Graphana - user manual

The main functionality of *Graphana* is the analysis of graphs concerning structural 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:
  - by random graph generators
  - by script
  - by GUI
- visualize graphs and algorithms
- load external java classes to import algorithms and graph libraries.
- do time measurements on algorithms

Contents

1 Program flow **2**
- 1.1 Text input and output . . . 2
- 1.2 Main menu . . . 3
- 1.3 Status bar . . . 4
- 1.4 Graph file set window . . . 4
- 1.5 Analysis . . . 4
- 1.6 Histograms . . . 5

2 Script language **7**
- 2.1 Operations . . . 7
- 2.2 Syntax (examples) . . . 8
- 2.3 Advanced usage . . . 10

3 Using graphana as framework **12**

Appendix A: Syntax **15**

Appendix B: Types **17**

Appendix C: Supported Graph Formats **18**

1 Program flow

Graphana can be started with the following command:

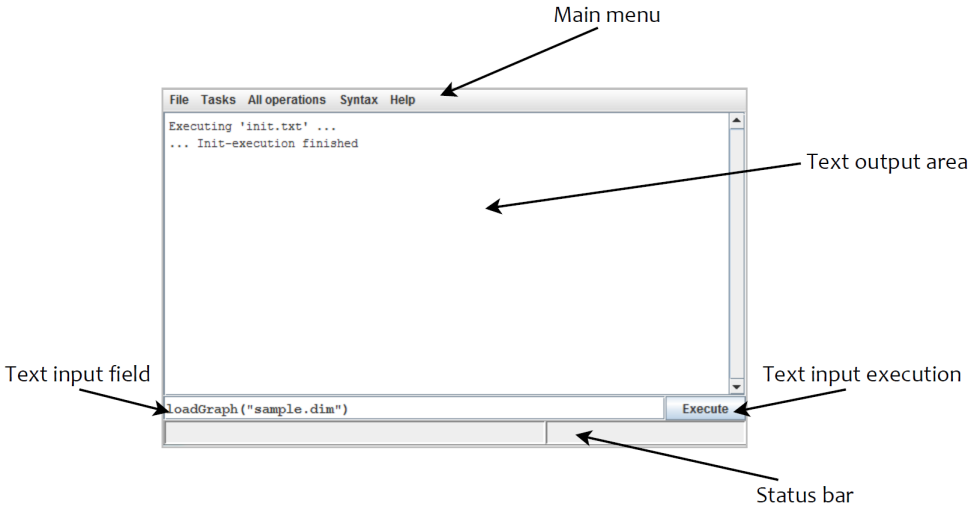
```
java -jar graphana.jar
```

Since *Graphana* sometimes needs much memory on large graphs, it is recommended to run it with a large heap as well as with a large stack, when planning to load large graphs. For example:

```
java -jar graphana.jar -Xms1024m -Xss16m
```

This runs graphana with 1GB heap space and 8MB stack memory. If the memory is not sufficient, *Graphana* may return an error when running respective algorithms on too large graphs.

When *Graphana* is started, a window appears containing the following elements:



1.1 Text input and output

The text input field gives access to all functionalities of *Graphana*. Text can be typed in, which will then be evaluated by *Graphana*. The text output will then print the given input and if the evaluation yielded a result, it will be printed as well.
The typed input can be executed by pressing enter or by pressing the **Execute** button. The following sequence of inputs form a simple use case:

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

In the examples of this manual, the user inputs are denoted with a '>' at the beginning of a line. All other lines are text output.

Firstly a graph is loaded from the given DIMACS file. In *Graphana*, the `loadGraph` statement is a so called **operation**. This will be described in detail later. After loading, the vertex and edge count of the loaded graph are printed by typing and executing `vertexCount` and `edgeCount`. In the end the program is closed. Other operations, including algorithms, are called the same way as in this simple example.

By pressing up and down, it can be navigated through previously executed inputs. When typing an operation, tab can be pressed to complete the operation identifier. For example, if "loadgr" is typed, then pressing tab yields "loadGraph()" (note that typing "load" and pressing tab won't work, because several operations start with "load"). When pressing tab while the caret is positioned between empty brackets, a dialog is opened to assist setting up the arguments of the call. So in the "loadGraph()" example, a dialog will appear to choose the file argument via open dialog.

