Gralog Manual

Felix Herron
felix.herron@tu-berlin.de

Roman Rabinovich
roman.rabinovich@tu-berlin.de

Adrian Alic
adrian.alic@campus.tu-berlin.de

12. November 2019

Inhaltsverzeichnis

Zusammenfassung

Gralog is a visual tool for working with graphs, logics, games, transition systems and other structures based on undirected and directed graphs. It can create, load, save and edit graphs in various formats.

The key focus of Gralog is its simplicity of use and a short time of learning how to use it.

A special property of Gralog is that it helps the developer to write programmes for graphs in any language capable of working with pipes. Gralog visualises the run of the programme and can keep track of values of user defined variables.

The interaction between Gralog and the external programme is performed by a simple, but powerful protocol. In the first version we implemented a library for Python that simplifies the interaction and abstracts away the use of pipes. This paper describes the protocol and the library, the External Programming Module (EPM). This includes documentation of methods and classes pertaining to the EPM and code examples for how to use these.

1 Documentation and Installation

You can download Gralog from https://github.com/gralog/gralog. First, make sure that Java version at least 10 is installed on your machine.

On Linux and Mac OS you can install Gralog by

```
cd install
./install-nix.sh
```

Call

```
./install-nix.sh -h
```

to get help on the installer options. If the installation script didn't work for some reason, read below how to install Gralog manually.

If option `-c` is given, the installer compiles Gralog in directory `/gralog` (which is created if necessary) or in the directory you give the script as a parameter:

```
./install-nix.sh -d /path/to/gralog
```

If run with option `-l` (i.e., *ell*), a symbolic link to the script starting Gralog is created in `/usr/bin`. For this, the installer asks the user for the administrator password.

By default (without option `-n`), the installer tries to install Python modules *networkx* and *python-igraph* with *pip*. They are needed to be able to use Gralog's Python module *Gralog.py*, which allows the user to write Python scripts working with graphs and conveniently visualise and debug them using Gralog.

If *pip* is not installed, the installer tries to install it. By default, *pip* will try to install the modules locally. (This corresponds to calling `pip install -user <module>`.) If you want a system-wide installation of those modules, call the installer with option `-s`. This will demand administrator rights.

If Python is not installed, the installation stops at this point (however, Python is installed by default on Linux machines and on Macs). If on your machine by default, *networkx* and *python-igraph* must be installed to a directory you do not have writing rights for, the installer asks for the administrator password.

On Windows Gralog is not tested yet. You can try to install it by running

```
gradlew.bat
```

in the Gralog root directory. This compiles Gralog if necessary and runs it. The *jar* file is `./build/dist/gralog-fx.jar`. Similarly, on Linux and Mac OS one can call

```
./gradlew
```

On Windows and on newer Macs it is possible to call the jar file by just clicking on it.

To use the python part on Windows, you have to install Python and the libraries *networkx* and *python-igraph* yourself. Then copy the file `./gralog-fx/src/main/java/gralog/gralogfx/piping/scripts/Gralog.py` to the directory with your Python script or somewhere where Python will find it.

The documentation can be found in directory `doc/`. The main manual file is *gralog.pdf*.

1.1 Manual installation

On Windows or if `install-nix.sh` didn't work, it should be not difficult to install Gralog manually.

The binaries can be found in `build/dist`. If you want to compile Gralog yourself, please, download the full version with sources (`gralog-all.zip`, not just (`gralog-bin.zip`) and run

```
./gradlew
```

on a *nix system and

```
./gradlew.bat
```

on Windows.

Now you can just run `build/dist/gralog-fx.jar`. (You can move the contents of `build/dist/` wherever you want, but keep it all together.) However, this still does not allow one to work with the Python plugin. To install the plugin, please, make sure you have installed Python with modules `networkx` and `python-igraph`. Then copy the file `gralog-fx/src/main/java/gralog/gralogfx/piping/scripts/Gralog.py` to some place where your Python will see it.

2 Working with the console

The console is a tool to work with graphs by entering commands rather than by clicking the mouse buttons. This is especially convenient if you want to create large subgraphs with a regular pattern.

By default, the console is the frame below the main frame with the graph. Here, in the bottom line, you can enter commands just like in a terminal.

2.1 Lists

The objects you work with are *lists:* lists of vertices and lists of edges. We concentrate on vertex lists in this manual, edge lists behave similarly. Where appropriate, we point out the differences.

The lists are used to connect them later.

Example 1. Running example For example, if one wants to create a perfect matching with 50 edges, one way to do that in Gralog is to create two disjoint lists of vertices and then connect them by drawing an edge from vertex i in the first list to vertex i in the second list. Well, for a matching we would use a generator (see Section **??**), but what if we want to add edges such that already exiting 100 vertices build a matching?

2.2 Creating lists, (de)selecting elements

All lists can be created out of the default list of all vertices and the internal list of all currently selected vertices. The latter can be made with the mouse in the obvious way or you can write in the command line:

```
select all vertices
```

to select all vertices. By the way, in a similar way, all vertices are deselected:

```
deselect all vertices
```

Now use the selection to create a new list. The corresponding command is `filter`: one filters the elements from an existing list according to certain criteria and saves them into another list. If the other list already exists, the new elements are appended to it (even if they are already contained in the target list).

```
filter all selected where no condition to VL1
```

When creating list ids as `VL1`, keep in mind that the names must be unique, in particular, it is impossible that a vertex list and an edge list have the same name.

