DNA

DYNAMIC NETWORK ANALYZER

Series Generation
Instruction Manual

René Wilmes

10.05.2016

Dresden University of Technology

Contents

Chapter 1

Introduction

The DNA – Dynamic Network Analyzer is a framework for graph-theoretic Analysis of Dynamic Networks. It offers sophisticated ways to generate and analyze graphs. The built-in Plotting- and LaTeX-output features allow to visualize and compare data by generating LaTeX-documents containing data and plots.

This manual will show how DNA can be used to generate and analyze graphs. It covers the most important aspects of graph generation using DNA including what options and parameters may be utilized in order to customize the outcome and reach certain results. Examples are going to be shown to give the user an insight on the impact of certain customizations.

Chapter 2

Series-Generation in DNA

This chapter will deal with the general series-generation mechanisms in DNA and how to use them. Different approaches and configurations will be shown on examples to give a quick and easy insight into DNA usage.

2.1 General

A series in DNA is composed of several objects:

$$S = (GraphGenerator, BatchGenerator, Metric[], SeriesDir, SeriesName)$$

The generation process uses a GraphGenerator to generate the graph and a BatchGenerator to compile graph changes into batches. Figure 2.1 illustrates the architecture and components of DNA. It offers a variety of different Graph- and BatchGenerators for different graphs and purposes. Due to the nature of dynamic graphs, metrics can either calculate their values based on initial information and additional updates or do a full recompute after each update. During generation DNA will store statistics, runtimes and computed data in series objects. A series contains one or more runs, each of which represent one separate simulation. A run is divided into several batches, which contain statistics, runtimes and metric datas, see 2.2. The *aggr*-directory envelopes aggregated data of all runs contained in the specific series. For more details on the DNA architecture and theoretical background check the DNA paper [1].

2.2 Getting started

The usual workflow is pretty simple. First one initializes a GraphGenerator, BatchGenerator and an array of metrics. Then a Series object is initialized with these inputs. The second step is the generation itself: Each Series-object has several generation-methods for different purposes.

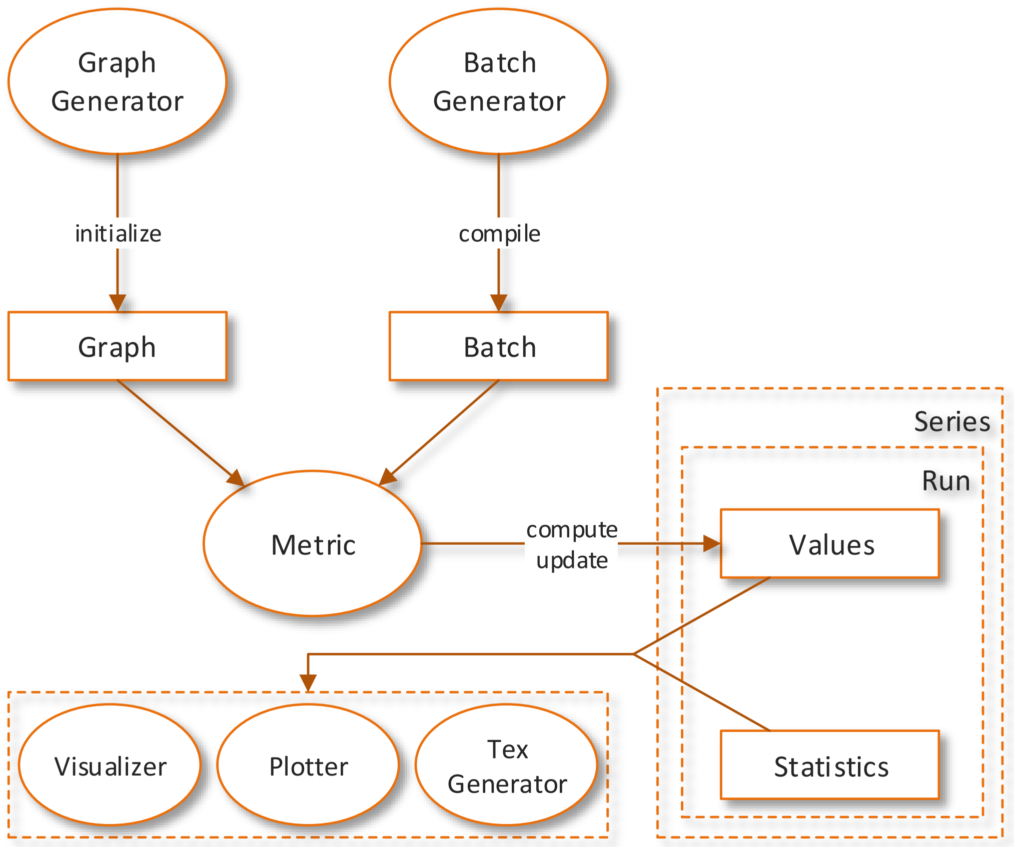


Figure 2.1 – DNA architecture and components.

