

Graphana - Operations and types

Contents

1	Commands	3
1.1	Program configuration	3
1.2	System operations	4
1.3	Time and date	5
1.4	Counters	5
1.5	Execution	6
1.6	System alerts	7
1.7	File output	7
1.8	File input	8
1.9	Graph visualization	8
1.10	Histogram creation	10
1.11	Histogram visualization	11
1.12	Colors	12
1.13	User interactions	13
1.14	Variables	13
1.15	Assertions	13
1.16	Bounds	14
1.17	Converting primitives	14
1.18	String operations	15
1.19	Complex type operations	15
1.20	Math functions	16
2	Graph operations	18
2.1	Graph creation	18
2.2	General graph properties	19
2.3	Graph loading	19
2.4	Graph libraries	20
2.5	Random graphs	21
2.6	Graph editing	21
2.7	Graph conversions	24
3	Algorithms	26
3.1	Vertex degrees	26
3.2	Flows	28
3.3	Connected Components	28
3.4	Trees	29
3.5	Treewidth	30
3.6	Connectivity	31
3.7	Clusters	32
3.8	Cluster vertex deletion	32
3.9	Cluster Editing	34
3.10	Feedback Sets	35
3.11	Miscellaneous graph parameters	36
4	Types	39
4.1	Primitive types	39

4.2	Graph types	40
4.3	Complex types	41
4.4	File types	42
4.5	Miscellaneous types	42

1 Commands

Every box in this section depicts one operation. The boxes are structured as follows:

operationKey
parameterName1: ParameterType1 parameterName2: ParameterType2 ... parameterNameN: ParameterTypeN [...] returns Return Type
The operation's description text.

Default values are denoted with a = after the parameter type followed by the value. If a parameter has a default value then passing an argument is optional. Some operations have three dots at the end of their parameter lists. These operations can receive arguments of the type of the last parameter in the list at any number.

Besides operations, some subsections contain descriptions of terms. These are not written in boxes.

1.1 Program configuration

getCurrentGraph
deepCopy: Boolean = false returns Graph
Returns the current graph. The returned Graph can be stored in a variable for example.

setCurrentGraph
graph: Graph returns void
Sets the given graph as the current graph.

setAlgorithmTimeout
timeOutMillis: PositiveInteger returns void
Sets the maximum computation time for an algorithm. If an algorithm which is executed afterwards exceeds the given time then the computation will be aborted and a timeout error will be returned. If 0 is given, then the timeout is disabled. The timeout is given in milliseconds, so a timeout of 1000 means one second. Initially, the timeout is set to 10000.

setScriptTimeout
timeOutMillis: <code>PositiveInteger</code> returns void
Sets the maximum script execution time. If a script which is executed afterwards exceeds the given time then the execution will be aborted and a timeout error will be returned. If 0 is given, then the timeout is disabled. The timeout is given in milliseconds, so a timeout of 1000 means one second. Initially, the timeout is disabled.

setTimeout
timeOutMillis: <code>PositiveInteger = 10000</code> returns void
Sets the algorithm timeout and the script timeout to the given value (see setAlgorithmTimeout and setScriptTimeout). The value is given in milliseconds.

setPrintWarnings
printWarnings: <code>Boolean = true</code> returns void
Calling this method enables or disables the output of warnings.

setCaching
enableCaching: <code>Boolean</code> returns void
This operation can be used to enable or disable caching of algorithm results. Some algorithms save their (interim) results to reuse them when called repeatedly or to provide them to other algorithms to increase the overall program performance. Initially, caching is enabled. There are circumstances under which caching is automatically disabled, for example if the runtime of an algorithm is measured.

1.2 System operations

import
Class: <code>ExistingFile</code> returns String
Imports the given ExistingFile into the program. The file must be a java class which is compatible with <i>Graphana</i> . After importing, the operations that are defined within the class are available in the program.

sleep
milliseconds: <code>PositiveInteger</code> returns void
Causes the program to sleep. The duration is given in milliseconds, so for example 1000 means one second.

1.3 Time and date

getTime
format: String = 'HH:mm:ss'
returns String
Returns the current system time as formatted String in the given format.

millisToString
milliseconds: PositiveInteger
returns String
Converts the given milliseconds into a formatted String .

getTimeMillis
returns Integer
Returns the current system time in milliseconds where 0 is 00:00.

getDate
format: String = 'yyyy/MM/dd'
returns String
Returns the current system date as formatted String in the given format.

1.4 Counters

startCounter
returns void
Starts the global counter. Every time this operation is called, the global counter will be reset.

getCounter
returns Integer
Returns the time difference between the call of startCounter and the current time in milliseconds. This operation does not stop the counter.

Algorithm timer:

The algorithm timer can be used to measure the runtimes of algorithms. It increases whenever an algorithm is running. So the timer is more accurate than the normal counter because only the runtime of the algorithm itself is measured, ignoring for example compatibility checks. Nevertheless, interferences with the java garbage collector may occur.

The algorithm timer is used via **startAlgorithmTimer** and **getAlgorithmTime**.

startAlgorithmTimer
returns void
Starts/restarts the algorithm timer . That means, that its value is set to 0.

getAlgorithmTime
returns Integer
Returns the current algorithm timer as Integer in milliseconds. The algorithm timer keeps running after calling this operation.

1.5 Execution

script
file: ExistingFile statements: ANY = '' ...
returns ANY
Executes the given ExistingFile as batch. The script must contain source code in <i>Graphana</i> syntax. The additional arguments are ignored and can be used to set up global variables which are used within the script for example.

executeString
statement: String
returns ANY
Executes the given String and returns the result of the execution. The given string must be source code in <i>Graphana</i> syntax. The additional arguments are ignored and can be used to set up global variables which are used within the statement for example. If the statement shall be executed multiple times it is recommended to use parse and executeTree instead of this command.

parse
source: String
returns ParseTree
Parses the given String and returns a ParseTree . The given source must be source code in <i>Graphana</i> syntax.

parseScript
script: ExistingFile
returns ParseTree
Parses the given ExistingFile and returns a ParseTree . The script must be source code in <i>Graphana</i> syntax.

executeTree
tree: ParseTree
returns ANY
Executes the given ParseTree and returns the result of the execution. The execution of a parse tree is much faster than the execution of a String with executeString .