2.3 Filter

Now we created a new list `VL1` containing all vertices of the graph. Let us now create a new list that contains only vertices with a certain property, say those whose colour is blue. We can now use `VL1` as a source of vertices.

```
filter VL1 where fill color blue to VLblue
```

Now the list `VLblue` contains all vertices with fill color `blue`. Ne create a new list containing all vertices whose fill color is red. Note in the following example that instead of the keyword `where` one can write `such that` or `st`. Note also that instead of `fill color` it is possible to use just `fill`:

```
filter VL1 st fill red to VLred
```

The full specification of the filter command is as follows. Note that Gralog does not distinguish between the upper and the lower cases.

```
FILTER <what> WHERE|ST|(SUCH THAT) <parameters> TO <list identifier> [ignored trash]
```

where

<what> := ALL VERTICES | ALL EDGES

| [ALL]? SELECTED VERTICES | [ALL]? SELECTED EDGES

| <list id>

<parameters> := <parameters> <parameters> | <boolean parameter>

| <fixed range parameter> <value>

| <numerical parameter> <cmp> <value>

<fixed range parameter> := SHAPE | TYPE | EDGE TYPE | EDGETYPE

<numerical parameter> := <integer parameter> | <float parameter> | <string parameter>

<integer parameter> := ID | DEGREE | INDEGREE | OUTDEGREE

<float parameter> := FILL | FILL COLOR | STROKE | STROKE COLOR

| THICKNESS | WIDTH | HEIGHT | SIZE | WEIGHT

<string parameter> := LABEL CONTAINS

<boolean parameter> := NO CONDITION | HAS SELFLOOP | DIRECTED | HAS LABEL

| LABEL EMPTY | NO LABEL

| BUTTERFLY // not implemented yet

In other words, we can write something like

```
FILTER sourceList ST FILL RED STROKE BLUE SHAPE ELLIPSE DEGREE > 3 TO targetList
```

Let us look at the grammar closer. In `<what>` there is no difference between `ALL SELECTED VERTICES` and `SELECTED VERTICES`, the result will be the same. The `<list id>` is the name of a list we defined previously, it can be a list of vertices or a list of edges.

The order of parameters in the parameter list is arbitrary. The list of parameters cannot be empty. If you do not want any specifications, choose `NO CONDITION`. In that case even if other parameters are given, they are not taken into account and the whole source list is given. This can be in particular used to duplicate every element in a list:

```
FILTER sourceList ST NO CONDITION TO newList
FILTER sourceList ST NO CONDITION TO newList
```

In this case `newList` will contain every element of `sourceList` twice. By the way, being in the command line, one can press ↑ (the arrow up key) to call previous commands.

Table 1 explains which values the parameters can have.

parameter	**possible values**	**applicable to**	**meaning**
SHAPE	ELLIPSE, RECTANGLE, DIAMOND	vertices	ellipse, rectangle, diamond
	SQUARE, CYCLE	vertices	not implemented yet
STROKE or STROKE COLOR (synonyms)	WHITE, BLACK, BLUE, GREEN, RED, GRAY, YELLOW, CYAN, MAGENTA, SILVER, MAROON, OLIVE, DARK_GREEN, PURPLE, TEAL, NAVY, ORANGE	both	the colour of the border; other colours are possible as properties of vertices, but not supported by the filter command
EDGE TYPE	SHARP, ROUND, BEZIER	edges	sharp or rounded corners or a Bézier curve
FILL or FILL COLOR (synonyms)	as for STROKE	vertices	the fill color, otherwise as for STROKE
WIDTH	float	vertices	???
THICKNESS	float	both?	???
HEIGHT	float	vertices	???
SIZE	float	vertices ?	???
WEIGHT	float	edges	???
ID	non-negative integer	both	the internal unique id of the vertex or of the edge
DEGREE	non-negative integer	vertices	the degree of the vertex; only undirected edges count
INDEGREE	non-negative integer	vertices	the in-degree of the vertex; only directed edges count
OUTDEGREE	non-negative integer	vertices	the out-degree of the vertex; only directed edges count
HAS SELFLOOP	no value	vertices	true if the vertex has a self-loop
NO SELFLOOP	no value	vertices	true if the vertex has no self-loop
LABEL EMPTY or NO LABEL (synonyms)	no value	vertices	true if the label is empty
LABEL CONTAINS	string	vertices	true if the label contains the given string

Tabelle 1: Parameters and their values

2.4 Sorting the lists

Before connecting lists of vertices (recall Example 1) we probably want to sort them, so that we know what vertex has number i. The lists can be sorted by

SORT <list id> <parameter>

where `parameter` is one of `LEFTRIGHT`, `RIGHTLEFT`, `TOPDOWN`, `BOTTOMUP`, `ID ASC`, `ID DECS`, `LABEL ASC`, `LABEL DESC`. The former four parameters imply a sorting according to the corresponding coordinate of vertices on the screen, `ID ASC` sorts by the id in the ascending order, `ID DESC` sorts by the id in the descending order and `LABEL` sorts alphabetically (ascending or descending) by the label. If the labels of two vertices are equal, the vertices keep their relative order.

2.5 Operations on lists (before connecting)

The following operations are currently supported:

operation	**meaning**
DELETE <list id>	deletes the list, but not the elements of the list from the graph
UNION <source1> <source2> TO <target>	creates the union of `source1` and `source2` and stores the result in `target`
INTERSECTION <list1 id> <list2 id> TO <list3 id>	creates the intersection of `source1` and `source2` and stores the result in `target`
DIFFERENCE <source1> <source2> TO <target>	creates the difference of `source1` and `source2` and stores the result in `target`
SYMMETRIC DIFFERENCE <source1> <source2> TO <target>	creates the symmetric difference of `source1` and `source2` and stores the result in `target`
COMPLEMENT <list id> TO <target>	creates the complement of `list id` and stores the result in `target`.
CONTRACT <vertex list id> | <edge list id> | SELECTED	builds a minor of the graph by contracting the subgraph induced by the elements of the list or by the selected elements (if `SELECTED` is specified and some elements are selected) to a single vertex. Works for all kinds of graphs. Not implemented yet.
BUTTERFLY CONTRACT <edge list id>	builds a butterfly minor of the graph by butterfly contracting all butterfly contractable edges. On undirected edges the operation has no effect. Not implemented yet.
SUBDIVIDE <edge list id> <n>	subdivides every edge from the list n times where n must be a natural number. Directed edges are replaced by directed paths of length $n+1$. Not implemented yet.

