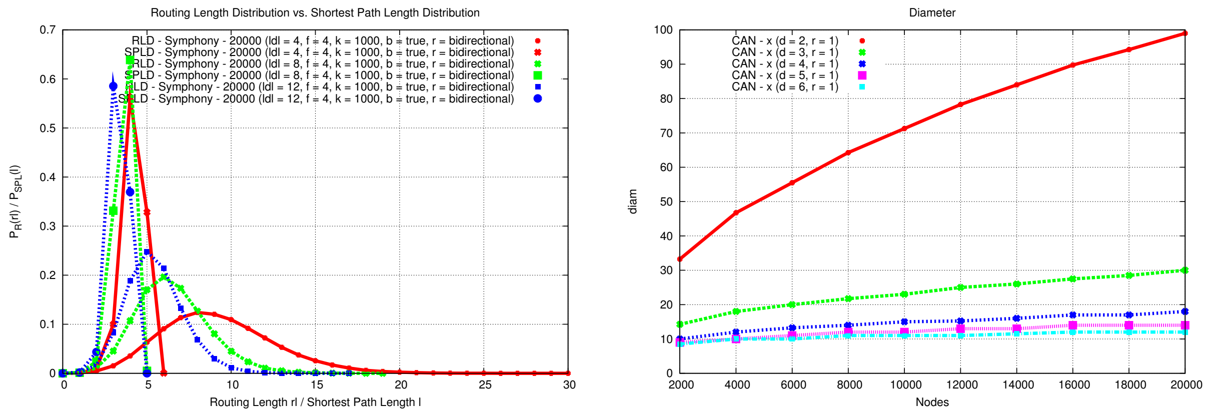
Listing 4.12: GTNA - combination of multiple metrics in one plot

4.4 Plot types

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 differing configurations of the same network. The results from all these different networks that are encapsulated in instances of series can be combined freely to generate plots with the data from all kinds of networks. Also, we can combine multiple metrics in the same plot to compare them and give evidence for possible correlations. We implemented three different types of plots that are useful in different szenarios: plots of multi-scalar metrics, plots of single-scalar metrics and the so-called plots by edges.

Multi-scalar plots

Multi-scalar metrics compute an array of values for every graph. Therefore, we can combine a list of different networks in one plot while the data set of every network is represented by a set of average values or confidence intervals. Figure 4.2(a) shows a plot of the shortest path length combined with the routing path length of different configurations of the Symphony network. These plots allows us to compare the respective properties given by the used multi-scalar metrics. In the presented example, we combined the shortest path length with the routing length distribution to depict the degree of how much the particular configurations utilize the underlying network topology during the routing procedure.



(a) Routing vs. shortest path length distribution (b) Development of the diameter

Figure 4.2: Single- and multi-scalar metrics for different network configurations

Single-scalar plots

Single-scalar metrics return only a single value for every input graph. This allows a more coarse grained comparison between much more different network configurations than the plots of multi-scalar metrics. The values from multiple configurations can therefore be combined in a single set that is represented as a single item in a single-scalar plot. For example, we can change one single parameter of a network configuration and observe its impact on the respective graph metric. In figure 4.2(b) we give an example of such a single-scalar plot. The five sets of CAN network configurations have the same number of realities but differ in the number of dimensions. Every set contains multiple configurations with network sizes between 1,000 and 20,000 nodes. Since the network size is the distinguishing parameter inside of each set, it is plotted on the x-axis. Thereby, the plot shows the development of the diameter of a CAN system during network growth for different configurations.

Plots by edges

The quality of a network topology is determined by the tradeoff between the number of edges per node and the values of examined graph-theoretic properties. Obviously, a network topology A is classified higher in respect to routing efficiency than topology B if they exhibit the same characteristic routing length but A only requires 80% of B's connections to achieve that. The single- and multi-scalar plots presented so far cannot display this ratio as they differentiate by configuration parameters only. Therefore we also implemented a type of plot that distinguishes network configurations by the resulting number of edges in the network graph rather that the configuration parameters. In these plots the number of edges in the respective network configuration is projected on the x-axis as shown by the example in figure 4.3.