1.2 Main menu

The main menu consists of the following items:

- **File:** Creating and loading graphs.
- **Tasks:** Default program tasks. Tasks are sequences of operations.
- **All operations:** Contains all *Graphana* operations. By clicking the respective operation a call assistant of the operation appears including a description of the operation. By setting up the arguments and pressing **Execute** the operation is executed. Alternately, **Insert** can be pressed to only insert the operation call as text into the text input field. It can then be executed by pressing enter.
- **Syntax:** Contains all keywords of the script language (the language is described in the next section). By clicking the repective keyword, it can be inserted into the text input field.
- **Help:** Documentation and informations about the program.

In general, everything which can be done with the main menu also can be done with the text input field (except for the **Help** item). The main menu can be seen as a collection of default text inputs to make it easier to work with the program especially without knowing all operations and the syntax.
For example: the same use case as shown in the previous subsection can be achieved with the main menu without knowing the commands by doing the following:

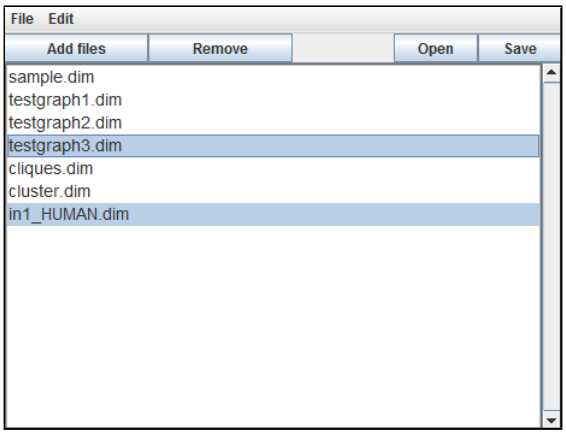
- **File→ Load graph**, clicking "open", choosing "sample.dim" and confirming.
- **All Operations → Graph operations → General graph properties → vertexCount → Execute** then pressing enter. The same with edgeCount.
- **File→ Exit**.

1.3 Status bar

The status bar informs the user about the current state of the program. When the program is idle, then nothing is displayed. During computations, a label shows the current activity and a progress bar depicts the percental progress of it. In addition, a stop button appears when starting a computation. This can be used to interrupt the current computation.

1.4 Graph file set window

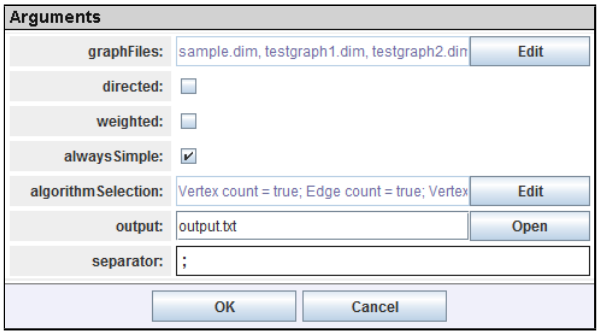
The graph file set can be opened by clicking **File** → **Graph file set**.



The graph file set offers quick user access to often used graph files. Graph files can be added to the list by pressing **Add files**. Multiple files can be chosen by holding Ctrl or Shift, respectivley. After confirming, the chosen files appear within the list. By double clicking one of the graphs in the list, the graph is loaded into the program. This is a quick alternative of using the `loadGraph` operation. Graph files can be marked and then removed with the **Remove** button or by pressing the delete key. In the list, multiple files can be marked at once by holding Ctrl or Shift. However, only one graph can be loaded at a time. The whole list can be saved by clicking **Save** and choosing a file and loaded by clicking **Open** and choosing the respective file. In addition, the current graph file list is always saved when closing *Graphana*. The list is then loaded with the next start of *Graphana*.

1.5 Analysis

One important functionality of *Graphana* is the analysis of multiple graph files. The analysis task can be found in the main menu under **Tasks**→ **Analysis**.



- **graphFiles:** The files to analyze. By pressing **Edit** a popup menu appears containing the same interface as contained in the previously described graph file set window. By

default, the set is equal to the main graph file set window, but the set can be edited freely. This has no influence on the main graph file set.