2.6 Connecting lists of vertices

One can either connect one list of vertices to another, which, however, can be the same list, or connect vertices of one list in a certain way. The first is done by the command

CONNECT <list1> <list2> =<formula> | BICLIQUE | MATCHING

The easier case is to use a standard way of connection, for example,

CONNECT upper lower MATCHING

which produces a matching connecting the ith vertex of `list1` to the ith vertex of `list2`

3 Setting up your External Python Program

To link your script to Gralog, create a python file and import (all of) it. This is easiest (and *recommended* if you want the provided code snippet to work) if the library and your script are in the same directory - python can be finicky about such things.

In the new file you created, paste the following code.

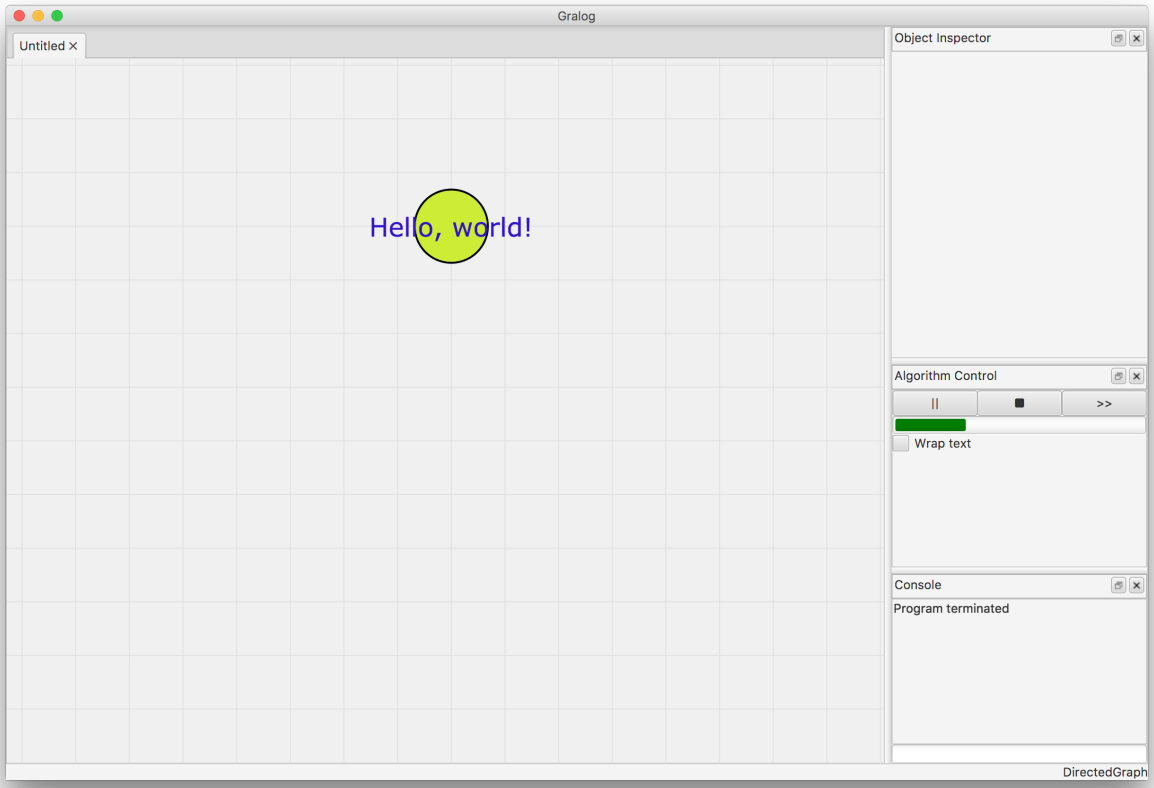
Code Snippet 1

```
#!/usr/bin/python
#HelloWorld.py
from Gralog import *
#A simple Gralog program which creates a vertex that says "Hello, world"

g = Graph(None); #uses the current graph that is open
v = g.addVertex();
v.setLabel("Hello, world!");
```

Now before you can run this code, open Gralog and select Preferences from the File menu. Navigate to General, then select the file you created at *Ext. Prog. Source File*. Click **"Ok".**

Now you should be ready to run. Navigate to File Menu and select Load Plugin". What you should get is something that looks like this:



3.1 Troubleshooting and Pro-Tips

3.2 Gralog Installation

First obviously make sure you have python installed (version ≥ 2.7). Also make sure that Gralog.py is in the directory as the file you wish to execute. Obviously you can mess around with the file structure to fit your needs but in this configuration they must be in the same directory. For other outstanding problems feel free to shoot an email at `gralog@tu-berlin.de` (TODO: make the email address.)

3.3 A Good Rule

Do not incorporate the „hashtag“ symbol into variable names or messages. The # symbol is reserved and using it may cause Gralog to misinterpret messages, which could result in spurious outcomes from your code.

4 Introduction

Now that you have the code running and set up, what will follow is a brief explanation of all of the functionality which we have built, structured in an intuitive manner.

4.1 The Graph Class

The first relevant class is Graph. In every program, you must choose a graph on which to execute your program. This is accomplished by instantiating the class Graph.

```
g = Graph();
```

In the constructor you specify which type of graph (directed, undirected, a Kripke structure, a finite automaton, a Büchi automaton) you would like. No argument means use the graph that is currently opened in Gralog, whatever (type) it may be.