1.6 System alerts

error
message: String returns void
Throws an error with the given text message and stops the execution of the statement and, if executed in a script, of the script.

warning
message: String returns void
Prints a warning with the given text together with some meta data.

alert
message: String title: String = 'Message' returns void
Shows a message dialogue containing the given text. The dialogue window will be titled with the given title.

1.7 File output

setOutputFile
file: File autoWriteConsoleOutput: Boolean = false autoWriteConsoleInput: Boolean = false returns void
Sets the current output file. After the output file is set, every WRITE call will write into the chosen file. If the given file does not exist, it will be created. Otherwise it will be overwritten. If autoWriteConsoleOutput is set to true then nearly every console output, including PRINT calls, errors and warnings, will be written into the file automatically. If autoWriteConsoleInput is set to true then also console inputs will be written into the file.

flushOutput
returns void
Flushes the current output file without closing it. So the file will be visible and up to date in the file system.

closeOutput
returns void
Closes the current output file. So the file will be visible and up to date in the file system and can be used by other programs. After closing the file it is not allowed to call WRITE until a new output file is set using setOutputFile .

writeWholeFile
file: File object: ANY returns void
Creates a text file which contains the string representation of the given Object . The file will be automatically closed after writing.

1.8 File input

readWholeFile
file: ExistingFile returns String
Reads the whole given (text) file and returns the content as one String .

getFiles
directory: String acceptedExtensions: Vector =) returns Set
Returns all files of the given directory as a set of File . Instead of a directory, a filename can be given alternatively. In this case, a set, which only contains the given file, will be returned.

1.9 Graph visualization

Visualization window:

Every graph visualization and **algorithm visualization** is done in a visualization window which can be minimized, maximized and closed. Within the window, the following actions can be performed:

Left click on a vertex: moving vertex.

Right click on empty space: adding a vertex.

Middle click and drag: scrolling through the view.

Mouse wheel: zoom in and out.

Right click and hold on a vertex and release on another vertex: creating an edge from the first vertex to the second or delete the respective edge, if it already exists.

Modifying the graph only works in the standard visualization and only if it is allowed (for example it is not possible in algorithm visualizations). So the right mouse button may have no effect.

Every window has a certain frame rate which determines, how often the graph is repainted per second. Repainting is necessary to make changes in the dates and states of the vertices and edges visible. If the frame rate is set to zero, then the graph must be repainted manually using **repaintGraph**.

Layout:

The layout determines how the vertices are positioned in a **visualization window**. The layout

is chosen for example as the first argument of the **showGraph** operation.

The following layouts are available in *Graphana*:

GRID
CYCLE
TREE
JUNG.CYCLE
JUNG.ISOM

For directed graphs, a root vertex must be given for every connected component when using the TREE layout. These are specified by writing a colon and the vertex identifiers separated by commas (for example TREE:v1,v5).

showGraph
<pre>layout: String = 'GRID' windowTitle: String = '' width: PositiveInteger = 640 height: PositiveInteger = 640 enableModification: Boolean = true frameRate: PositiveInteger = 10 returns void</pre>
<p>Visualizes the graph in the visualization window with the given title. If no such window exists, a new one will be created. The layout is set by passing the respective layout keyword (e.g. "Jung.ISOM","TREE" ...).</p> <p>With width and height the dimensions of the window can be set.</p> <p>If allowModification is set to false then the graph cannot be modified within the visualization window, so right click will not have any effect.</p> <p>The parameter frameRate sets the frame rate of the visualization window. If zero is given then the window will not update frequently.</p>

repaintGraph
<pre>windowTitle: String = 'Graph' returns void</pre>
<p>Refreshes the graph visualization in the visualization window with the given title.</p>

closeGraphView
<pre>windowTitle: String = 'Graph' returns void</pre>
<p>Closes the graph visualization window with the given title.</p>

Algorithm visualization:

Some algorithms support algorithm visualization, which is a step-by-step algorithm output. If algorithm visualization is enabled and a respective algorithm is executed, the visualization starts automatically. Depending on the algorithm, the graph or multiple graphs are visualized in one or more **visualization window(s)** after every important step of the algorithm. One can iterate through the steps by pressing enter in the console. To abort the visualization, 'fin' can be typed in.

The algorithm visualization blocks **caching** and the **algorithm timer**. Algorithm visualiza-

tion can be enabled or disabled using `setAlgorithmOutput`. Initially, algorithm visualization is disabled.

setAlgorithmVisualization
<code>showOutput: Boolean</code> <code>returns void</code>
This operation enables or disables algorithm visualization .

setAlgorithmVisualizationParams
<code>layout: String = 'GRID'</code> <code>width: PositiveInteger = 640</code> <code>height: PositiveInteger = 640</code> <code>returns void</code>
Sets the visualization parameters for the algorithm visualization , which can be enabled using <code>setAlgorithmOutput</code> . The parameters have the same meaning as the respective parameters in the <code>showGraph</code> operation.

1.10 Histogram creation

newHistogram
<code>estimatedValues: PositiveInteger = 64</code> <code>xScale: PositiveInteger = 1</code> <code>returns Histogram</code>
Creates a new empty Histogram . The returned Histogram can be filled with values using <code>setHistogramValue</code> or <code>incHistogramValue</code> . Initially, a value is zero. With the parameter <code>estimatedValues</code> the initial memory allocation can be set. The capacity is nearly unlimited - <code>estimatedValues</code> only has a slight effect on performance.

setHistogramValue
<code>histogram: Histogram</code> <code>index: PositiveInteger</code> <code>value: Float</code> <code>returns void</code>
Sets the value associated with the given index in the given histogram.

incHistogramValue
<code>histogram: Histogram</code> <code>index: Float</code> <code>incValue: Float = 1.0f</code> <code>returns void</code>
Increments the value associated with the given index in the given Histogram by the given <code>incValue</code> , which may be negative, too.

csvToHistogram

```
csv: String
separator: String = ':'
returns Histogram
```

Converts a CSV string into a **Histogram** which then can be used for example for visualization.

