Graphana - user manual

The main functionality of *Graphana* is the analysis of graphs concerning structual properties. These are measured using **graphparameters**. The main input of *Graphana* are graphs and user inputs. The main output are analysis results.

In addition, *Graphana* is able to:

- generate graphs:
  - per random graph generators
  - per script
  - per GUI
- visualize graphs and algorithms
- load external java classes to import algorithms and graph libraries.
- do time measurements on algorithms

If you only want to use the standard analysis functionality of *Graphana*, take a look at the 'quickstart.txt'. This manual goes deeper into the usage and the different functionalities of *Graphana*.

Contents

1 Program flow

Within a shell, the graphana.jar can be started as follows:

```
java -jar graphana.jar
```

The running program is controlled via console inputs.

1.1 Simple session

When a > is shown in the console, the program waits for an user input. The typed input can be executed by pressing enter.
The following sequence of inputs is a simple use case:

```
>loadDIMACS("sample.dim")
>vertexCount
6700
>edgeCount
21274
>QUIT
```

Firstly a graph is loaded from the given DIMACS file. Afterwards, the vertex and edge count of the loaded graph are printed and in the end the program is closed. Other operations, including algorithms, are called the same way as in this simple example.

1.2 Operations

The user inputs (and other inputs that will be described later), as the ones of the simple session example, contain so called **statements**. In simple cases, statements only contain **operation** calls to modify the current graph or to run algorithms. Operations are called using the following syntax:

```
operationKey( argument_1, argument_2, ...argument_n )
```

The most important operations are listed in 'graphana_ops.pdf'. The signature (keyword, parameters, return type) and the description of a particular operation can also be printed at runtime with the HELP keyword:

```
HELP operationKey
```

If only `HELP` is typed in, then a runtime help is printed.
If an operation does not need any arguments then no brackets need to be written, but it is recommended.

For example, the vertex cover size algorithm has the following signature:

```
vertexCoverSize (useGreedy:Boolean) :  Integer
```

That means, the algorithm is called via 'vertexCoverSize' and needs one argument of type

Boolean (to decide, which heuristic to use). Furthermore, the signature indicates that the return value is of type Integer.
Calling the algorithm looks like this:

```
vertexCover(true)
```

In the example, `true` is passed as argument and therefore, a greedy algorithm is used.
If an operation returns a result then the result can be shown for example as a console output, which is the default, or can be written into a text file. The latter will be described later. Some types of results can also be visualized.
Some operations have optional parameters. Within the operation signature, these are denoted with a `=` and a default value behind the parameter type:

```
operationKey( parameter_1 [= default_1], parameter_2 [= default_2], ...)
```

If no argument is given for these then the respective default value will be used.

There are three different types of operations:

- **Graph operations:** During the whole runtime of the program, there is exactly one current graph to operate on (more graphs can be managed in the background, but that is not important for the beginning). Graph operations are working on this current graph. For example loading a graph or adding vertices are graph operations.
- **Algorithms:** Algorithms are special graph operations, which compute graphparameters in most cases. A sample for a simple algorithm is the average vertex degree of a graph.
- **Commands:** These are used to configure the program or to get some system information. Commands do not operate on graphs. For example there are commands for time measurements.

Graphana internally uses various **graph libraries** to create graphs and operate on them. Every graph is built up in exactly one graph library. At runtime the user can chose which library to use. The chosen library influences the performance and the set of available algorithms. Some algorithms do not depend on the internal graph library, others are specialised on particular ones. If it is necessary, *Graphana* automatically converts from one graph library to the one which is needed.
Some libraries do not allow every graph configuration. For example, in JGraphT it is not possible to create graphs which are directed and not simple. Initially, the JUNG2-library is set. It allows every configuration.

1.3 Syntax (examples)

Statements may not only contain operation calls. However, these are sufficient for the basic usage of *Graphana*. In this subchapter, the underlying *Graphana*-syntax is explained with the use of examples.