The Graph class can be seen as the moderator of the program. You will use it to do all of the surface-level, more general commands pertaining to the graph itself. The more intricate details will be done using the following two:

4.2 The Vertex Class

Each vertex is represented as an object of the class Vertex. It is distinguished by its unique ID.

```
v = g.createVertex();
```

All methods pertaining to the individual vertices, such as their color, neighbours, or label, are most easily manipulated using methods of this class. For example:

```
v = g.createVertex(id=42);
v.setLabel("f00");
neighbours = v.getNeighbours();
myLabel = v.getLabel();
v.delete();
```

4.3 The Edge Class

Each edge is represented as an object of hte Edge class. It is distinguished by its unique ID; however, in graphs without multi-edges, it can also be distinguished by its source and target vertices.

```
e = g.createEdge(v1,v2,directed=False);
```

All methods pertaining to the individual edges, such as their color, adjacent edges, or label, are most easily manipulated using methods of this class. For example:

```
e = g.createEdge(v1,v2,directed=False,id=451);
e.setLabel("f00");
adjacentEdges = e.getAdjacentEdges();
target = e.getTarget();
e.delete();
```

4.4 Printing to the Console

To print to the Gralog console, please use the method gPrint(str: message). Please note that this message must be a single-line string. No new-lines please!

```
#!/usr/bin/python
#HelloWorld.py
from Gralog import *
g = Graph(None);
#end boiler plate

gPrint("hello world on the Gralog Console yeaaaahhh");
```

5 Global Functions

gPrint(String: message) *returns* `void`

Passes the message to Gralog to display in the console. Make sure this is in fact a string, or it will not work. Also please avoid new-lines/multiline messages, as they will not work.

Also: **PLEASE PLEASE PLEASE** do **not** use just **normal print** - this will crash the program immediately and you will be sad, and we will not be sympathetic because we explicitly warned that this may happen. :'(

6 Class Vertex - Documentation

6.1 Instance Variables

There are instance variables - however, they are not meant to be directly accessed by the user and are thus not detailed here.

Relevant Methods *Note: optional parameters are in square brackets*

6.2 Manipulating Methods

delete() *returns* `void`(corresponds to *deleteVertex*)
Deletes self from the graph.

connect(Vertex: v1, [Integer: edgeId]) *returns* `Edge` (corresponds to *addEdge*)
Creates a new `Edge` object from self to `v1`. The directedness of the Edge will correspond to the type of Graph the edge is being added to (directed Graph → directed edge, automaton → transition, etc.). If un-directed, v.connect(v1) is equivalent to v1.connect(v). If no `edgeId` is passed, a suitable id is chosen. If an id is passed that has already been assigned, a suitable new one is silently chosen.

getAllEdgesBetween(Vertex: v1) *returns*`void` (corresponds to *getAllEdgesBetween)*

Deletes all edges in the graph between self and `v1` respecting the direction if the graph has directed edges. If no such edges exist, nothing happens.

6.3 Getter Methods

getId() *returns* `int`
Returns the id of self

getLabel() *returns* `string`
Returns the label of self

getFillColor() *returns* `string`
Returns the (fill) color of self, in the format that the user *had* specified (hex or rgb).

getColor() *returns* `string`
Same as getFillColor()

getStrokeColor() *returns* `string`
Returns the stroke color of the `Vertex` object, in the format that the user *had* specified (hex or rgb).

get(String: property) *returns* `class specified by property`
Returns the value of `prop`. For example, v.get(label") would return a string. Possible values for `prop` are label, radius, fillColor, strokeColor, and shape, although the most common queried attributes have their own query methods. This is also useful if you wish to define your own vertex attributes in an extension of the Vertex class.

getNeighbours() *returns*`list of Vertex objects` (corresponds to *getNeighbours*)

Returns a list of all the vertices connected to self, regardless of direction.

getOutgoingNeighbours() *returns* `list of Vertex objects` (corresponds to *getOutgoingNeighbours*)

Returns a list of all the vertices v' such that there is an edge from self to v' in directed graphs (equivalent to getNeighbours in undirected graphs)

getIncomingNeighbours() *returns* `list of Vertex objects` (corresponds to *getIncomingNeighbours*)

Returns a list of all the vertices v' such that there is an edge from v' to self in directed graphs (equivalent to getNeighbours in undirected graphs)

getIncidentEdges() *returns* `list of Edge objects` (corresponds to *getIncidentEdges*)

Returns a list of all the Edges e with an endpoint at self, regardless of their direction.

getOutgoingEdges() *returns* `list of Edge objects`

Returns a list of all the Edges e with an *source Vertex* = self.

getIncomingEdges() *returns* `list of Edge objects` (corresponds to *getIncomingEdges*)

Returns a list of all the Edges e with an *target Vertex* = self.

6.4 Setter Methods

setLabel(String: label) *returns* `void`
(corresponds to *setVertexLabel*)

Sets the vertex's label.

setRadius(float: radius) *returns*`void`(corresponds to *setVertexR*)

Sets the radius (ie width and height) of self to `radius`.

setHeight(float: height) *returns* `void`

Sets the height of self to `height`.

setShape(String: shape) *returns* `void` (corresponds to *setVertexShape*)

Sets the shape of self to `shape`. Currently supported shapes are "ellipse",“diamond",and "rectangle".

setWidth(float: width) *returns* `void`

Sets the width of self to `width`.

An explanation of width, height, and radius

setCoordinates((Integer,Integer): coordinates) *returns* `void`
Sets the vertex's coordinates to the coordinate pair passed. It is possible to specify only one of the two, for example by saying v.setCoordinates((None,42)); which sets the vertex's y coordinate to 42 and doesn't affect the x coordinate.

setFillColor([string: colorHex],[(int,int,int): colorRGB]) *returns* `void` (corresponds to *setVertexFillColor*)

Sets the *fill* colour to the Hex code colour specified as a string or the RGB colour specified. colorHex can also be a string of a common colour, such as `red"` or `"green.` Full list of gralog supported color names

Note that one of the two optional color parameters *must* be filled, and both of them *cannot* be filled.