2.2.1 First approach

The example in 2.3 shows how one could generate a random, undirected graph with initially 100 nodes and 300 edges. With each batch 5 nodes and 20 edges are being randomly added while also 15 edges are being removed. The metrics *DegreeDistributionR* (recomputed) and *UndirectedClusteringCoefficientU* (updated) are chosen to be computed. The resulting filesystem structure can be seen in figure 2.4.

2.3 Auto-generated extra-values

The DNA is able to compute additional extra values for distributions and nodevaluelists. This is enabled by default and can be finely configured in the *settings.properties* configuration file. These include *minimum*, *maximum*, *median*, *average*, *binsize*, *denominator* and additional upper bounds for distributions. Please note, that the median will return the value, which divides the data in two even 50% portions. If such a value does not exist, e.g. when the number of data points is even, the returned value will be the mean between the two median points.

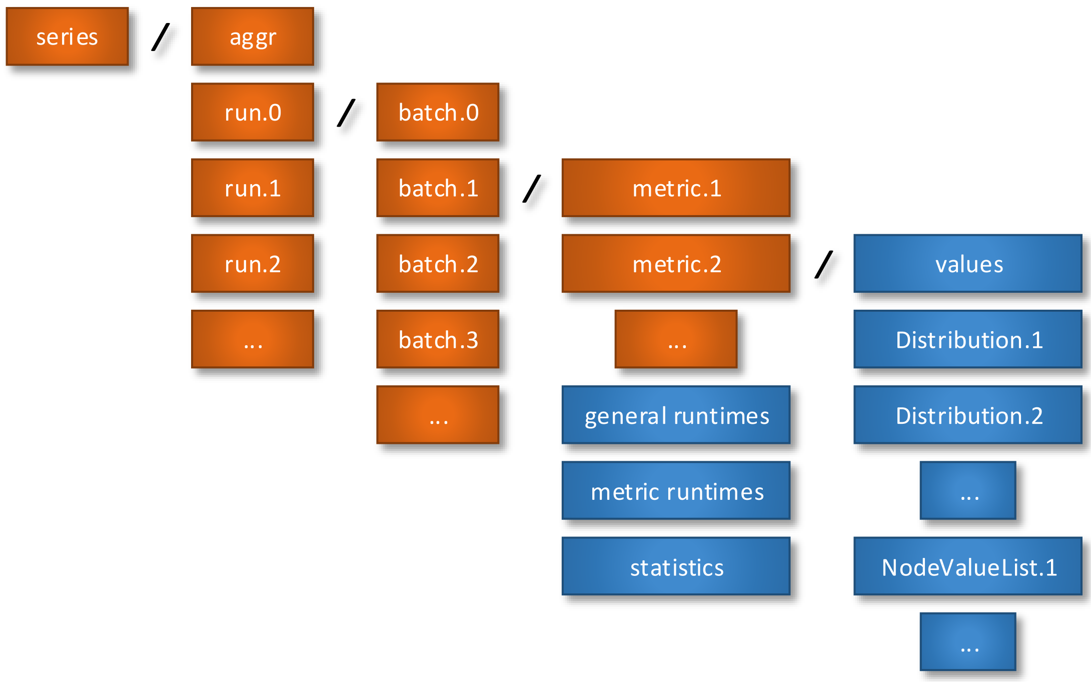


Figure 2.2 – General filesystem structure for storing the results of a series.

Consider the following example:
The metric *DegreeDistributionR* calculates a distribution *DegreeDistribution*. In each batch DNA will automatically compute the additional values *DegreeDistribution_AVG*, *DegreeDistribution_MAX*, *DegreeDistribution_MED* and *DegreeDistribution_MIN* based on the distribution. Note: It is also possible to generate *DegreeDistribution_DENOMINATOR*, *DegreeDistribution_BINSIZE* and additional upper bounds. However, this is disabled by default and may be enabled on demand.

