The Rocs Handbook

Tomaz Canabrava
Andreas Cord-Landwehr

Contents

Abstract

Rocs is a graph theory tool by KDE.

Chapter 1

Introduction

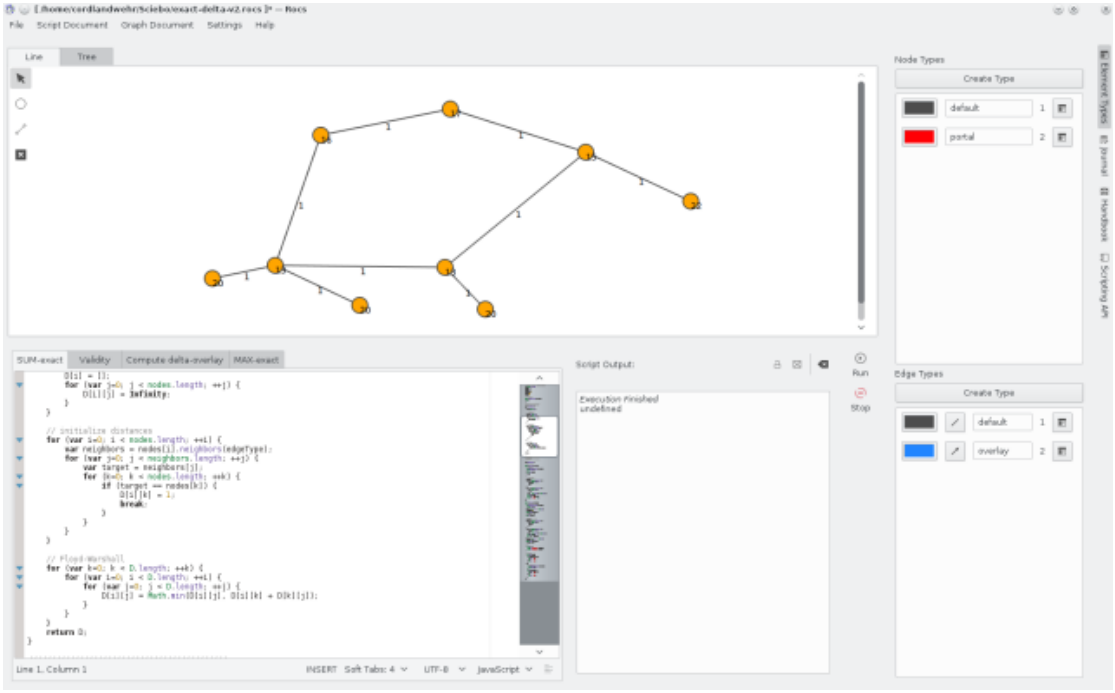
This chapter provides an overview of the core features and the typical workflows. The most important parts are Section 1.2 and chapter 3, which together should allow every new user to directly start using Rocs.

1.1 Goals, Target Audience, and Workflows

Rocs is a Graph Theory Tool for everybody interested in designing and studying graph algorithms. In particular, those are

- lecturers, who want to demonstrate algorithms to their students,
- students and researchers, who want to see how their algorithm perform, and
- everybody who is interested in data structures and algorithms.

For all them, Rocs provides an easy to use graphical editor for creating graphs, a powerful scripting engine to execute algorithms, and several helper tools for simulations, experiments, and graph exports. The typical way of using Rocs is to create a graph, either by hand (i.e., dragging nodes and edges to the whiteboard), or by using one of the graph generators. Graph algorithms then can be implemented and executed on the created graph and all changes, which the algorithm performs, are visible immediately in the graph editor.



1.2 Rocs in a Nutshell

Every Rocs session is a project: when opening Rocs an empty project is created, when loading some project it becomes the current project. Hereby, a project itself consists of *graph documents*, *scripts/algorithms*, and a *journal*.

1.2.1 Graph Documents

A graph document represents the content of a whiteboard in the graph editor. It contains information about the user defined node and edge types, their properties, and about the already created nodes and edges. This is, Rocs understands the set of all nodes and edges of a graph document to form a (not necessarily connected) graph. Everything belonging to a graph document is accessible by the script engine via the global object **`Document`**.