1.11 Histogram visualization

showHistogram

```
histogram: Histogram
titleKey: String = 'Histogram'
clearPrevious: Boolean = true
width: Integer = 800
height: Integer = 600
returns void
```

Visualizes the given **Histogram** using the window with the given title. If no such Window is shown, a new one will be created. If `clearPrevious` is set to `false` then previously shown histograms of the window won't be deleted. The dimension of the output window can be set by the parameters `width` and `height`.

addHistogramToView

```
histogram: Histogram
titleKey: String = 'Histogram'
returns void
```

Does the same as `showHistogram` with `ClearPrevious` set to `false`.

setHistogramViewMode

```
linesMode: Boolean
thickLines: Boolean = true
logScale: Boolean = false
beginAtZero: Boolean = true
returns void
```

Configures histogram visualization in general. This will influence every histogram visualization which is done after this operation.

If `linesMode` is set to `true` then lines will be drawn instead of bars. With `boldLines` set to `true` the lines have a width of 3px instead of 1px.

refreshHistogramView

```
titleKey: String = 'Histogram'
returns void
```

Refreshes the visualization of histograms associated with the given title. This operation must be called after changing **Histogram** values to make the changes visible to the user.

setHistogramViewColors
<pre>color: Color ... returns void</pre>
<p>Configures the colors of the bars or lines of all histogram visualization which is called after this operation. The first given Color is used for the first added Histogram of a visualization, the second Color for the second one and so on. If there are more histograms to output than colors given then it restarts with the first Color.</p>

clearHistogramView
<pre>titleKey: String = 'Histogram' returns void</pre>
<p>Removes all the histograms of a visualization associated with the given title. The visualization window remains visible.</p>

getHistogramFromView
<pre>titleKey: String = 'Histogram' index: PositiveInteger = 0 returns Histogram</pre>
<p>Extracts the Histogram with the given index from a visualization associated with the given title. The indices of the histograms are set by the order they were added into the visualization.</p>

1.12 Colors

newColor
<pre>red: PositiveInteger green: PositiveInteger blue: PositiveInteger alpha: PositiveInteger = 255 returns Color</pre>

newFloatColor
<pre>red: Float green: Float Blue: Float Alpha: Float = 1.0f returns Color</pre>

gray
<pre>value: PositiveInteger returns Color</pre>
<p>Returns a gray color with the given brightness. The brightness must be a value between 0 (black) and 255 (white)</p>

fGray
value: Float returns Color
Returns a gray color with the given brightness. The brightness must be a value between 0 (black) and 1 (white)

1.13 User interactions

ask
question: String = '' returns String
Pauses the execution, waits for a user input and returns the String which was entered by the user.

pause
message: String = 'Press Enter...' returns void
Pauses the execution until the user presses enter. A message can be given. This message will be printed before the execution pauses.

1.14 Variables

typeof
variable: ANY returns String
Returns the type name of the given value.

removeVariable
identifier: String returns Boolean
Removes the variable with the given identifier . The value will be deleted from memory and calling defined on the variable afterwards will return false .

1.15 Assertions

assert
condition: Boolean message: ANY = '' returns void
Does nothing, if the given Boolean is true. Otherwise, an error is thrown together with a message that can be given.

assertEq
value1: ANY value2: ANY message: ANY = '' returns void
Does nothing, if the two given values are equal. Otherwise, an error is thrown together with a message that can be given.

1.16 Bounds

newInterval
lowerBound: Float upperBound: Float returns Interval
Creates and returns a new Interval with the given bounds.

getLowerBound
bounds: Interval returns Float
Returns the lower bound of the given Interval .

getUpperBound
bounds: Interval returns Float
Returns the upper bound of the given Interval .

1.17 Converting primitives

asFloat
integer: Integer returns Float
Converts an Integer into a Float . This can be used for example to enforce float division when dividing two integers.

parseInt
string: String returns Integer
Converts a String into an Integer by parsing the string.

parseFloat
string: String returns Float
Converts a String into a Float by parsing the string.

parseBool
string: String returns Boolean
Converts a String into a Boolean by parsing the string. The strings "true" and "1" result in true and the strings "false" and "0" result in false. The strings are not case-sensitive.

1.18 String operations

toString
object: ANY returns String
Returns the String representation of the given object which can be of any type.

split
string: String regex: String = '\n' trim: Boolean = true returns Vector
Splits the given String at the given regular expression and returns multiple strings as a Vector .

startsWith
string: String prefix: String returns Boolean
Returns true iff the given String starts with prefix.

endsWith
string: String postfix: String returns Boolean
Returns true iff the given String ends with postfix.

1.19 Complex type operations

getSize
iterable: Iterable returns PositiveInteger
Returns the number of elements in the given Iterable .

getVectorSize
vector: Vector returns PositiveInteger
Returns the number of entries of the given Vector .

setVectorSize
vector: Vector newSize: PositiveInteger returns void
Sets the number of entries of the given Vector to the given number. The values of the vector remain the same.

getSetCardinality
set: Set returns PositiveInteger
Returns the cardinality of the given Set .

setInsert
set: Set value: ANY returns void
Inserts the given element into the given Set . The element is inserted even if an equal element exists in the given set.

1.20 Math functions

round
number: Float returns Integer
Converts a Float into an Integer by rounding the given value.

random
lowerBound: Integer upperBound: Integer returns Integer
Returns a random Integer which is bigger or equal to lowerBound and smaller or equal to upperBound .

sqrt
x: Float returns Float
Returns the square root of the given value.

sqr
x: Float returns Float
Returns the square of the given value.

pow
base: Float exp: Float returns Float
Returns base to the power of exp .

sin

x: Float

returns Float

Returns the sine of the given value.

cos

x: Float

returns Float

Returns the cosine of the given value.

tan

x: Float

returns Float

Returns the tangent of the given value.

cotan

x: Float

returns Float

Returns the cotangent of the given value.