2.3.1 Upper bounds

The minimum and maximum of a distribution may not be very meaningful in a lot of cases. Therefore, additional upper bounds can be generated by DNA. For instance, one may be interested in how the majority of values is distributed. This can be achieved by computing the upper bound for X% of the data. The upper bound of 95% will compute a value, which is a upper bound for 95% of the data. The generation of upper bounds is disabled by default but can be enabled by setting:

```
Config.overwrite("GENERATE_DISTRIBUTION_PERCENT_VALUES", "true")
```

Furthermore, one can explicitly specify, which bounds will be computed by setting a list of doubles (in String format delimited by ','). For instance, computing the upper bound for 95% and 60.5% will look like this:

```
Config.overwrite("EXTRA_VALUE_DISTRIBUTION_PERCENT", "95,␣60.5")
```

```
public static void main(String[] args) throws AggregationException,
    IOException, MetricNotApplicableException,
    InterruptedException {
  String seriesDir = "data/series/";

  // initialization, 100 nodes and 300 edges
  GraphGenerator gg1 = new RandomGraph(GDS.undirected, 100, 300);

  // 5 nodes-added, 0 nodes-removed, 20 edges-added, 15 edges-removed
  BatchGenerator bg1 = new RandomBatch(5, 0, 20, 15);

  // choose metrics
  Metric[] m1 = new Metric[] { new DegreeDistributionR(),
      new UndirectedClusteringCoefficientU() };

  // create series and generate it
  Series s = new Series(gg1, bg1, m1, seriesDir, "series");

  // generate 2 runs with 100 batches each
  SeriesData sd = s.generate(2, 100);
}
```

Figure 2.3 – Simple generation example.

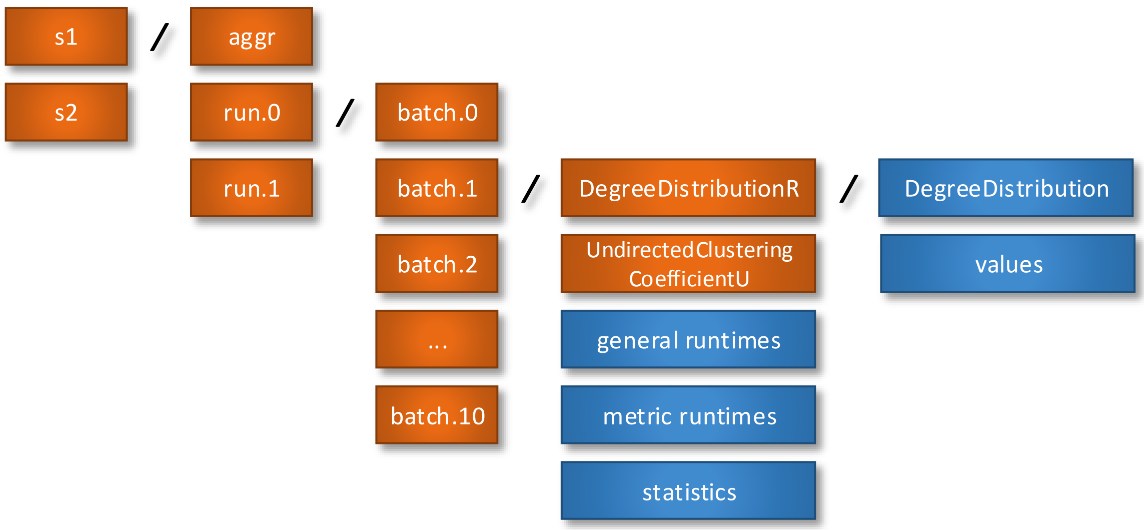


Figure 2.4 – Filesystem structure for example 2.3.

Chapter 3

Evaluation

In order to evaluate the data and reach certain conclusions from the results it is neccessary to take a view at the data directly or atleast to visualize them in a plot or graph. For this purpose DNA offers sophisticated plotting mechanisms, automatic LaTeX output, which includes data tables and plots, and a graphical user interface namely the DNA-Visualizer.

3.1 Aggregation

The aggregation process is enabled by default and takes place at the end of the generation process. During aggregation the data from all runs will be aggregated and stored in a separate aggregation directory *../aggr/* next to the run directories. The aggregations filesystem structure is analogous to the structure of a single run. The only difference is the extended data representation. All values contain the aggregated *average*, *minimum*, *maximum*, *median*, *variance*, *variance_low*, *variance_up*, *confidence_low* and *confidence_up* in this order. A simple example follows:

A series is generated with 2 runs and 100 batches. The metric-value *degreeMax* has the its values shown in table 3.1 and its aggregated data in table 3.2 respectively.

degreeMax		
batch	**run.0**	**run.1**
0	13	15
1	12	16
2	12	15
...	...	...