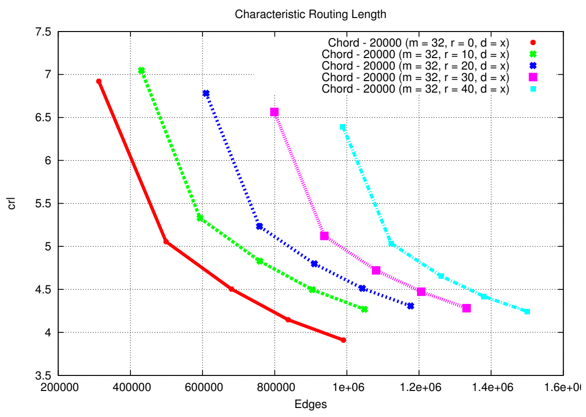
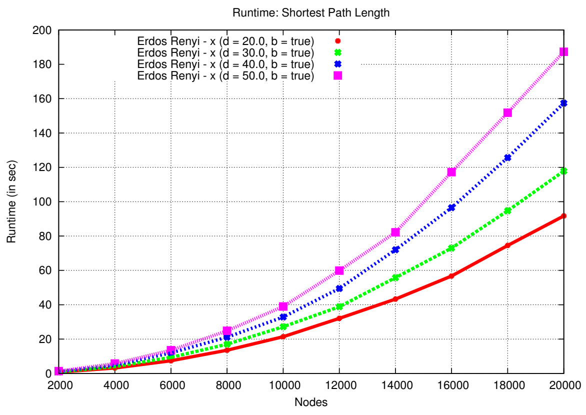


Figure 4.3: Characteristic routing length for different configuration of the Chord network

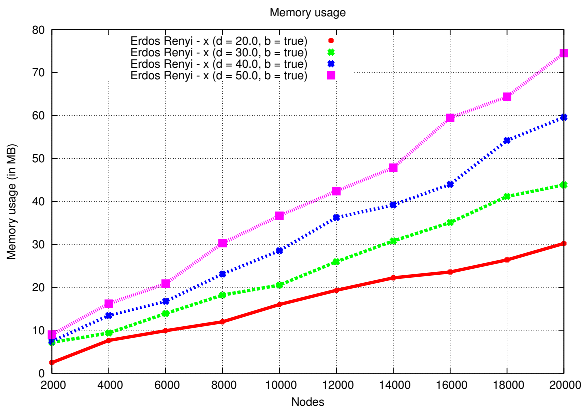
This kind of plot also allows for a comparison of topology sets that don't even share a parameter to differentiate them. In the given example, the characteristic routing length of different network topologies with varying configurations is combined in one plot. This allows for a very precise evaluation of how efficiently different networks and their respective configurations use the additional edges and take advantage of them.

4.5 Runtime and Memory Usage

Graph-theoretic topology analysis for complex networks like current P2P networks have to be performed with network sizes of several thousand nodes. The P2P network Tapestry [34] for example was simulated with 4096 nodes, Chord [29] was evaluated with networks containing 10.000 nodes and the analysis of Symphony [20] was performed 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 have GTNA running on regular desktop computers.
For the evaluation of GTNA we used 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 analyze network topologies with more than 20.000 nodes, requiring less than 2 GB of memory. We generated ErdösRényi random graphs with an average degree of 20, 30, 40 and 50 for the evaluation of GTNA. Currently, GTNA is implemented in Java. We used Java version 1.5.0_22 for all experiments.
We have chosen the shortest path length metric for the runtime evaluation, since it is the most CPU consuming metric currently implemented and therefore determines the runtime of the whole analysis. Figure 4.4(a) shows the runtime in seconds needed for the computation of the shortest path length metrics for network sizes between 1.000 and 20.000. The runtime highly depends on the number of nodes, but even a network of 20.000 nodes with average degree of 50 is analyzed in about 3 minutes. The runtime increases exponentially due to the complexity of the shortest path length computation.
Figure 4.4(b) shows the required memory for the computation of all metrics for the described networks topologies. While the memory consumption growth linearly with the network size, only 75 MB are needed for the analysis of an ErdösRényi random graph with 20.000 nodes an an average degree of 50.



(a) Runtime of the shortest path length metric



(b) Memory usage for all metrics

Figure 4.4: Runtime and memory usage of GTNA for ErdösRényi random graphs

Metric	**1.000**	**2.500**	**5.000**	**7.500**	**10.000**	**12.500**	**15.000**	**17.500**	**20.000**
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

Table 4.4: Runtime of all metrics for ErdösRényi random graph with an average degree 50

The runtime for every metric and the cumulative memory usage of all computations for an ErdösRényi graph of average degree 50 and different sizes is given in table 4.4. These measurements clearly show the usability of GTNA on regular desktop computers. Furthermore, the limiting factor was not the CPU, but the required memory. The available 2 GB were completely consumed by a simulation with 600.000 nodes which required 1,9 GB of RAM.