2 Graph operations

In the explanations of this section, G is the given graph, V its vertices and E its edges. When a runtime is given then n is the number of vertices, m the number of edges and $\Delta(G)$ is the sum of both.

Every box in this section depicts one algorithm. The boxes are structured as follows:

algorithmKey
parameterName1: ParameterType1 parameterName2: ParameterType2 ... parameterNameN: ParameterTypeN [...] returns Return Type
The algorithm's description text. For some algorithms: Runtime: The algorithms runtime in O -notation Graph preconditions: List of preconditions Compatible libraries: List of supported graph libraries

Arguments are handled in the same way as they were explained in the previous section. The **algorithm timer** only counts if one of the algorithms of this section is called.

Algorithms which support **algorithm visualization** are marked with a * after the algorithm key.

2.1 Graph creation

Graph configuration:

In *Graphana* a graph configuration is the combination of the properties *directed*, *weighted* and *loop-free forced*. If a graph is forced to be loop-free then no loops can be inserted.

createGraph
directed: Boolean = false weighted: Boolean = false forceLoopFree: Boolean = false library: String = 'KEEP' returns void
Creates a graph with the given graph configuration and sets it as the current graph. With the parameter library a name of a graph library can be given. The graph will then internally be created as a graph of the respective library. If the argument is set to KEEP or omitted then the previously used library will be used. An already created graph will be completely deleted and recreated.

2.2 General graph properties

vertexCount
returns Integer
Returns the number of vertices.

edgeCount
returns Integer
Returns the number of edges.

graphSize
returns Integer
Returns the sum of the vertex count and edge count.

isDirected
returns Boolean
Returns true , if and only if the graph is directed.

isWeighted
returns Boolean
Returns true , if and only if the graph is weighted.

isLoopFree
returns Boolean
Returns true , if and only if the graph does not contain any loops.

isForceLoopFree
returns Boolean
Returns true , if no loops can be added to the graph.

2.3 Graph loading

loadGraph
filename: ExistingFile returns void
Sets the current graph by loading a DIMACS or a dot file. Depending on the given file format, the operation does either the same as loadDIMACS or loadDot .

loadDIMACS
filename: ExistingFile returns void
Loads the given DIMACS File . If the graph is directed then every edge in the file is seen as an directed edge and vice versa. So if the graph is undirected, there can only be one edge per vertex pair even if there are two in the file. If the graph is unweighted then the weights within the file will be ignored. If the graph is forced to be loop-free then loops in the file will be ignored . For huge files the number of lines to read can be limited using 'MaxLines'.

loadDot
<pre>filename: ExistingFile ignoreWeights: Boolean = false ignoreLayoutAttributes: Boolean = false returns void</pre>
<p>Loads the given dot File. Since a dot file directly contains information of whether the graph is directed or not, the resulting graph will be directed if and only if it is directed in the dot file. If the graph is unweighted then the weights within the file will be ignored. If the graph is forced to be loop-free then loops in the file will be ignored .</p>

2.4 Graph libraries

Graph library:

Graphana can internally use different graph libraries. Which library is used influences performance and the set of available algorithms. The usage itself does not depend on the chosen library. So graph construction, graph loading etc. always works in the same way. In addition libraries can be converted into each other (either manually by calling **setLibrary** or automatically if a called algorithm is not compatible with the current library).

setLibrary
<pre>libraryName: String returns void</pre>
<p>Sets the current graph library. The graph will be converted into the given graph library. Initially, the JUNG2 library is set.</p>

getLibrary
<pre>returns String</pre>
<p>Returns the name of the current graph library as a String.</p>

getAvailableLibraries
<pre>returns Set</pre>
<p>Returns the names of all available graph libraries as a Set of String. Each of the given names is a valid library input for the setLibrary operation or the createGraph operation.</p>

2.5 Random graphs

createErdosRenyiGraph

vertexAmount: PositiveInteger = 10
connectionProbability: Float = 0.3f
minWeight: Integer = 1
maxWeight: Integer = 1
returns void

Creates a random Erdős-Rényi-Graph. The new graph has `VertexCount` vertices. Every vertex pair is connected with a probability of `ConnectionProbability`. The weight of every edge is a random value between `MinWeight` and `MaxWeight`. These values only have an effect if the graph was previously created as a weighted graph (see `createGraph`).

createRandomClusterGraph

clusterAmount: PositiveInteger = 10
minClusterSize: PositiveInteger = 10
maxClusterSize: PositiveInteger = 20
additionalEdgesAmount: PositiveInteger = 8
minWeight: Integer = 1
maxWeight: Integer = 1
returns void

Creates a random cluster graph. The resulting graph contains up to `ClusterAmount` clusters.

addRandomClique

membershipProbability: Float
returns void

Adds a clique in the graph by adding edges using the existing vertices. Every vertex of the graph is part of the clique with a probability of `MemberShipProbability`. Setting the value to 1 means that the whole graph will be a clique or will be complete, respectively. By setting it to 0, the operation has no effect.

createPGeneratedGraph

vertexCount: Integer
a: Float = 0.5f
b: Float = 0.6f
returns void

Creates a p-generated random graph.
Note that the resulting graph is always undirected, unweighted and loop-free.

2.6 Graph editing

resolveVertexNameClashes

resolve: Boolean
returns void

If `resolve` is `true` then name clashes will be automatically resolved when adding a vertex (e.g. with `addVertex`) with an identifier which is already used by an existing vertex of the graph. Initially name clashes are not resolved.