Color setting example

setColor([string: colorHex],[(int,int,int): colorRGB]) *returns* `void`

Same as setFillColor

setStrokeColor([int: colorHex],[(int,int,int): colorRGB])) *returns* `void` (corresponds to *setVertexStrokeColor*)

Sets the *stroke* colour to the Hex code colour specified as a string or the RGB colour specified. colorHex can also be a string of a common colour, such as `red"` or `"green.` Full list of gralog supported color names

Note that one of the two optional color parameters *must* be filled, and both of them *cannot* be filled.

Color setting example

***setVertexRadius(Vertex: v, float: radius)** *returns*`void` *See proper method*

Sets the radius (ie width and height) of the given vertex shape to `radius`.

setVertexHeight(Vertex: v, float: height) *returns* `void`

Sets the height of the given vertex to `height`.

setVertexWidth(Vertex: v, float: width) *returns* `void`

setProperty(String: propertyName, String: propertyValue) *returns* `void` (corresponds to *setVertexProperty*)

Sets self's attribute specified by `propertyName` to the value in `propertyValue`. Be careful using this method! If you can, use the pre-programmed methods instead.

Sets the width of the given vertex to `width`.

An explanation of width, height, and radius

7 Class Edge - Documentation

7.1 Instance Variables

There are instance variables - however, they are not meant to be directly accessed by the user and are thus not detailed here.

Relevant Methods *Note: optional parameters are in square brackets*

7.2 Manipulating Methods

delete() *returns* `void` (corresponds to *deleteEdge*)
Deletes self from the graph.

7.3 Getter Methods

getId() *returns* `int`
Returns the id of self

getLabel() *returns* `string` (corresponds to *getEdgeLabel*)
Returns the label of self

getWeight() *returns* `float` (corresponds to *getEdgeWeight*)
Returns the weight of self

get(String: property) *returns* `class specified by property` (corresponds to *getEdgeProperty*)
Returns the value of `prop`. For example, e.get(label") would return a string. Possible values for prop are label, weight, isDirected, thickness, color, type, and edge-Type, although the most common queried attributes have their own query methods. This is also useful if you wish to define your own vertex attributes in an extension of the Vertex class.

getColor() *returns* `string`
Returns the color of self, in the format that the user *had* specified (hex or rgb).

getSource() *returns* `Vertex`
Returns the source endpoint of self

getTarget() *returns* `Vertex`
Returns the target endpoint of self

get(String: property) *returns* `class specified by property`
Returns the value of `prop`. For example, e.get(label") would return a string. Possible values for `prop` are label, radius, fillColor, strokeColor, and shape, although the most common queried attributes have their own query methods. This is also useful if you wish to define your own vertex attributes in an extension of the Vertex class.

getProperty(String: property) *returns* `class specified by property`
Returns the value of `prop`. Same as *get(String: property)*

getAdjacentEdges() *returns* `list of Edge objects`

Returns a list of all the Edges that share an endpoint with self.

7.4 Setter Methods

setLabel(String: label) *returns* `void`
(corresponds to *setEdgeLabel*)

Sets the vertex's label.

setContour(str: contour) *returns* `void` (corresponds to *setEdgeContour*)

Sets the contour of passed self. Possible values are `"dashed"`, `"dotted"`, `plain"`.

setColor([string: colorHex],[(int,int,int): colorRGB]) *returns* `void` (corresponds to *setEdgeColor*)
Sets the colour to the Hex code colour specified as a string or the RGB colour specified. colorHex can also be a string of a common colour, such as `red"` or `"green.` Full list of gralog supported color names

Note that one of the two optional color parameters *must* be filled, and both of them *cannot* be filled.

Color setting example

setWeight(float: weight) *returns* `void` (corresponds to *setEdgeWeight*)

Set's the weight of self to the given weight.

setThickness(float: weight) *returns* `void` (corresponds to *setEdgeThickness*)

Set's the thickness of the line of self to the given thickness.

setProperty(String: propertyName, String: propertyValue) *returns* `void` (corresponds to *setEdgeProperty*)

Sets self's attribute specified by `propertyName` to the value in `propertyValue`. Be careful using this method! If you can, use the pre-programmed methods instead.

8 Class Graph

8.1 Instance Variables

Instance Variable	Meaning and Usage

Dictionary *vertices*	A dictionary that holds all of the Vertex objects known to the graph. This should ideally not be changed by the programmer.
Dictionary *edges*	A dictionary that holds all of the Edge objects known to the graph. This should ideally not be changed by the programmer.
Integer *id*	The id of the graph that is used in communication with gralog. This should ideally not be changed by the programmer.
Dictionary *variablesToTrack*	Objects in format (name,value). These are displayed in the Algorithm Control Panel. These may be changed.
Dictionary *variablesToTrack*	Objects in format (name,value). These are displayed in the Algorithm Control Panel. These may be changed.

Relevant Methods *Note: optional parameters are in square brackets*

Graph([str: format]) *returns* a Graph Object.

This is the constructor for the Graph class. It returns one of two things, depending on the parameter passed. If None is passed as the parameter, then *no new graph is created*. Rather, the graph that is currently open in Gralog is passed to your program and a reference to it is stored in the Graph object. This Graph object now also contains all relevant (rudimentary) features of the open graph. The other parameter options are:

- nothing, e.g. `g = Graph()`, (this is the default, which is interpreted as "undirected")
- `"undirected"`,
- `"directed"`,
- `"buechi"`,
- `"kripke"`,
- `"automaton"`.

Use of these parameters causes Gralog to create a new (empty) graph of the requested type, a reference to which is then passed back to your program and stored in the returned Graph object.