Table 3.1 – Data values of *degreeMax*.

batch	**avg**	**min**	**max**	**med**	**var**	**var-**	**var+**	**conf-**	**conf+**
0	14.0	13.0	15.0	15.0	1.0	1.0	1.0	5.013	22.987
1	14.0	12.0	16.0	16.0	4.0	4.0	4.0	-3.975	31.975
2	13.5	12.0	15.0	15.0	2.25	2.25	2.25	0.019	26.981
...	...	...	...	...	...	...	...	...	...

Table 3.2 – Aggregated values of *degreeMax*.

3.1.1 Manual aggregation

Several occasions require a series to be aggregated manually. This might be the case when different runs are generated in parallel and aggregation shall take place when all generations are finished. Therefore the aggregation flag should be set as *false* during generation. The aggregation can be started at a later point with:

```
Aggregation.aggregateSeries(SeriesData sd)
```

For a more elaborate example see the code snippet in 3.1. It shows how to properly read and aggregate the series followed by a simple plotting call. Note that the aggregation process, including all computtations and writing of the aggregated data to the filesstem, takes place in the *aggregateSeries* call. Setting the aggregation to the *SeriesData* object is only required for plotting.

```
public static void main(String[] args) throws AggregationException,
    IOException, MetricNotApplicableException,
    InterruptedException {
  String seriesDir = "data/series/";
  String plotDir = "data/series/plots/";

  // read the series (without aggregation or actual values)
  SeriesData sd = SeriesData.read(seriesDir, "series", false, false);

  // aggregate
  AggregatedSeries aggS = Aggregation.aggregateSeries(sd);

  // set aggregation
  sd.setAggregation(aggS);

  // plot
  Plotting.plot(sd, plotDir);
}
```

Figure 3.1 – Reading, aggregating and plotting a series.

3.1.2 Configuration

The dynamic nature of generation may cause different runs of the same series to contain different amounts of batches. This leads to the question how aggregation may handle unavailable values due to differing amounts of batches. Essentially there are two ways to approach this problem, either ignore the absent values at all or assume a specific value for them. The following sections show how one may change which approach will be used and how this affects the results using a simple example of 3 runs each with different ammount of batches.

Ignore unavailable values (/n-mode)

This is the default mode. The aggregation will completely ignore any unavailable values:

degreeMax				
batch	**run.1**	**run.2**	**run.3**	**average**
32	10	12	14	**12**
33	11	-	17	**14**
34	12	-	-	**12**

Table 3.3 – Aggregation for runs with different amount of batches in /n-mode.

Assume unavailable values (/n+1-mode)

The second way to handle absent values is to assume their values to be zero. This mode can be enabled on demand with:

```
Config.overwrite("AGGREGHATION_IGNORE_MISSING_VALUES", "false");
```

The following table shows the differences to /n-mode.

degreeMax				
batch	**run.1**	**run.2**	**run.3**	**average**
32	10	12	14	**12**
33	11	-	17	**9.33**
34	12	-	-	**4**

Table 3.4 – Aggregation for runs with different amount of batches in /n+1-mode.

3.2 Plotting

The plotting mechanisms included in DNA are complex and therefore only a basic example will be shown here. For more details feel free to check the plotting instruction manual. The class *dna.plot.Plotting.java* offers several public methods for plotting. They usually take an array of *SeriesData* objects and a destination directory as input. When the array holds multiple *SeriesData*'s they will all be compared and featured in the same plots. For each plot DNA will generate a gnuplot script in the destination directory and use this to create the respective plot. Code 3.2 shows how to generate and plot two series and the figures 3.3, 3.4, 3.5, 3.6 show some of the resulting plots.

```
public static void main(String[] args) throws AggregationException,
    IOException, MetricNotApplicableException,
    InterruptedException {
  String dir = "data/scenario.1/";

  // init and generation
  GraphGenerator gg1 = new RandomGraph(GDS.undirected, 100, 300);
  BatchGenerator bg1 = new RandomBatch(5, 0, 20, 15);

  GraphGenerator gg2 = new RandomGraph(GDS.undirected, 200, 400);
  BatchGenerator bg2 = new RandomBatch(10, 0, 25, 30);

  Metric[] m = new Metric[] { new DegreeDistributionR(),
      new UndirectedClusteringCoefficientU() };

  Series s1 = new Series(gg1, bg1, m, dir + "s1/", "s1");
  Series s2 = new Series(gg2, bg2, m, dir + "s2/", "s2");

  SeriesData sd1 = s1.generate(2, 10);
  SeriesData sd2 = s2.generate(2, 10);

  // plotting
  Plotting.plot(new SeriesData[] {sd1, sd2}, dir + "plots/");
}
```

Figure 3.2 – Generating two series with the same metrics and plotting them.