addVertex
<pre>identifier: String = '' returns Vertex</pre>
<p>Adds a vertex with the given name to the current graph. The added vertex is then identified by the given identifier. If auto-resolving of name clashes is activated (see resolveVertexNameClashes) then underscores will be added to the given identifier until there is no vertex with the same identifier. If not and there is a name clash then no vertex will be added. If no identifier is given, a default identifier will be used (default identifiers are enumerated). The new or the already existing vertex is returned.</p>

addVertices
<pre>identifier: String ... returns void</pre>
<p>Adds multiple vertices. With every given identifier the operation adds a vertex just as addVertex does</p>

addVertexRow
<pre>amount: PositiveInteger startIndex: Integer = 0 prefix: String = 'v' Cluster: Boolean = false returns void</pre>
<p>Adds overall amount vertices. The operation enumerates the added vertices, starting at startIndex. The name of an added vertex will be the prefix concatenated with the number. If cluster is set to true then all added vertices are connected with each other.</p>

addEdge
<pre>startVertex: Vertex endVertex: Vertex weight: Float = 1.0f returns void</pre>
<p>Adds an edge between the two given vertices (see Vertex). A weight can be given, but will be ignored, if the graph is unweighted.</p>

setEdgeWeight
<pre>edge: Edge weight: Float = 1.0f returns void</pre>
<p>Sets the weight of the given Edge. An error is returned if the graph is unweighted.</p>

removeVertex
<pre>vertex: Vertex ... returns void</pre>
<p>Removes the given Vertex or the given vertices, respectively, from the graph. That means one ore more vertices can be given.</p>

removeVertexSet
vertices: Iterable ...
returns void
Removes all vertices of the given Iterable .
removeEdge
edge: Edge ...
returns void
Removes the given Edge or the given edges, respectively, from the graph. That means one or more edges can be given.
removeEdgeSet
edges: Iterable ...
returns void
Removes all edges of the given Iterable .
clearGraph
returns void
Removes all vertices from the graph.
deleteLoops
returns void
Deletes all loops from the graph in order that the graph is loop-free after this operation. However, loops can be inserted afterwards. To disallow this, see forceLoopFree .
forceLoopFree
returns void
Deletes all loops from the graph. Furthermore, loops cannot be inserted afterwards.
allowLoops
returns void
After the call of this operation, loops can be added into the graph.
mergeGraph
sourceGraph: Graph
returns void
Merges the graph with the given sourceGraph . Every vertex and edge of the given graph will be added to the graph as deep copies. Only dates of the vertices and edges, if existing, are not copied deep.

graphGUI
deleteGraph: Boolean = false drawWindowWidth: PositiveInteger = 640 drawWindowHeight: PositiveInteger = 640 frameRate: PositiveInteger = 0 returns void
Opens a visualization window with the standard grid layout with the purpose of editing the graph visually. If <code>deleteGraph</code> is set to <code>true</code> then all vertices are deleted before editing and the graph as well as the visualization window is empty.

2.7 Graph conversions

setGraphConfig
directed: Boolean weighted: Boolean forceLoopFree: Boolean returns void
Converts the current graph into a graph with the given graph configuration whereas the graph library remains the same. If the given graph configuration is forbidden in the respective graph library then an error will be returned.

asDirected
returns Graph
Returns an equivalent directed graph. The returned graph contains the same vertices as the original graph. For every undirected edge in the original graph two directed edges are created in the returned graph. If the original graph is already directed then the graph is returned without any changes.

toDirected
returns void
Converts the current graph into a directed graph. The converted graph contains the same vertices as the original graph. For every undirected edge in the original graph two directed edges are created in the converted graph. If the current graph is already directed then nothing happens.

asWeighted
returns Graph
Returns an equivalent weighted graph. The returned graph contains the same vertices as the original graph. For every unweighted edge of the original graph, an edge with the weight 1 is created in the returned graph. In the returned graph, edge weights can be set. If the original graph is already weighted then the graph is returned without any changes.

toWeighted**returns void**

Converts the current graph into a weighted graph. The converted graph contains the same vertices as the original graph. For every unweighted edge of the original graph an edge with the weight 1 is created in the converted graph. After this call, edge weights can be set in the current graph.

If the current graph is already weighted then nothing happens.

graphAsDIMACS**returns String**

Returns a **String** containing the DIMACS representation of the graph.

3 Algorithms

Algorithms are special Graph operations. The boxes in this section are structured the same way as in the previous section.

Algorithms which support **algorithm visualization** are marked with a * after the algorithm key.

3.1 Vertex degrees

averageDegree
returns Float
Returns the average degree of all vertices.
Graph preconditions: not empty

maxDegree
returns Integer
If the graph is undirected then the degree of the vertices with the largest number of neighbors is returned. Otherwise the maximum of maxIngoingDegree and maxOutgoingDegree is returned.
Graph preconditions: not empty

maxIngoingDegree
returns Integer
Returns the ingoing edge count of the vertices with the largest number of ingoing edges. If the graph is undirected then the returned value is equal to the maxDegree return value.
Graph preconditions: not empty

maxOutgoingDegree
returns Integer
Returns the outgoing edge count of the vertices with the largest number of outgoing edges. If the graph is undirected then the returned value is equal to the maxDegree return value.
Graph preconditions: not empty

minDegree
returns Integer
If the graph is undirected then the degree of the vertices with the smallest number of neighbors is returned. Otherwise the minimum of minIngoingDegree and minOutgoingDegree is returned.
Graph preconditions: not empty

minIngoingDegree
returns Integer
Returns the ingoing edge count of the vertices with the smallest number of ingoing edges. If the graph is undirected then the returned value is equal to the minDegree return value.
Graph preconditions: not empty

minOutgoingDegree
returns Integer
Returns the outgoing edge count of the vertices with the smallest number of outgoing edges. If the graph is undirected then the returned value is equal to the minDegree return value.
Graph preconditions: not empty

degreeDistribution
returns Histogram
Returns a Histogram with a mapping from vertex degrees to the amount of vertices that have the respective degree.
Graph preconditions: undirected, not empty

ingoingDegreeDistribution
returns Histogram
Returns a Histogram with a mapping from vertex degrees to the amount of vertices that have the respective ingoing degree. If the graph is undirected then the returned histogram is equal to the degreeDistribution return value.
Graph preconditions: not empty

outgoingDegreeDistribution
returns Histogram
Returns a Histogram with a mapping from vertex degrees to the amount of vertices that have the respective outgoing degree. If the graph is undirected then the returned histogram is equal to the degreeDistribution return value.
Graph preconditions: not empty

distanceDistribution
returns Histogram
Returns a mapping of d to the number of vertices that have the distance d .
Graph preconditions: not empty
Compatible libraries: JUNG2