The first syntax example deals with handling variables:

```
>greetingVar = 'Hello';
>PRINT greetingVar + " World!\n";
Hello World!
```

Firstly a variable is created and assigned by writing the identifier 'greetingVar' followed by a = and the value 'Hello', which is a constant string. Therefore, the variable `greetingVar` is a string from now on. Every variable is global and (usually) lives until quitting the program. String constants can be surrounded either by quotation marks or by tick marks (" or '). The values and also the types of variables can be changed at any time by assigning the variable to a new value.
Afterwards, console output is done by writing the keyword `PRINT` (attention: `PRINT` is not an operation, but a keyword of the syntax which does not need brackets to pass the argument). The printed sample string is concatenated with the + symbol. The '\n' yields a linebreak.

The next example deals with handling numbers:

```
>number = 5;
>PRINTLN number*(3+5) + 16%3;
41
>PRINTLN ++number;
42
>number+5
47
```

The variable `number` is set to 5, therefore it's type is Integer. In the second line, a mathematical term is calculated and printed. The % symbol stands for modulo. The next input increases `number` and prints out the new value. If the ++ symbol would have been placed after the `number` then it still would be increased, but the non-increased value would have been printed. The last input has no `PRINTLN` but still the result is printed. The reason is, that the result of the input, if it has one, is always printed. In this case, the result of 'number+5' is 47. Even an assignment has a result: the assigned value. To prevent printing the result, the input must end with a ; (like it was done in the previous inputs). Leaving out the semicolon is a comfortable way for quickly printing results (as seen in the very first example).

The semicolon is also necessary to execute more than one statement within one line. In the following example, the statement contains several sub statements seperated by semicola:

```
>PRINTLN " 'number' before:\t"+number; number*=2; PRINT " 'number' after:\t"; number
 'number' before: 42
 'number' after: 84
>quit
```

In this example, four statements are executed iterativly:

- Print the value of the number with no changes.
- Multiply the number with 2.
- Print "'number' after:" but not number
- Execute `number`, which means in this case just take the value of number

The last statement is not closed with a semicolon so the value of number is the end result of the input and is printed even without a `PRINT` keyword.

Statements also can be read from a script file. This is done with the 'script' command:

```
script("scripts/operations.txt");
```

Scripts are written with the same syntax as console inputs, but they are not read linewise. So a statement with no semicolon followed by another statement in the next line is a syntax error in most cases. Line comments begin with `//`. Block comments are surrounded with `/*` and `*/` and can be nested. Comments are completely ignored when executing the script.

For example a script looks like the following one:

```
//Create and load graph
createGraph(false,false,true,JGraphT);
loadGraph("graphs/sample.dim");

//Print some properties
PRINTLN "Graph size: " + graphSize();
PRINTLN "Vertex cover: " + vertexCoverSize(false);
PRINTLN "Max flow between 'pita' and 'fan': " + maxFlow($pita,$fan);
```

At first, the graph is created using the 'createGraph' command. This command initializes an empty graph with the given configuration: In the example, the first three arguments determine that the graph is undirected, unweighted and always simple. The fourth argument determines the graph library which is to be used internally. In this case, the JGraphT library is set. As a reminder for the arguments to pass, `HELP createGraph` can be typed in. After the graph is created and initialized, it is constructed by loading a DIMACS file. The graph is unweighted and therefore weights are ignored when reading the file.
After the graph was loaded, some data and graphparameters are printed. The `PRINTLN` keyword prints the given **String**. In the example, some graph operations and algorithms are called. Firstly, `graphSize` is called. This graph operation does not need any arguments so the brackets are empty. In this case, the brackets may be omitted (like it was done in the very first example). The operation `maxFlow` needs two arguments. In the example, two vertices are given as arguments. Vertex constants are written with `$`*`vertexIdentifier`*. A vertex with the respective identifier must exist in the current graph.
As already mentioned above, `PRINT` and `PRINTLN` are no operations, but keywords, which do not need any brackets. The result of the whole Expression will be printed.