3.3 LaTeX

The *dna.latex.Latex.java* class holds several methods to generate complete latex-documents based on input series. Since the latex process itself is complex and can be configured in several ways it is recommended to check out the latex instruction manual for mor details. The code 3.7 shows a simple example on how to generate a series and then create a latex document including plots.

3.4 Visualization

The DNA visualizer is a graphical user interface designed to illustrate and visualize data from single runs. A visualizer can be opened by calling the *main()*-method from the *dna.visualization.MainDisplay.java* class. The default configuration already allows to view all data contained in the desired run. For more instructions on how to use the visualizer and especially how to configure it check out the visualization instruction manual. See figure 3.8 for an example.

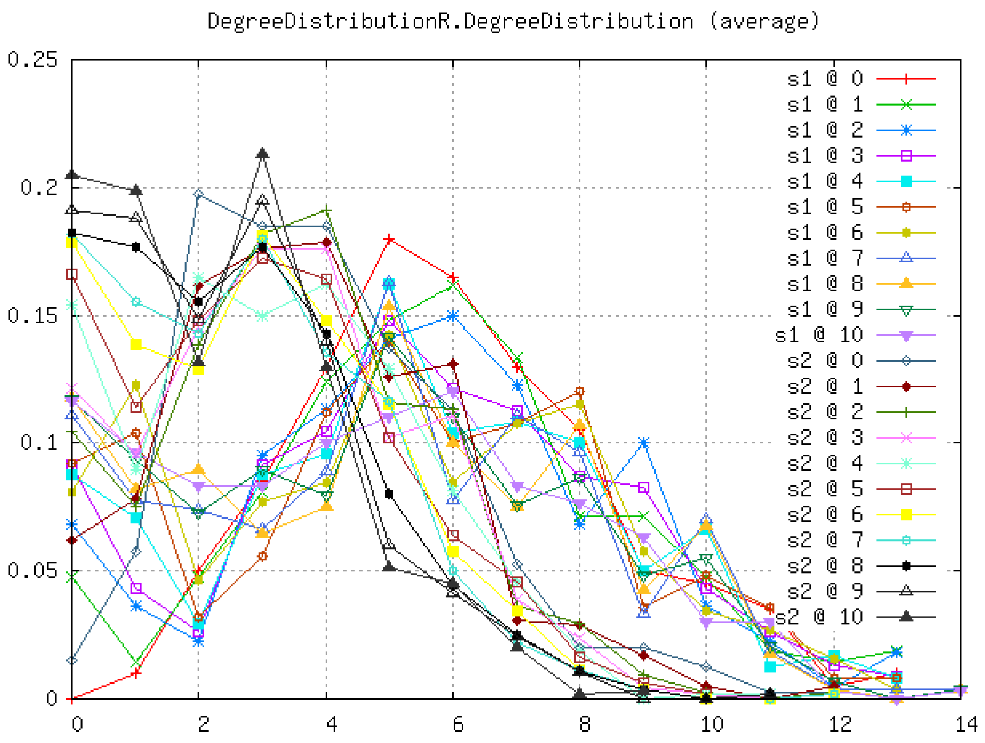


Figure 3.3 – DegreeDistribution plot from the example code snippet 3.2.