3.2 Flows

maxFlow
source: Vertex sink: Vertex returns Float
Returns the max flow between the two given vertices (see Vertex).
Graph preconditions: not empty Compatible libraries: JUNG2, JGraphT

minCut
source: Vertex sink: Vertex returns Vector
Returns the min cut between the two given vertices (see Vertex).
Graph preconditions: not empty Compatible libraries: JUNG2

Gomory-Hu-Tree:

The *Gomory-Hu-Tree* $T = (V, E_T)$ of a graph $G = (V, E_G)$ is a tree in order that every pair $(v, w) \in V$ has the same max flow as in G .

gomoryHuTree *
ignoreWeights: Boolean = false returns Graph
Returns the Gomory-Hu-tree of the graph.
Runtime: $O(n^2 + m^2)$ Graph preconditions: undirected, not empty, loop-free Compatible libraries: JUNG2

3.3 Connected Components

getConnectedComponentCount *
returns PositiveInteger
Returns the number of connected components.

getConnectedComponent *
componentIndex: PositiveInteger returns Graph
Returns the connected component with the given index. The index must be a value between 0 and the connected component count (see getConnectedComponentCount) minus one. The indices are given internally. The returned graph is a deep copy of the respective connected component.
Graph preconditions: undirected, not empty

getConnectedComponentByVertex *
vertex: String returns Graph
Returns the connected component in which the given vertex is contained. The returned graph is a deep copy of the respective connected component.
Graph preconditions: undirected, not empty

getStronglyConnectedComponentCount
returns Integer
Returns the number of strongly connected components.
Graph preconditions: loop-free

3.4 Trees

Tree:

A *tree* is an acyclic graph.

isTree *
returns Boolean
Checks, whether the graph is a tree .
Graph preconditions: undirected

3.5 Treewidth

setTreewidthUpperBoundHeuristics

heuristic: String

...

returns void

Since the computation of the treewidth is NP-complete, *Graphana* uses some heuristics for this problem. The heuristics are implemented in LibTW from www.treewidth.com.

The heuristics which are to be used are passed as **Strings**. If multiple heuristics are given then every heuristic will be executed and the best result will be returned.

The following **Strings** are valid treewidth upper bound heuristic keys:

GREEDYFILLIN

GREEDYDEGREE

ALLSTARTLEXBFS

By default, GREEDYFILLIN is set. For informations on the different heuristics, see www.treewidth.com.

The chosen treewidth upper bound heuristics influence the following algorithms: **treewidthUpperBound**, **treewidthBounds**, **treewidthExact**.

setTreewidthLowerBoundHeuristics

heuristic: String

...

returns void

Since the computation of the treewidth is NP-complete, *Graphana* uses some heuristics for this problem. The heuristics are implemented in LibTW from www.treewidth.com.

The heuristics which are to be used are passed as **Strings**. If multiple heuristics are given then every heuristic will be executed and the best result will be returned.

The following **Strings** are valid treewidth lower bound heuristic keys:

MAXMINDEGREEPLUSLEASTC

MAXCARDSEARCH

RAMACHANDRAMURTHI

ALLSTARTMAXCARDSEARCH

MAXMINDEGREE

MAXMINDEGREEPLUSMAXD

MAXMINDEGREEPLUSMIND

ALLSTARTMAXMINDEGREE

ALLSTARTMAXMINDEGREEPLUSLEASTC

ALLSTARTMINORMINWIDTH

MINORMINWIDTH

MINDEGREE

By default, MAXMINDEGREEPLUSLEASTC is set. For informations on the different heuristics, see www.treewidth.com.

The chosen treewidth lower bound heuristics influence the following algorithms: **treewidthLowerBound**, **treewidthBounds**.

treewidthUpperBound
returns Integer
Returns an upper bound of the treewidth. The heuristics which are to be used can be set with setTreewidthUpperBoundHeuristics .
Graph preconditions: not empty Compatible libraries: LibTW

treewidthLowerBound
returns Integer
Returns a lower bound of the treewidth. The heuristics which are to be used can be set with setTreewidthLowerBoundHeuristics .
Graph preconditions: not empty Compatible libraries: LibTW

treewidthExact
returns Integer
Returns the treewidth using the 'TreewidthDP' algorithm from www.treewidth.com . The algorithm has a NP runtime. Before the actual computation starts, an upper bound is established by using one or more heuristics. Which heuristics are to be used for this can be set with setTreewidthUpperBoundHeuristics . For further information on the algorithm, see www.treewidth.com .
Graph preconditions: not empty Compatible libraries: LibTW

3.6 Connectivity

largestKConnected *
k: Integer returns Integer
Returns the cardinality of a maximum $V' \subset V$ in order that V' is k-edge-connected depending on the parameter k.
Graph preconditions: undirected, not empty, loop-free Compatible libraries: JUNG2

edgeConnectivityDistribution *
returns Histogram
Returns a mapping from k to largestKConnected (k) as a Histogram . The first value is $k = 0$. The last value is the largest k where largestKConnected (k) does not return 0.
Graph preconditions: not empty Compatible libraries: JUNG2

3.7 Clusters

Cluster:

A *cluster* is a connected component in which all vertices are connected with each other.

Cluster graph:

A *cluster graph* is a graph which consists only of clusters (see **Cluster**).

isClusterGraph *
returns Boolean
Returns true if and only if the graph is a Cluster graph .
Graph preconditions: undirected

3.8 Cluster vertex deletion

CVD:

Abbreviation for "cluster vertex deletion"

Cluster vertex deletion set:

A set $C \subseteq V$ in order that $(V \setminus C, E_C)$ is a **cluster graph** (the set of edges $E_C \subseteq E$ contains all edges which are not incident to any vertex in C).

CVD-set:

Abbreviation for **Cluster vertex deletion set**

CVD-heuristics:

Since finding a **CVD-set** is NP-complete, *Graphana* supports several heuristics for this problem which differ in runtime and cardinality of the resulting set.