The next sample script demonstrates, how to write some graphparameters into a formatted text file:

```
//Write graphparameters into a file
setOutputFile("Graphparameters.txt",true);
WRITE "Vertex count: " + vertexCount() + ", ";
WRITE "Edge count: " + edgeCount() + "\n";
WRITELN "Average Degree: " + avrgDegree();
PRINT "Diameter: " + diameter();
closeFile();
PRINTLN "Wrote into 'Graphparameters.txt'";

//Save degree distribution as a CSV file
distribution = degreeDistribution();
writeWholeFile("degrees.txt",distribution);
```

At first, the output file is set. The file writing will be done into the chosen file from then on. The second argument of the `setOutputFile` determines, whether (almost) every console output (including warnings and errors) is also to be written into the output file. The `WRITE` keyword

works in a similar way as the `PRINT` keyword, but writes into the file, which was set previously, instead of the console output. In the example, three strings are explicitely written into the file. Afterwards, there is a `PRINT` keyword. The given string will be printed in the console output as always, but since the second argument of the `setOutputFile` at the beginning is set to true, the string will also be written into the output file. When the file writing is complete, the file must be closed using `closeOutputFile`. Now the file is complete and can be read in the file system. After the call, there is no output file set and therefore it is not allowed to call `WRITE` until a new output file is set.
The last part of the example demonstrates how to create a whole text file at once. At first, a **Histogram** is created representing the degree distribution of the current graph. The operation `writeWholeFile` only needs two arguments: The first one is the target file and the second one any object. The text representation of the given object will be written into the target file. The file will be closed automatically. In the example, a **Histogram** is passed. The string representation of a histogram is a CSV string. So the resulting file is a CSV file and can be used in respective external programs for example.

1.4 Program parameters

Graphana can be started with certain program arguments. All program parameters are optional. The first program parameter can be used to pass a filename of a script file. This script file will be executed before the first user input is possible. If the file ends with a `QUIT`, then *Graphana* closes without any user input. After the first argument, an arbitary number of statements can be given in *Graphana*-syntax. These arguments will be executed prior to the given script. This can be used for example to set global variables which are used inside the script. If the keyword **-post** is passed instead of a statement, then all following arguments will be executed after the script is finished. Note that since graphana is not running yet, the syntax of the shell has to be regarded, especially the fact, that whitespaces are seperators for arguments. It is recommended to put the arguments in quotation marks.
In addition to the script that is passed as argument, the script 'init.txt' is always executed if the respective file exists in the current directory.

So all in all the order of execution when starting the program is the following:

1. The 'init.txt' - script file
2. The second and following arguments interpreted as statements until the **-post** keyword
3. The script file, given as the first argument
4. The arguments that come after the **-post** keyword as statements

The following example shows a sample program call of the standard analysis script:

```
java -jar graphana.jar analysis "graphDir='graphs/'"
"output='graphparameters.txt'" -post QUIT
```

The first argument of the example is the file 'analysis'. The script accesses some global variables some of which are set in the following two statements (before the script execution starts). So semantically, these are arguments for the script call. The first one determines the directory from which the graph files are to be loaded. The second one determines the relative path of the output file in which the graphparameters are to be written into. What follows is the **-post** keyword. Hence the `QUIT` keyword is executed after the script execution is completed so the program closes instead of waiting for user input.

The use of the analysis script, which is quite important, is described more detailed in 'quick-start.txt'.

1.5 Advanced usage

What follows in this subsection is necessary to write more complex script files.