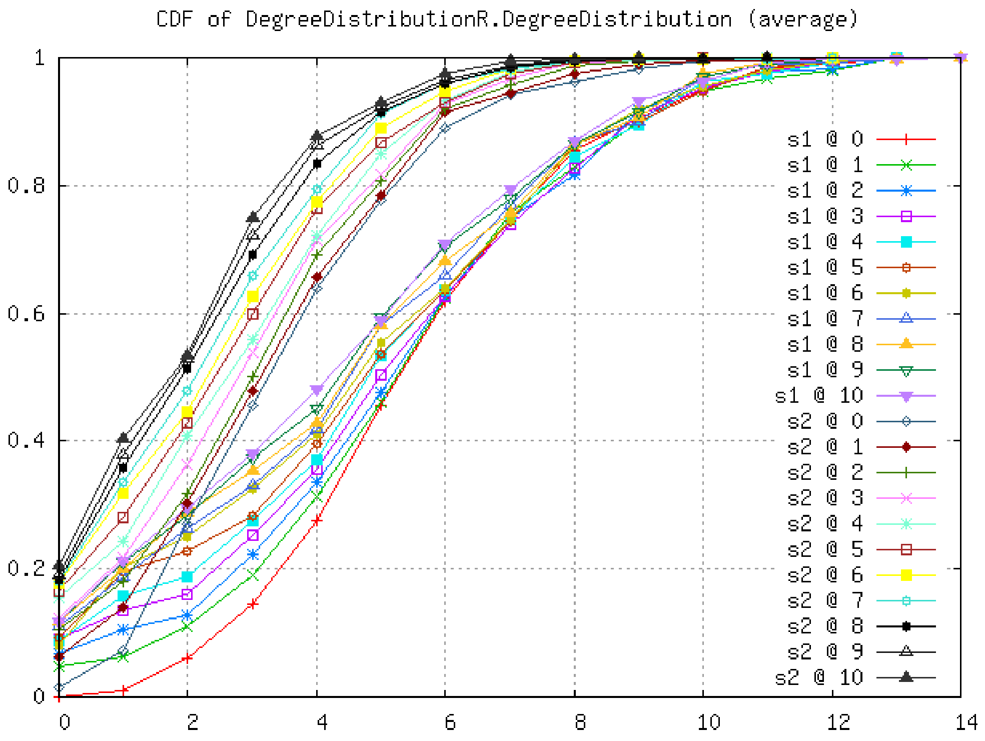


Figure 3.4 – CDF plot of the DegreeDistribution from the example code snippet 3.2.

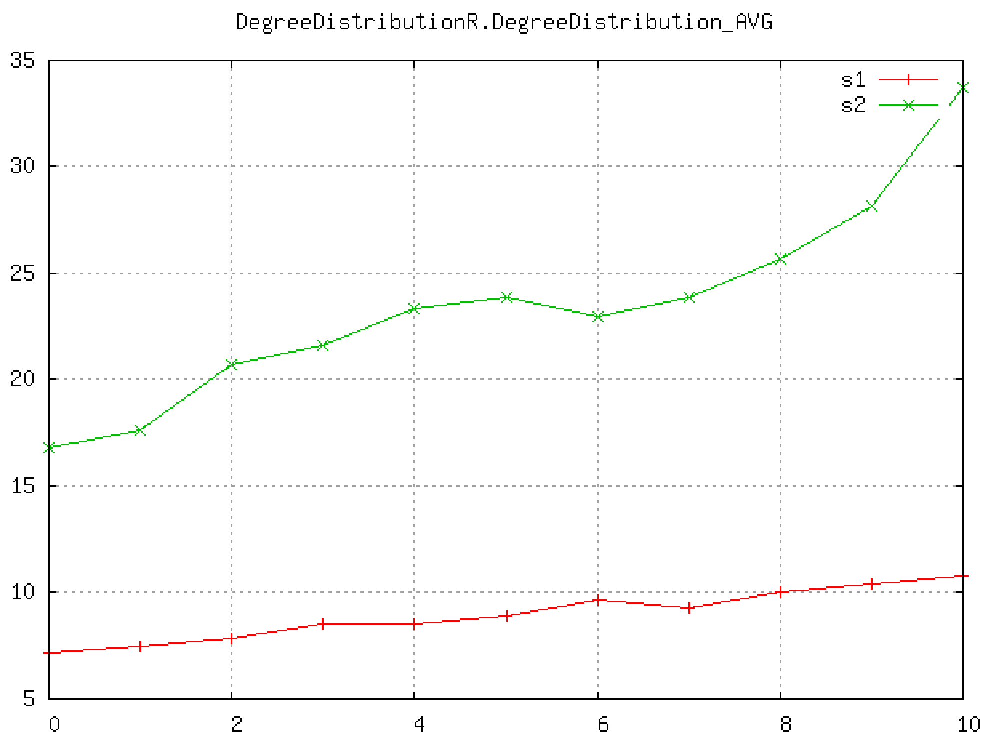


Figure 3.5 – Plot of the auto-generated average value for the DegreeDistribution from the example code snippet 3.2.