Which heuristic(s) shall be used, can be set with **setCVDHeuristics**. A CVD-heuristic consists of two parts: the search strategy to search for nodes which may be deleted and the delete strategy to delete one or more vertex of the found candidates. In *Graphana* there are two search strategies and three delete strategies. So in combination, there are six heuristics.

The search strategies are:

Successive (key: "SUCC"):

The candidates are found by regarding each vertexes neighbors. This strategy is recommended for very sparse graphs.

Runtime: $O(n \cdot m)$ Connected components (key: "CC"):

The candidates are found by recursively splitting the graph into connected components. This strategy is especially recommended for dense graphs. In most cases the runtime is better than the runtime of the "Successive" strategy.

Runtime: $O((n + m) + |C| \cdot (n + m) \cdot \Delta(G) + |C| \cdot (\Delta(G))^2)$

Where $|C|$ is the cardinality of the CVD set.

The delete strategies are:

All (key: "ALL"): Deletes all found candidates. This strategy ensures, that the cardinality of the resulting CVD-set is not more than three times as large as the cardinality of an optimal solution.

First (key: "FIRST"):

Deletes the candidate which was found first.

Maximum degree (key: "MAX")

Deletes a candidate with the highest **degree**.

setCvdHeuristics

heuristic: String

...

returns void

Sets the **CVD-heuristics** which shall be used when computing a **CVD-set**. The heuristics are given as **Strings** containing the key of search strategy and the key of the deletion strategy, separated by a minus character. So for example "CC-MAX" is a valid string. More than one heuristic can be set by passing them within one call. A new call of setCVDHeuristics resets the heuristics. If multiple heuristics are given, then the respective algorithms will execute all of them and return the best result. So the computation time increases but the results are getting more accurate.

The chosen cvd heuristics influence the following algorithms: **cvdSet**, **cvdSize**, **cvdBounds**, **toClusterGraph**, **maximumIndependentSetByCVD**

toClusterGraph

returns void

Deletes vertices in order that the graph becomes a **cluster graph**. The number of deleted vertices may not be optimal (see **CVD-heuristics**).

Graph preconditions: undirected, not empty, loop-free

getMaximumIndependentSetByCVD

returns List

Computes the maximum independent set with a parameterized algorithm which uses a **CVD-set** as parameter.

Graph preconditions: undirected, not empty, loop-free

getCvdSet

returns List

Tries different heuristics to compute a **CVD-set**.

Graph preconditions: undirected, not empty, loop-free

getCvdSetSize
returns Integer
Same as getCVDSet but only returns the size of the solution.
Graph preconditions: undirected, not empty, loop-free

3.9 Cluster Editing

CE:

Abbreviation for "cluster editing". Task: Find a minimum size set of edges to add or delete such that the result is a **cluster graph**.

edge deletion:

The minimum size set of edges to delete such that the result is a **cluster graph**.

clusterEditing
returns List
Returns a minimum size set of edges to add or delete such that the result is a cluster graph . The edge branching algorithm in use has a runtime of
Runtime: $O(n^{2.61})$
Graph preconditions: undirected, not empty

clusterEditingSize
returns Integer
Returns the solution size, see clusterEditing .
Graph preconditions: undirected, not empty

clusterEditing3APX
returns List
It is an expected (randomized) 3-approximation for cluster editing. The result is the best of 5 runs.
Graph preconditions: undirected, not empty

clusterEditing3APXSize
returns Integer
Returns the solution size, see clusterEditing3APX .
Graph preconditions: undirected, not empty

clusterEditing3APXRuns
runs: Integer returns List
Same as clusterEditing3APX ; takes the number of runs as its argument.
Graph preconditions: undirected, not empty

clusterEditing3APXRunsSize
runs: Integer returns Integer
Returns the solution size, see clusterEditing3APXRuns .
Graph preconditions: undirected, not empty

edgeDeletionAPX
returns List
A variaton of clusterEditing3APX for edge deletion . The result is the best of 5 runs.
Graph preconditions: undirected, not empty

edgeDeletionAPXSize
returns Integer
Returns the solution size, see edgeDeletionAPX .
Graph preconditions: undirected, not empty

edgeDeletionAPXRuns
runs: Integer returns List
Same as edgeDeletionAPX ; takes the number of runs as its argument.
Graph preconditions: undirected, not empty

edgeDeletionAPXRunsSize
runs: Integer returns Integer
Returns the solution size, see edgeDeletionAPXRuns .
Graph preconditions: undirected, not empty

3.10 Feedback Sets

Feedback edge set size:

The *feedback edge set size* is the minimum number of edge deletions that are necessary in order that the graph becomes a **tree**.

The feedback edge set size for a connected component is $|E| + 1 - |V|$.

feedbackEdgeSetSize *
returns Integer
Returns the feedback edge set size .
Graph preconditions: undirected

feedbackVertexSet *
returns Set
Graph preconditions: undirected, loop-free

feedbackVertexSetSize *
returns PositiveInteger
Graph preconditions: undirected, loop-free

feedbackArcSet
returns List
Heuristic for computing the <i>feedback arc set</i> of a directed graph; that is the number of edges that have to be removed in order to obtain an acyclic graph. Uses sifting, which is similar to a two sided insertion sort.
Runtime: $O(n^2)$.
Graph preconditions: loop-free

feedbackArcSetSize
returns Integer
A call to this algorithm is equivalent to <code>getSize(feedbackArcSet())</code> .
Graph preconditions: loop-free

3.11 Miscellaneous graph parameters

vertex cover:

A *vertex cover* S is a set of vertices in order that every edge $e \in E$ has at least one endpoint in V . The vertex cover size is the cardinality of a minimum vertex cover.

dominating set:

A *dominating set* S is a set of vertices such that every node in the graph has at least one neighbour in S .

betweenness:

The *betweenness* of a Vertex/Edge is the number of shortest paths that use the Vertex/Edge.