The first example deals with conditions:

```
if(averageDegree()<=5 && maxDegree()<=10)
    setCVDHeuristics("SUCCESSIVE - MAXDEGREE");
else
    setCVDHeuristics("CONNECTEDCOMPONENT - ALL");
PRINTLN cvdSize();
```

The example takes the average degree and the maximum degree of the current graph to decide which cvd heuristic (see 'graphana_ops.pdf') might be the most applicable for the graph.

The next example deals with for-loops:

```
for(i=1;i<=120;i++)
{
    createRandomGraph(i,0.5);
    writeWholeFile("graphs/random_graph_"+i+".dim", graphAsDimacs());
};
```

Note the semicolon at the very end of the sample. In *Graphana* also statement blocks are closed with semicola.
In the loop, 120 graphs are created with a simple random graph generator and saved as particular dimacs files.

Another loop is the for-each-loop. The loop iterates over a **Set** or a **Vector**.
The following example iterates over a constant set:

```
foreach(x in {3,7,10,11})
{
    PRINTLN x*2;
};
```

The for-each-loop is useful when working with multiple input files (in most cases graph files). The following example uses the `getFiles` command, which returns a set of files that are contained in the given directory (non-recursive) and have one of the given file extensions ("‘.dim"’ in this case):

```
foreach(file in getFiles("graphs/","dim"))
{
```

```
loadGraph(file);
    PRINTLN "--- " + file + " ---";
    PRINTLN "Vertices: " + vertexCount + ", edges: " + edgeCount;
    PRINTLN "Average degree: " + averageDegree
     + ", max degree: " + maxDegree;
};
```

2 Using graphana as framework

The functionality of *Graphana* can also be accessed within a java application. This section demonstrates the easiest way to do this.

To use the framework in an eclipse project, the 'graphana.jar' must be added as external jar. This can be done by clicking **Project** → **Properties** → **Java build path** → **Libraries** → **Add external jars** and choosing 'graphana.jar'.

The easiest way to initialize the framework and to execute operations is to create a `GraphanaAccess` instance. This class initializes *Graphana* and automatically registers all the default operations and libraries:

```
GraphanaAccess graphanaAccess = new GraphanaAccess();
```

After the framework is initialized, operations can be called by using the `executeX` methods where X stands for the type which is expected to be returned. So for example, if an operation, which returns an integer, is executed, then `executeInt` must be called. The methods expect a string in graphana syntax as input argument.

The following example initializes the framework, creates a random graph and prints the average degree:

```
GraphanaAccess graphanaAccess = new GraphanaAccess();
graphanaAccess.execute("createErdosRenyiGraph(10,0.5)");
float averageDegree = graphanaAccess.executeFloat("averageDegree()");
System.out.println(averageDegree);
```

The next example loads a graph instead of creating one and prints the number of connected components:

```
graphanaAccess.execute("loadgraph('graphs/sample.dim')");
int ccCount = graphanaAccess.executeInt("getConnectedComponentCount()");
System.out.println(ccCount);
```

If there is already a graph instance within the application, it is also possible to apply graph operations on this graph by using the method `GraphanaAccess.setCurrentGraph`. The method can either be called with a JUNG2 graph, a JGraphT graph or an instance of `GraphLibrary` (the latter is described in 'graphana_extension.pdf'). All graph operation calls which follow after the `setCurrentGraph` call will use the given graph.
The following example demonstrates the usage of the framework with a externally created graph instance:

```
public static void main(String[] args)
{
  //Construct sample JGraphT-graph
  SimpleWeightedGraph<String,Object> jGraphTgraph =
    new SimpleWeightedGraph<String, Object>(
      JGraphTWeightedStatusEdge.class
      );
    jGraphTgraph.addVertex("V1");
    jGraphTgraph.addVertex("V4");
    jGraphTgraph.addVertex("V8");
    jGraphTgraph.addVertex("TT");
    jGraphTgraph.addVertex("X");
    jGraphTgraph.setEdgeWeight(jGraphTgraph.addEdge("V1", "V4"), 7);
    jGraphTgraph.addEdge("V8", "V4");
    jGraphTgraph.addEdge("TT", "X");


    //Use framework
    GraphanaAccess graphanaAccess = new GraphanaAccess();

    graphanaAccess.setGraph(jGraphTgraph);
    System.out.println(
      graphanaAccess.executeInt("vertexCover(true)")
      );
}
```

The sample firstly creates a small JGraphT graph using the methods of JGraphT. Afterwards, the framework is initialized and the just created graph is given in to compute the vertex cover size of the graph.
If no graph is given in (like in the first two examples) then an empty JUNG2 graph is set.