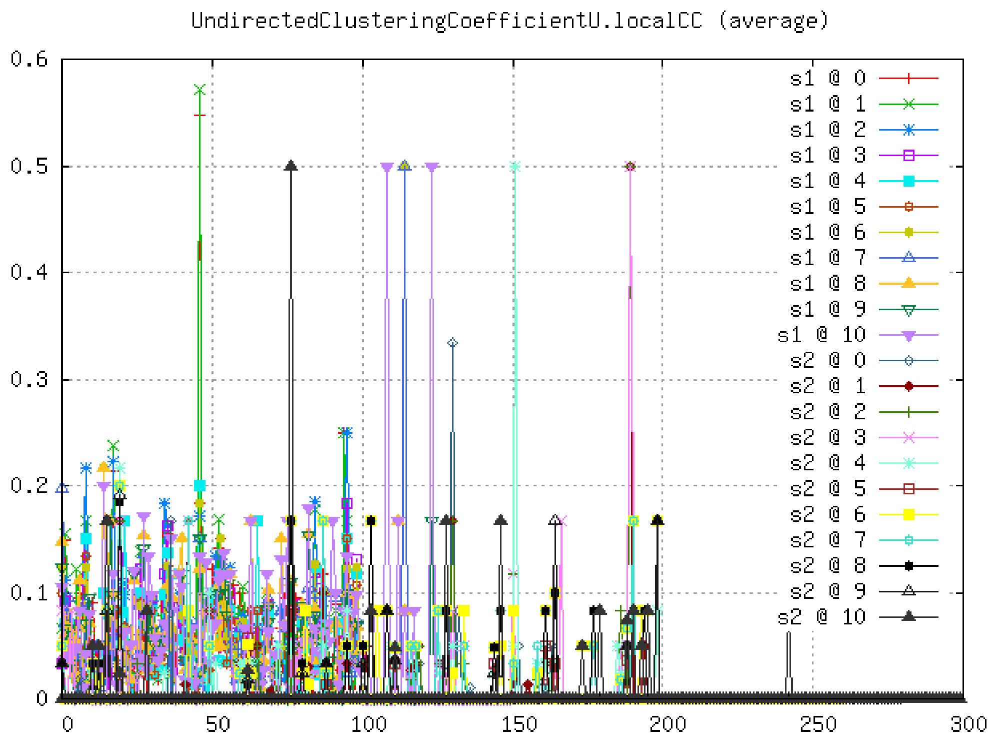


Figure 3.6 – Plot of the NodeValueList localCC from the example code snippet 3.2.

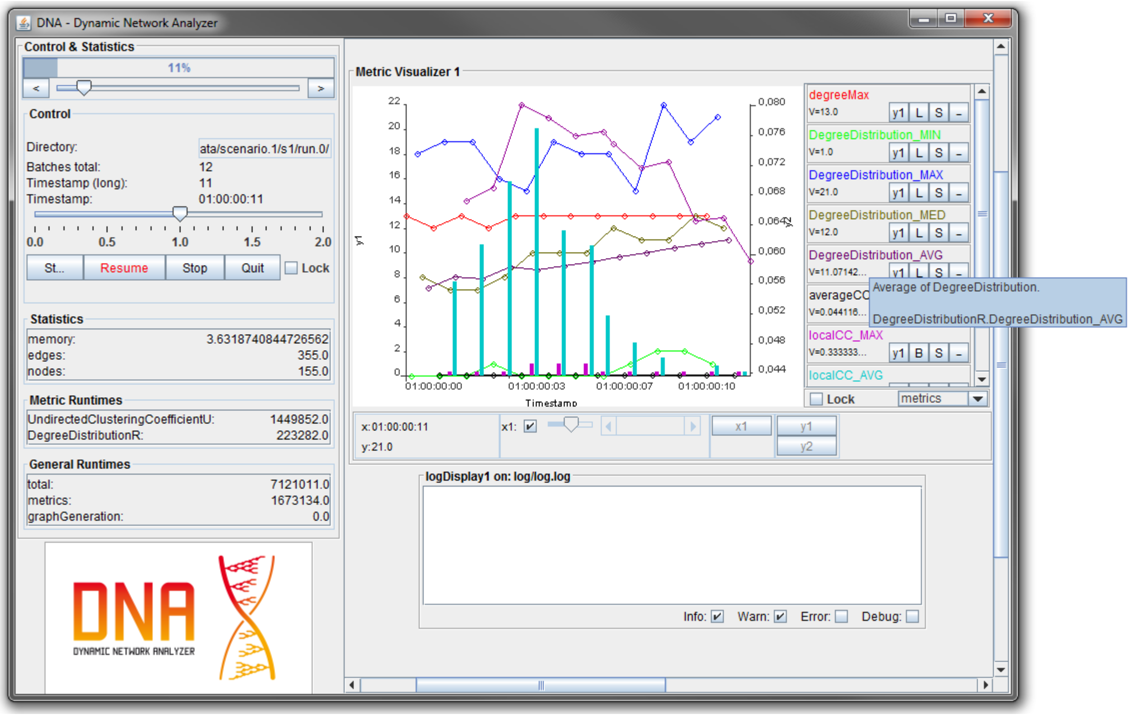
```
public static void main(String[] args) throws AggregationException,
    IOException, MetricNotApplicableException,
    InterruptedException {
  String dir = "data/scenario.1/";

  // init and generation
  GraphGenerator gg1 = new RandomGraph(GDS.undirected, 100, 300);
  BatchGenerator bg1 = new RandomBatch(5, 0, 20, 15);
  Metric[] m = new Metric[] { new DegreeDistributionR(),
      new UndirectedClusteringCoefficientU() };

  Series s1 = new Series(gg1, bg1, m, dir + "s1/", "s1");
  SeriesData sd1 = s1.generate(2, 100);

  // plot & generate latex document
  Latex.writeTexAndPlot(sd1, dir, "preamble.tex");
}
```