- **directed, weighted:** With these checkboxes, the respective property can be set for the graphs to load. The settings only apply to graph formats which do not specify this properties. For example, the DIMACS format does not specify if the graph is directed as opposed to the Dot format. So when loading a graph file of the Dot format, the **directed** checkbox has no effect.

- **forceLoopFree:** If this checkbox is checked, then loops are always ignored when loading graphs.

- **algorithmSelection:** The algorithms to compute on every chosen graph. After pressing **Edit**, the checkboxes can be used to determine which algorithms to run. Every selected algorithm will be run on every chosen graph file. For a description of the algorithms, see "graphana_ops.pdf".

- **output:** The file to write the result into. The analysis result is a CSV table in which the columns are the selected algorithms and the rows are the chosen graph files. The values are the algorithm results.

- **separator:** The value separator for the CSV output.

1.6 Histograms

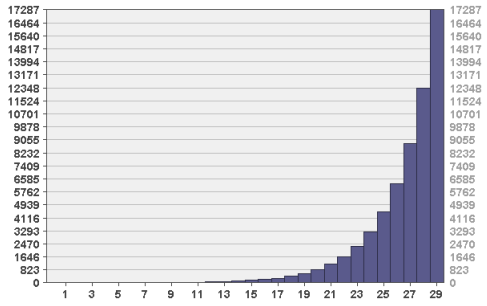
Some graph algorithms deliver not a single number but a 2D-mapping. These mappings can be saved as CSV file or can be visualized directly within the program. One possibility is the use of the **Show histogram** task of the **Tasks** menu. After choosing the algorithm to execute and pressing **Execute** the histogram viewer is displayed and shows the result of the chosen algorithm. For example, if "degrees" was chosen, then the X-axis is the degree and the Y-axis the number of vertices with that degree.

The histogram viewer contains a menu bar in the upper edge:

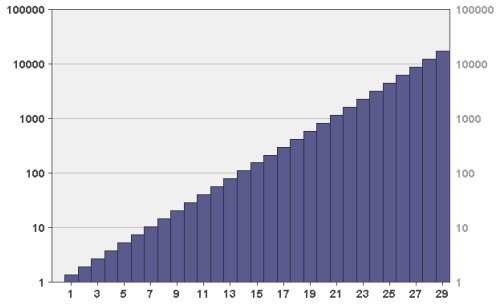
The left buttons can be used to export the histogram as a CSV string. It can be exported as a file or it can be put it into the clipboard (for example to insert it directly into a spreadsheet program like *OpenOffice*).

The checkboxes on the right hand side influence the visual output of the histogram:
If **logarithmic scale** is activated, then the Y-Axis is a log scale axis. This is useful, whenever the values are varying widely. Especially, the log scale yields a linear output for exponential mappings, as depicted in the following example:

Normal scale

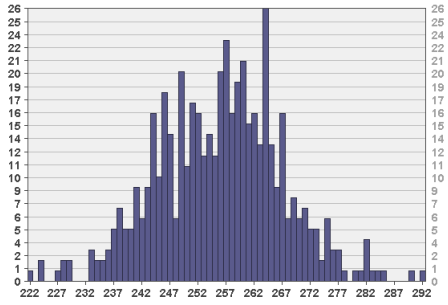


Log scale

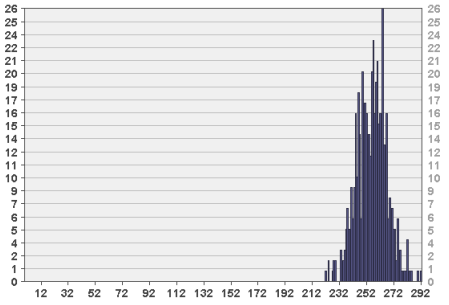


If **Begin at index 0** is not checked, then the X-axis starts at the first index where the value is non-zero. Otherwise the X-axis always starts at index zero:

Begin at first non-zero index

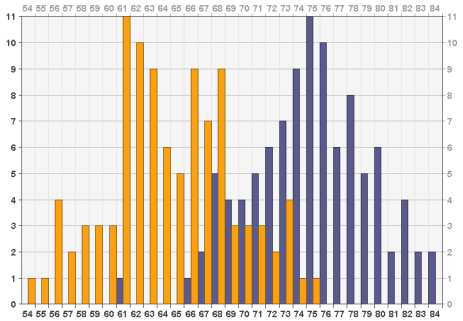


Begin at zero index

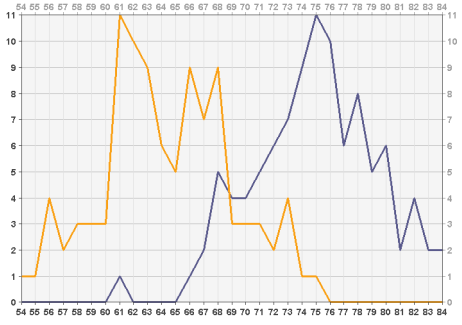


The histogram view can display multiple histograms at once. For example, after the degrees of one graph are displayed, another graph can be loaded without closing the histogram view. When running the histogram task again without enabling the **clearPrevious** checkbox, the degree distribution will be displayed within the same view. This enables a visual comparison. Sometimes it is better to select the **Lines** checkbox when displaying multiple histograms:

Multiple histograms



The same with lines mode



To use separate views instead, different **title**s must be given when running the histogram task.

2 Script language

The user inputs (and other inputs that will be described later), as the ones of the simple session example, contain so called **statements**. This section describes syntax and semantics of these statements.

2.1 Operations

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*)

If an operation does not need any arguments then no brackets need to be written, but it is recommended.
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.

The vertex cover size algorithm is one example operation. It 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 for undirected graphs to contain loops. Initially, the JUNG2-library is set. It allows every configuration.

2.2 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;
6
>number+5
11
```

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 11. 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 separated by semicolons:

```
>PRINTLN " 'number' before:"+number; number*=2; PRINT " 'number' after:"; number
 'number' before: 6
 'number' after: 12
>quit
```

In this example, four statements are executed iteratively:

- 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 parsed line wise. 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 forced to be loop-free. 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` 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.

2.3 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 loop-free 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;
};
```

3 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 an 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 console user interface by calling

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

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 the ”graphana_ops.pdf” can be used. Alternatively, `HELP` followed by the operation, the term or the type can be entered as text input in the running program.

Appendix C: Supported Graph Formats

Currently (v2) Graphana supports the following Graph formats:

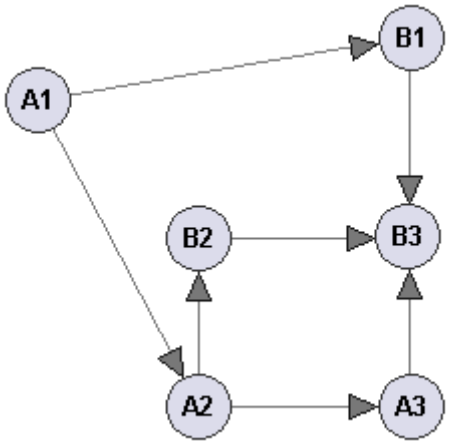
DOT

```
digraph g {
        node [shape=plaintext]
        A1 -> B1
        A2 -> B2
        A3 -> B3

        A1 -> A2 [label=f]
        A2 -> A3 [label=g]
        B2 -> B3 [label="g'"]
        B1 -> B3 [label="(g o f)'" tailport=s headport=s]

        { rank=same; A1 A2 A3 }
        { rank=same; B1 B2 B3 }
}
```

generates the following graph:



See `http://www.graphviz.org/Documentation.php` for the specification of the DOT Language. To load a DOT graph from the console do:

```
loadDOT("/path/to/file")
```

DIMACS

See `http://prolland.free.fr/works/research/dsat/dimacs.html` for the specification of the DIMACS format. In general each line starting with the edge descriptor 'e' describes an edge, e.g.:

```
e a b
```

is an edge from a to b. It's optionally possible to supply weights, thus:

```
e a b 1.0
```

is a weighted edge from a to b. To load a DIMACS graph from the console do:

```
loadDIMACS("/path/to/file")
```

METIS

The METIS Graph format is described here: `http://people.sc.fsu.edu/~jburkardt/data/metis_graph/metis_graph.html`. The first line lists the number of nodes, edges and optionally weights. Overall there are at least N+1 lines, with N being the number of nodes. The **i**-th line represents the adjacency list of the node **i**. To load a METIS graph from the console do:

```
loadMETIS("/path/to/file")
```