8.2 Graph Manipulating Methods

Note: every time a `Vertex` object is a function *parameter*, the programmer may pass *either* a `Vertex` object *or* a valid vertex id; likewise for `Edge` objects as parameters and edge id's. In order to avoid redundancy, there are not duplicate method descriptions.

Another Note: vertex/edge manipulating methods (such as getting/setting label or some other attribute) can be done using the following methods. However, they are actually not designed to be directly used. Rather, you must use the methods in the Vertex and Edge classes that correspond to each. If you do use these methods directly, it may cause inconsistency between python and Gralog, leading to avoidable conundra and discontent. Corresponding methods are linked, and methods to be avoided are marked with a *.

addVertex([int: vertexId][(double, double): pos]) *returns* `Vertex` object

Creates a new `Vertex` object. If both coordinates coordinates in coordinate Vector `(x_coord,y_coord)` are *not* passed, random coordinates are chosen If no `vertexId` is passed, a suitable id is chosen by Gralog automatically. If an id is passed that has already been assigned, a suitable new one is silently chosen.

The `vertex` returned has the final id assigned to it.

Example

***deleteVertex(Vertex: v)** *returns* `void` *See proper method*

Deletes the given vertex from the graph. If the vertex does not exist, the program continues[1]

***addEdge(Vertex: source, Vertex target,[int: egdeId])** *returns* `Edge` object *See proper method*

Creates a new `Edge` object from the source vertex to the target vertex. The directedness of the Edge will correspond to the type of Graph the edge is being added to (directed Graph → directed edge, automaton → transition, etc.). If un-directed, the order of target and source vertex is irrelevant. If no `edgeId` is passed, a suitable id is chosen. If an id is passed that has already been assigned, a suitable new one is silently chosen.

***deleteEdge(Edge: e)** *returns* `void` *See proper method*

Deletes the edge from the graph. If the edge does not exist, nothing happens.[1]

deleteEdge((Vertex,Vertex): edge) *returns* `void` [2]

Deletes the edge with the greatest id from the list of all edges between the vertex in the first position of the tuple `edge` and the vertex which is the second position in tuple `source` respecting the direction if the graph has directed edges. If no such edge exists, nothing happens.

existsEdge((Vertex,Vertex): edge) *returns* `void` [2]

Returns whether there exists an edge between the first vertex of the tuple `edge` and the second (or from the first to the second in directed graphs).

existsEdge(Edge: edge) *returns* `Boolean`

Returns whether that specified edge (ie. the one with ID associated with the object) currently exists in the specified graph in Gralog.

existsVertex(Vertex: vertex) *returns* `Boolean`

Returns whether that specified Vertex (ie. the one with ID associated with the object) currently exists in the specified graph in Gralog.

***getAllEdgesBetween((Vertex,Vertex): vertexPair)** *returns* `void` *See proper method*

Returns all edges in the graph between the vertex in the first position of the tuple `vertexPair` and the vertex which is the second position in tuple `vertexPair` respecting the direction if the graph has directed edges. If no such edges exist, nothing happens.

8.3 Setter Functions

***setVertexFillColor(Vertex: v,[string: colorHex],[(int,int,int): colorRGB])** *returns*`void` *See proper method*

Sets the *fill* colour of the given `Vertex` to the Hex code colour specified as a string or the RGB colour specified. colorHex can also be a string of a common colour, such as `red"` or `"green.` Full list of gralog supported color names

[1] you will however receive a warning message in the console

[2] make sure this is actually a tuple, or it will not work! Also note vertices and vertex id's can be mixed and matched, should that be practical, such that you could call g.edgeMethodFoo((Vertex,id)) or g.edgeMethodFoo((id,Vertex)) and they would be equivalent

Note that one of the two optional color parameters *must* be filled, and both of them *cannot* be filled.

Color setting example

***setVertexStrokeColor(Vertex: v,[string: colorHex],[(int,int,int): colorRGB]))** *returns* `void` *See proper method*

Sets the *stroke* colour of the given `Vertex` to the Hex code colour specified as a string or the RGB colour specified. colorHex can also be a string of a common colour, such as `red"` or `"green."` Full list of gralog supported color names

Note that one of the two optional color parameters *must* be filled, and both of them *cannot* be filled.

Color setting example

***setEdgeContour(Edge: edge, str: contour)** *returns*`void` *See proper method*

Sets the contour of passed `Edge`. Possible values are `"dashed"`, `"dotted"`, `plain"`.

***setEdgeColor(Edge: v, [string: colorHex],[(int,int,int): colorRGB]))** *returns* `void` *See proper method*

Sets the colour of the given `Edge` to the Hex code colour specified as a string or the RGB colour specified. colorHex can also be a string of a common colour, such as `red"` or `"green."` Full list of gralog supported color names

Note that one of the two optional color parameters *must* be filled, and both of them *cannot* be filled.

Color setting example

setVertexDimension(Vertex: v, float: width, str: dimension) *returns* `void`

Sets the dimension of the given vertex to the dimension specified. This functionality is primarily useful for non-standard shapes, if you were to extend gralog to include such a thing. Other than in this case, it is recommended that you use the built-in functions.

***setVertexShape(Vertex: v, str: shape)** *returns* `void` *See proper method*

Sets the given vertex to be the specified shape. Currently supported are "`ellipse"`, `"diamond"`, and `rectangle"`.

***setEdgeWeight(Edge: e, float: weight)** *returns* `void` *See proper method*

Sets edge weight to `weight`. Not to be confused with setEdgeThickness

***setEdgeThickness(Edge: e, float: thickness)** *returns* `void` *See proper method*

Sets edge thickness to `thickness`. Not to be confused with setEdgeWeight

***setEdgeProperty(Edge: edge, String: propertyName, String: propertyValue)** *returns* `void` *See proper method*

Sets `edge`'s attribute specified by `propertyName` to the value in `propertyValue`.