Figure 3.7 – Generating series *s1* and create a latex document including plots.



Directory: ata/scenario.1/s1/run.0/
Batches total: 12
Timestamp (long): 11
Timestamp: 01:00:00:11

Statistics
memory: 3.6318740844726562
edges: 355.0
nodes: 155.0

Metric Runtimes
UndirectedClusteringCoefficientU: 1449852.0
DegreeDistributionR: 223282.0

General Runtimes
total: 7121011.0
metrics: 1673134.0
graphGeneration: 0.0

degreeMax V=13.0
DegreeDistribution_MIN V=1.0
DegreeDistribution_MAX V=21.0
DegreeDistribution_MED V=12.0
DegreeDistribution_AVG V=11.07142...
averageCC V=0.044116...
localCC_MAX V=0.333333...
localCC_AVG

Average of DegreeDistribution.

DegreeDistributionR.DegreeDistribution_AVG

x:01:00:00:11
y:21.0

Info: ✓ Warn: ✓ Error: ☐ Debug: ☐

Figure 3.8 – Default configured DNA visualizer.

Chapter 4

Configuration

The configuration approach used in DNA is designed to be flexible and to offer the user additional options rather than limiting them. Configuration files are traditionally stored in the *config/* directory and named by their respective category and end with the suffix *.properties*. All config values set in the configuration files represent default values and changing them will have impact on the DNA workflow. However the configuration is very dynamic and the user may change configurated values during runtime via:

```
Config.overwrite(<config-key>, <property>);
```

Note that this command only overwrites the keys temporarily and has no effect on the actual configuration files. It is also possible to read configuration values from the properties-files. For this purpose the *dna.util.Config.java* class offers several get-methods:

```
Config.get(<config-key>); //returns string
Config.getBoolean(<config-key>);  // returns boolean
Config.getDouble(<config-key>); // returns double
...
```

Be aware that if a certain <config-key>is not present when a get-method is called, it will return *null*.

4.1 Zip-Modes

The generation of series with lots of runs, batches and metrics can lead to a vast amount of single files on the filesystem. In order to reduce the amount of files aswell as the overall disk space allocation it is possible to enable zip-mode. Note that the use of zip-files leads to a higher cpu load which will result in increased runtimes. The zip-modes can either be enabled via the *settings.properties* configuration file with the config-key *GENERATION_AS_ZIP* or during runtime with the shown shortcut methods. Note that internally the shortcuts do nothing but setting the config-key to the corresponding value. Zips are disabled by default:

```
    Config.zipNone();
or
    Config.overwrite("GENERATION_AS_ZIP", "none");
```

4.1.1 Batches

In zipped-batch mode every batch will be written and read as a single zip-file. May be enabled via the following command:

```
        Config.zipBatches()
    or
        Config.overwrite("GENERATION_AS_ZIP", "batches");
```

4.1.2 Runs

In zipped-run mode every run will be written and read as a single zip-file. Note that this greatly reduces the amount of files (one for each run and an additional for the aggregation) but also greatly increases cpu load, because the zip-file has to be accessed for each read and write operation.

```
        Config.zipRuns()
    or
        Config.overwrite("GENERATION_AS_ZIP", "runs");
```

4.2 Visualization and plotting

The DNA plotting and visualization modules each have more complex configuration structures. For details check the respective instruction manuals. However, in order to successfully use plotting it is neccessary to configure the path to gnuplots binary files correctly. Per default it is set to the linux default path:

```
GNUPLOT_PATH = /usr/bin/gnuplot
```

If you are running on windows systems the configuration could look like this (default gnuplot installation directiry):

```
Config.overwrite("GNUPLOT_PATH",
  "C://Program␣Files␣(x86)//gnuplot//bin//gnuplot.exe");
```

Bibliography

[1] Benjamin Schiller, Thorsten Strufe, *Dynamic Network Analyzer - Building a Framework for the Graph-theoretic Analysis of Dynamic Networks*, Summersim, July 2013.