Note that only one `GraphanaAccess` instance should be created per application, since the constructor invokes a complete initialization of the framework. Furthermore, if graphs are not passed as `GraphLibrary`, they are converted within `setCurrentGraph` so this method should only be called, if the graph changed.

All previous samples did not do error handling. Therefore, if errors would occur, the stack trace would be printed and the program would be closed. The `GraphanaAccess` methods throw `GraphanaRuntimeExceptions`. These can be catched to extract detailed error informations and to print appropriate error messages. The following sample tries to create a graph and to visualize it, whereas some exceptions are thrown:

```
public static void main(String[] args)
{
    //Initialize
    GraphanaAccess graphanaAccess = new GraphanaAccess();

    //Output a string
    try{
      //fails because of incomplete statement
      graphanaAccess.execute("PRINT␣'Hello");
    }catch(GraphanaRuntimeException exception) {
      System.err.println(exception.getMessage());
    }

    //Create some vertices and edges
    try{
```

```
    graphanaAccess.execute("createGraph(true, true);");
    //fails because of forgotten second integer argument:
    graphanaAccess.execute("addVertexRow(10, 'vertex')");
    //not executed:
    graphanaAccess.execute("addEdge($vertex1, $vertex2)");
    graphanaAccess.execute("addEdge($vertex4, $vertex7)");
  }catch(GraphanaRuntimeException exception) {
    System.err.println(
        "Error @" + exception.getInputKey() + ": " +
        exception.getExecutionError().getStringRepresentation()
        );
    //or simply: System.err.println(exception.getMessage());
  }

  //Visualize the graph
  try{
    graphanaAccess.getUserInterface().showGraph("GRD",true);
  }catch(ArrangeException exception) {
    //fails because of invalid layout key:
    System.out.println("Graph visualization failed: "
      + exception.getStringRepresentation());
  }

  //Print a sum
  try{
    //fails because of wrong return type:
    System.out.println(
    graphanaAccess.executeInt("1.0 + 3.2")
    );
  }catch(GraphanaRuntimeException exception) {
    System.err.println(exception.getMessage());
  }
}
```

Furthermore, it is possible to simply start the user interface by calling

```
graphanaAccess.getUserInterface().mainLoop();
```

This will start the same user interface as the one of the execution of graphana.jar. The method is blocking until the user quits.

Appendix A: Syntax

The following table describes the whole syntax of the *Graphana* script language. Syntax which is written in bold square brackets is optional:

Name	**Syntax**	**Description**
Application	Ide`(`x_1`,`x_2`, ...`x_n`)`	Executes the operation with the key Ide with the given arguments. Returns the result of the execution
Addition	x `+` y	Returns the sum of x and y or the concatenation if x or y is a String
Subtraction	x `-` y	Returns the difference of x and y
Multiplication	x `*` y	Returns the product of x and y
Division	x `/` y	Returns the quotient of x and y. If x and y are integers then integer division is done
And	x `&&` y	Returns true if and only if x and y are both true. If x is false then y is not evaluated
Or	x `||` y	Returns true if and only if x or y is true. If x is true then y is not evaluated
Unary Minus	`-`x	Returns the negative value of x
Not	`!`x	Returns true if and only if x is false
Equal to	x `==` y	Returns true if and only if x is equal to y
Not equal to	x `!=` y	Returns true if and only if x is not equal to y
Less than	x `<` y	Returns true if and only if x is smaller than y
Greater than	x `>` y	Returns true if and only if x is greater than y
Less than or equal	x `<=` y	Returns true if and only if x is smaller than or equal to y
Greater than or equal	x `>=` y	Returns true if and only if x is greater than or equal to y
Assignment	Ide `=` X	Assigns X to the variable Ide. Creates the variable if it does not exist. Returns X
Postfix increment	$Ide{+}{+}$	Increases the value of the (existing) variable Ide by 1. Returns the value of Ide before it was increased
Prefix increment	${+}{+}Ide$	Increases the value of the (existing) variable Ide by 1. Returns the value of Ide after it was increased
Postfix decrement	$Ide{-}{-}$	Decreases the value of the (existing) variable Ide by 1. Returns the value of Ide before it was decreased
Prefix decrement	${-}{-}Ide$	Decreases the value of the (existing) variable Ide by 1. Returns the value of Ide after it was decreased
Statements	X_1`;`X_2`; ...;`X_n	Executes X_1 to X_n. Returns result of X_n

Name	**Syntax**	**Description**
If-then-else	`if(`*Cond*`)` *ThenStmnt* `[ else` *ElseStmnt* `]`	Executes *ThenStmnt* if *Cond* is true. Executes *ElseStmnt* if it is given and *Cond* is false. Returns: • Result of *ThenStmnt* if *Cond* is true • Undefined if *Cond* is false and no else is given. • Result of *ElseStmnt* if *Cond* is false and an else is given.
While loop	`while(`*Cond*`)` *Stmnt*	Executes *Stmnt* as long as *Cond* (Boolean) is fullfilled.
For loop	`for(`*Init*`;`*Cond*`;`*Iter*`)` *Stmnt*	Firstly executes *Init* and then repeatedly *Stmnt* and *Iter* as long as *Cond* (Boolean) is fullfilled.
Vector	`(`x_1`,`x_2`, ...` x_n`)`	Returns a vector with the given entries.
Vector access	*Ide*`[`*i*`]`	Returns the *i*-th entry of the (existing) vector *Ide*. The first entry is at i=0. The last entry is `size(`*Ide*`)-1`.
Set	`{`x_1`,`x_2`, ...` x_n`}`	Returns a set containing the given values.
Try-catch	`try` *TryStmnt* `[catch(`*ErrIde*`)` *CatchStmnt*`]`	Executes *TryStmnt*. If an error occurs, execution is aborted, the error is stored in *ErrIde* and *CatchStmnt* is executed. Returns true if and only if there was no error in the execution of *TryStmnt*

Further information:

- The *Graphana* language is not case sensitive, but the identifiers of vertices are.
- Whitespaces can be inserted arbitrarily.
- It is not possible to declare variables with the same identifier as a prefix keyword of the syntax, an operation or a type.
- Identifiers must not start with a digit.

Appendix B: Some types

As already demonstrated in the previous subsection, some operations are called with one or more arguments of various types. A list of the most important types for a quick overview is given below.

Type	**Description**	**Examples (in *Graphana*-syntax)**
Integer	Integral number	67 -45
PositiveInteger	Natural number	32 0
Float	Floating point number	5.6 -3.0
Boolean	Truth value	true false
String	Character string	”Long Text” ’Name’
File	A file or a filename as string	”directory/new_file.txt”
ExistingFile	An existing file	”directory/file.txt”
Graph	A whole graph	`getCurrentGraph()`
Histogram	Histogram or CSV-string	`distanceDistribution() newHistogram()`
Color	RGB-Color	`color(255, 128, 0)`
Vertex	A vertex of the graph	`$v1`
Edge	An edge of the graph	`$v1−−$v2`

To lookup operations, terms and all types you can read the ”graphana_ops.pdf” or type `HELP` followed by the operation, the term or the type in the running program.