***setVertexProperty(Vertex: vertex, String: propertyName, String: propertyValue)** *returns* `void` *See proper method*

Sets `vertex`'s attribute specified by `propertyName` to the value in `propertyValue`.

***setVertexLabel(Vertex: v, str: label)** *returns* `void` *See proper method*

Sets the edge's label to the `label`

***setEdgeLabel(Edge: v, str: label)** *returns* `void` *See proper method*

Sets the vertex's label to the `label`

8.4 Getter Functions

getGraph(str: graphFormat) *returns* `str`

Returns a string representation of the graph. Several formats are supported: GraphXML (parameter "XML"), TikZ (parameter tikz"), Trivial Graph Format (paramter TGF"), and Gralog Trivial Graph Format (pamater "GTGF"). Read more about them here.

This is useful in at least the following two scenarios: 1: to take the graph Gralog has given, and import it into an external library for processing there (such as NetworkX, or TODO: another popular library). 2: to copy the contents of one graph into another Gralog graph. See an example here.

getVertices() *returns* `list of Vertex objects`

Returns a list of all the vertices in the graph.

getAllVertices() The same as `getVertices()`.

getEdges() *returns* `list of Edge objects`

getAllEdges() The same as `getEdges()`.

Returns a list of all the edges graph.

***getNeighbours(Vertex: vertex)** *returns* `list of Vertex objects` *See proper method*

Returns a list of all the vertices connected to `vertex`, regardless of direction.

***getOutgoingNeighbours(Vertex: vertex)** *returns* `list of Vertex objects` *See proper method*

Returns a list of all the vertices v' such that there is an edge from `vertex` to v' in directed graphs (equivalent to getNeighbours in undirected graphs)

***getIncomingNeighbours(Vertex: vertex)** *returns* `list of Vertex objects` *See proper method*

Returns a list of all the vertices v' such that there is an edge from v' to `vertex` in directed graphs (equivalent to getNeighbours in undirected graphs)

***getIncidentEdges(Vertex: vertex)** *returns* `list of Edge objects` *See proper method*

Returns a list of all the Edges e with an endpoint at `vertex`, regardless of their direction.

getOutgoingEdges(Vertex: vertex) *returns* `list of Edge objects` (corresponds to *getOutgoingEdges*)

Returns a list of all the Edges e with an *source Vertex* = `vertex`.

getIncomingEdges(Vertex: vertex) *returns* `list of Edge objects`

Returns a list of all the Edges e with an *target Vertex* = `vertex`.

getAdjacentEdges(Edge: edge) *returns* `list of Edge objects`

Returns a list of all the Edges e' that share an endpoint with `edge`.

***getEdgeWeight(Edge: edge)** *returns* `float` *See proper method*

Returns the weight of `edge`.

***getEdgeLabel(Edge: edge)** *returns* `string` *See proper method*

Returns the label of `edge`.

***getEdgeProperty(Edge: edge, String: prop)** *returns* `type of prop` *See proper method*

Returns the value of `prop` in the passed `Edge`. For example, g.getEdgeProperty(e1,"weight") would return a float. Possible values for `prop` are label, weight, isDirected, thickness, color, type, and edgeType, although the most common queried attributes have their own query methods. This is also useful if you wish to define your own edge attributes in an extension of the Edge class.

getVertexProperty(Vertex: vertex, String: prop) *returns* `class specified by prop`

Returns the value of `prop` in the passed `Vertex`. For example, g.getVertex(v1,label") would return a string. Possible values for `prop` are label, radius, fillColor, strokeColor, and shape, although the most common queried attributes have their own query methods. This is also useful is you wish to define your own vertex attributes in an extension of the Vertex class.

End: private!

requestVertex() *returns* `Vertex`
Waits for user to click a vertex, then returns the vertex that the user clicks on.

requestRandomVertex() *returns* `Vertex`
Chooses a random vertex from the graph.

requestEdge() *returns* `Edge`
Waits for user to click a edge, then returns the edge that the user clicks on.

requestRandomEdge() *returns* `Edge`
Chooses a random edge from the graph.

requestInteger() *returns* `Integer`
Waits for user to click a integer, then returns the integer that the user clicks on.

requestFloat() *returns* `Float`
Waits for user to click a float, then returns the float that the user clicks on.

requestString() *returns* `String`
Waits for user to click a string, then returns the string that the user clicks on.

8.5 Control Functions

pause(String[]: *args) *returns* `void`
Waits for user to either press the space bar, or the "playïcon in the Algorithm Control Panel. While paused, tracked variables are displayed in the Algorithm Control Panel. Optionally, in the args input, you can add the following options:

- you can give, as the first parameter, the rank of the pause. In this way you can give the pauses a hierarchy. If during a pause of rank n", you press the "skip"button the Algorithm Control Panel, afterwards all pauses of rank n or higher will be ignored.
- You can add variables you would like to track individually in the moment, in the form of name-value

```
g.pause(("name1","val1"),("name2",var2),...("nameN",varN));
```

Either strings or variables can be passed as the value, and a string representation of the variables will be displayed by Gralog.

track(String: name, Variable: var) *returns* `void`
Specifies variable `var` to be tracked under the name `name`. Whenever the script is programmatically paused (ie. not by just clicking the pause button), that key-value pair will be shown in its most updated form in the Algorithm Control Panel.

Word to the wise: if you pass an int, str, or other primitive-acting variable, the tracked value will not be updated as the variable itself it updated. In this case, you will have to re-track the variable (I'm sorry, but python is too high-level language to support integer pointers and the like). However, a Vertex, for example, being a pointer to an object in memory, will be displayed in its most updated form each time.

unTrack(String: name) *returns* `void`

Disassociates the name `name` from any value it may be tracking. Now, if the script is programmatically paused, it will no longer be displayed with its former value.