vertexCoverSize
useGreedy: Boolean = true
returns Integer
This algorithm offers two heuristics: If useGreedy is set to true then a $n \log n$ - approximation is used. Otherwise a 2-approximation is used. The $n \log n$ - approximation delivers better results in many practical cases.
Runtime: $O(n + m)$
Graph preconditions: undirected, loop-free

vertexCoverSizeBothHeuristics
returns Integer
Calls both heuristics of vertexCoverSize and returns the minimum of both results.
Runtime: $O(n + m)$
Graph preconditions: undirected, loop-free

hIndex
returns Integer
The <i>h-index</i> is the largest number n in order that n nodes have at least n neighbors.
Runtime: $O(n + \Delta(G))$
Graph preconditions: not empty

hIndexPlus
returns Integer
The <i>h-indexPlus</i> is the number n of nodes to delete such that the remaining nodes have a degree of at most n .
Graph preconditions: undirected, loop-free

k-degenerate:

A graph is *k-degenerate* if and only if there is an induced subgraph which contains a vertex with a degree at most k .

degeneracy
returns Integer
The <i>degeneracy</i> is the smallest number k in order that the graph is k-degenerate .
Runtime: $O(m)$
Graph preconditions: undirected, not empty, loop-free

dominatingSet
useReduction: Boolean = false
returns Set
A heuristic to calculate a (hopefully minimal) dominating set. The result set is returned. The 1-degree reduction rule is always applied. If useReduction is set to true there will be a further reduction step beforehand (
Runtime: $O(n^3)$.
Graph preconditions: undirected, loop-free

dominatingSetSize
useReduction: Boolean = false
returns Integer
Same as dominatingSet but returns only the size of the result set.
Graph preconditions: undirected, loop-free

betweennessVertex
returns Histogram
Calculates the betweenness of all vertices and returns a histogram.
Graph preconditions: not empty
Compatible libraries: JUNG2

betweennessEdge
returns Histogram
Calculates the betweenness of all edges and returns a histogram.
Graph preconditions: not empty
Compatible libraries: JUNG2

neighborhoodDiversity
returns Integer
A graph has a neighborhood diversity of at most w , if there exist a partition of V into at most w sets, such that all the vertices in each set have the same neighborhood.
Graph preconditions: not empty

4 Types

Every box in this section depicts one type. The boxes are structured as follows:

TypeName
The type's description text.
Samples: sample1 sample2 ... sampleN

4.1 Primitive types

Integer
An integral number with the range -2,147,483,648 to 2,147,483,647.
Samples: 8 -10 0

PositiveInteger
Essentially the same as Integer but with the range 0 to 2,147,483,647.
Samples: 3 0

Boolean
A truth value with two possible values.
Samples: true false

Float
A floating point number with the range 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive and negative).
Samples: 4.6 -2.0
An Integer is automatically converted into a Float , if it is necessary.

String

A string of characters. Constant strings can either be written in quotation marks or in tick marks. When using quotation marks, tick marks can be written inside the string and vice versa without escaping.

See **Escape characters** to get a list of supported escape characters.

Samples:

```
'word'
```

```
"long text"
```

```
"A string with\n\t'tick marks' and\n\t\"quotation marks\""
```

Character

A single character.

See **Escape characters** to get a list of supported escape characters.

Samples:

```
'A'
```

```
'\n'
```

Escape characters:

Within constant **Strings** or **Characters**, the following expressions are valid escape characters:

```
\n    Line break  
\t    Tab  
\    Backslash  
\"    Quotation marks  
'    Tick mark
```

4.2 Graph types

Graph

A graph including vertices, edges, configuration, vertex- and edge data, name and algorithm cache.

Sample: `getCurrentGraph()`

Vertex

A single vertex of a graph. A constant vertex can be written with `$(vertex identifier)`. This will deliver the vertex with the given identifier of the respective graph. If no such vertex exists, an error will occur.

Samples:

```
$(v2)
```

```
getVertexByIdent('v2')
```


Edge

A single edge of a graph. An edge can be identified by the two incident vertices (in directed graphs by their ordering, too).

An edge can for example be delivered using `vertex1 -- vertex2` in undirected and `vertex1 -> vertex2` in directed graphs.

Samples:

```
getEdge($v0,$v2)
$v0 -- $v2
```

4.3 Complex types

Vector

A vector holds multiple ordered values and can be of any size. An entry of the vector can be of any individual type (also Vector again). The particular entries can be accessed with `vector[index]` where `index` is an **Integer** beginning at 0.

Vectors can be used in foreach-loops (see 'graphana_manual.pdf').

Samples:

```
(1,2,3,4)
()
((2.4,4.2,6.4),(-5.6,7,10.2),(3,2.1,0))
```

See **Complex type operations** for a list of operations on vectors.

Set

A set holds multiple unordered values and can be of any size. An element of the set can be of any individual type (also Set again).

Sets can be used in foreach-loops (see 'graphana_manual.pdf').

Samples:

```
{1,2,3,4}
{}
{"A string",$aVertex,{2.3,8.5,-6}}
```

See **Complex type operations** for a list of operations on sets.

Iterable

An Iterable cannot be created directly. An argument of an operation call is casted into an Iterable if it is a **Vector** or a **Set**.

4.4 File types

File

Files are given as **Strings** containing the relative or absolute filename. The file does not have to exist.

Samples:

```
"C:/absolute/path/file.ext"
```

```
"relative/path/file.ext"
```

ExistingFile

Nearly similar to **File** but the file must exist.

Sample: "path/to/an/existing/file.ext"

4.5 Miscellaneous types

Histogram

A histogram contains a mapping from integral numbers to float numbers. Every entry can be accessed in particular.

Samples:

```
createHistogram(30)
```

```
degreeDistribution()
```

See **Histogram creation** for a list of operations on histograms.

Interval

An interval has a minimum and a maximum value ("bounds"). The bounds can be extracted using **getLowerBound** and **getUpperBound**

Samples:

```
newInterval(-3,7)
```

```
cvdBounds()
```

ParseTree

A **String** can be parsed and converted into a parse tree. This tree can then be executed any number of times and does not need to be parsed again, which improves performance. Note that the execution of a tree may return different results depending on global variables for example. So if a script shall be executed very often, it makes sense to convert it into a parse tree once using **parse**, assign it to a variable and execute it repeatedly (for example within a loop) using **executeTree**.

Sample: `parse("2 + x*(3+y)")`