A simple example for how to use variable Tracking can be found here.

9 Code Examples

addVertex([int: vertexId][(double: x_coord, double: y_coord)]) *returns* `Vertex` object

Example 2.

addVertex Example

```
#!/usr/bin/python
#HelloWorld.py
from Gralog import *
g = Graph(None);
#end boiler plate

v = g.addVertex(); #adds a vertex with a random id and random coordinates
v.setLabel("v: " + str(v.getId()));#shows the id gralog assigned on the vertex in Gralog

v1 = g.addVertex(420);
v1.setLabel("v1: " + str(v1.getId()));

v2 = g.addVertex((-10,2.2)); #adds a vertex at (0,2.2) and random id*
v2.setLabel("v2: " + str(v2.getId()));

v3 = g.addVertex(0,(11,-3)); #adds a vertex at (11,-3) with id=0
v3.setLabel("v3: " + str(v3.getId()));

g.pause();

#now we try and add a new vertex with an id that is in use

vDuplicate = g.addVertex(420);
vDuplicate.setLabel("dup: " + str(vDuplicate.getId())); #it ends up not being 420
```

*technically this should be `v2 = g.addVertex(pos=(0,2.2))`; however, in order to make it user friendly the library allows for the user to misuse parameters in order to preserve clarity and brevity

Color Setting examples *Example 3.*

Color Setting Example

```
#!/usr/bin/python
#HelloWorld.py
from Gralog import *
g = Graph("directed");
#end boiler plate

v = g.addVertex();
v.setColor("PUCE");
g.pause();
v.setColor(colorHex="123456");
g.pause();
```

```
v.setColor(colorRGB = (255,255,255));
g.pause();
v.setStrokeColor("red");
g.pause();
v.setStrokeColor("#101010");
g.pause();
v.setStrokeColor(colorRGB = (1,2,3));
g.pause();
v2 = g.addVertex();
e = g.addEdge(v,v2);
e.setColor("green");
g.pause();
e.setColor(colorHex="99999f");
g.pause();
e.setColor(colorRGB = (23,45,131));
#the following will not work
e.setColor((23,45,131));
```

Color Setting examples *Example 4.*

getGraph Example

```
#!/usr/bin/python
#Showing how to load a graph from python into a string, then transfer it to another graph
from Gralog import *
g = Graph("directed");
g2 = Graph("directed");
#end boiler plate

#populate g with 10 vertices in random locations
for x in range(10):
  g.addVertex();

gString = g.getGraph("GTGF");#load g into a string
g2.setGraph("GTGF",gString);#set g2 to graph isomorphic to g
```

Variable Tracking Example *Example 5.*

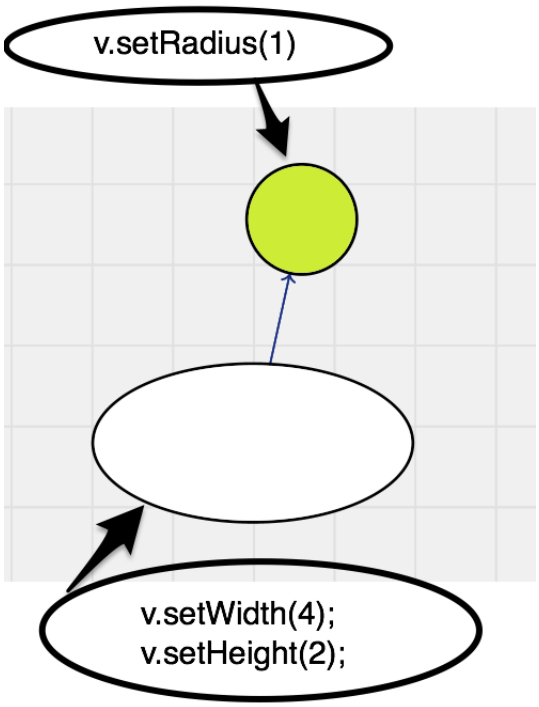
Variable Tracking Example

```
#!/usr/bin/python
#TrackingExample.py
from Gralog import *
g = Graph("directed");

vertices=[];
g.track("vertices",vertices);
for x in range(5):
  v = g.addVertex();
  v.setLabel("v" + str(x));
  vertices.append(v);
g.pause();#the array with the vertices will be displayed
g.unTrack("vertices");
for x in vertices:
  g.track("label of vertex "+str(x),x.getLabel());
g.pause();#each vertex will be displayed with its label
```

```
g.pause(("hello","world"));#each vertex, as well as the pair "hello","world" will be
    displayed
```

An explanation of the width, height, and radius properties of a `Vertex` object



9.1 Graph formats in detail

The four graph formats Gralog supports are the following: GraphML, Trivial Graph Format, TikZ, and Gralog Trivial Graph Format. The first three are cross-platform compatible with other libraries, and can be imported and exported easily with the Gralog Python library. The last is primarily useful for intra-gralog communication. It is a simple extension of TGF, that introduces edge id's in the format. This allows simple graphs to be passed between the Gralog Python programs and Gralog without losing any basic data that cannot be retrieved. The locations, colors, labels, weights etc. are not preserved in this format (hence trivial) but for simple uses it is quick and practical.

Tabelle 3: Color Names supported by Gralog

Gralog color name	**Hex Value**
WHITE	#FFFFFF
BLACK	#000000
BLUE	#0000FF
GREEN	#00FF00
RED	#FF0000
GRAY	#808080
YELLOW	#FFFF00
CYAN	#00FFFF
MAGENTA	#FF00FF
SILVER	#C0C0C0
MAROON	#800000
OLIVE	#808000
DARK GREEN	#008000
PURPLE	#800080
TEAL	#008080
NAVY	#000080
ORANGE	#FF4500