1.2.2 Edge Types

In some scenarios, graphs consist of different types of edges (e.g., an undirected graph plus the tree edges computed by a breadh-first-search algorithm) that shall be handled and displayed differently. For this, besides a default edge type, you can define arbitrary other edge types. Each edge type has its individual visual representation, dynamic properties, and can be set to be either undirected or directed. The scripting interface provides convenience methods to specifically access only edges of specific types.

1.2.3 Node Types

Analog to edge types, you can define different types of nodes of a graph (e.g., to give some nodes special roles). Each node type has its own visual representation and dynamic properties.

1.2.4 Properties

Every (node or edge) element can have properties. Those properties must be setup at the corresponding node or edge type. Properties are identified and accessed by their names and can contain any value. To create new or change existing properties, use the **Element Types** sidebar and use the  **Properties** button to open the property dialog.

You can also use the scripting engine to access registered properties and change their values. In the following example we assume that the property ″weight″ is registered for the default edge type.

```
var nodes = Document.nodes()
for (var i = 0; i < nodes.length; ++i){
    nodes[i].weight = i;
}
for (var i = 0; i < nodes.length; ++i){
    Console.log("weight of node " + i + ": " + nodes[i].weight);
}
```

1.3 Tutorial

In this section we want to create an example project to explore some of the most important functions of Rocs. The goal is to create a graph and a script that illustrates a simple 2-approximate algorithm for the *minimum vertex cover* problem. The minimum vertex cover problem is the problem to find a subset of graph nodes C of minimal size such that each graph edge is connected to at least one node in C. This problem is known to be NP-hard and we want to illustrate how to find an approximation of factor 2 by computing a matching in the given graph.

Our goal is to visualize the relationship of the matching and the minimum vertex cover. For this, we want to specify two edge types, one to display matching edges and one type to display ″ordinary″ edges, as well as two node types that we use to distinguish nodes contained in C and those not contained in C.

1.3.1 Generating the Graph

For creating the graph, we use a default graph generator provided by Rocs. This can be found in the main menu at **Graph Document** → **Tools** → **Generate Graph**. There, we select a **Random Graph** with 30 nodes, 90 edges, and with seed 1 (the seed is the starting seed for the random graph generator; using the same seed multiple times results in same and reproducible graphs).

1.3.2 Creating the Element Types

We use the **Element Types** and create a second node type as well as a second edge type. For both new types we open the properties dialog by using the respective **Properties** buttons and set the IDs to `2`. Furthermore, we change the colors of elements of these two new types (to distinguish them from the default types). Finally, we set all edge types to be bidirectional, and the IDs of the default types to `1`.

1.3.3 The Algorithm

At last we have to implement the approximation algorithm. For this we use the following implementation:

```
for (var i=0; i < Document.nodes.length; i++) {
    Document.nodes[i].type = 1;
}
for (var i=0; i < Document.edges.length; i++) {
    Document.edges[i].type = 1;
}

var E = Document.edges(); // set of unprocessed edges
var C = new Array();      // matching edges
while (E.length > 0) {
    var e = E[0];         // we take first edge e={u,v}
    var u = e.from();
    var v = e.to();
    e.type = 2;           // set edge to be a matching edge
    E.shift();            // remove e (i.e., E[0]) from edge list
    C.push(u);            // add u to C
    C.push(v);            // add v to C

    // mark u,v as nodes in C
    u.type = 2;
    v.type = 2;

    // remove from E all edges incident to u or v
    var adjacent = u.edges();
    for (var i=0; i < adjacent.length; i++) {
        var index = E.indexOf(adjacent[i]); // find the index
        if (index != -1) {
            E.splice(index, 1); // remove it if really found
        }
    }
    var adjacent = v.edges();
    for (var i=0; i < adjacent.length; i++) {
        var index = E.indexOf(adjacent[i]); // find the index
        if (index != -1) {
            E.splice(index, 1); // remove it if really found
        }
    }
}
Console.log("Vertex Cover contains " + C.length + " nodes.");
```

1.3.4 Execute the Algorithm

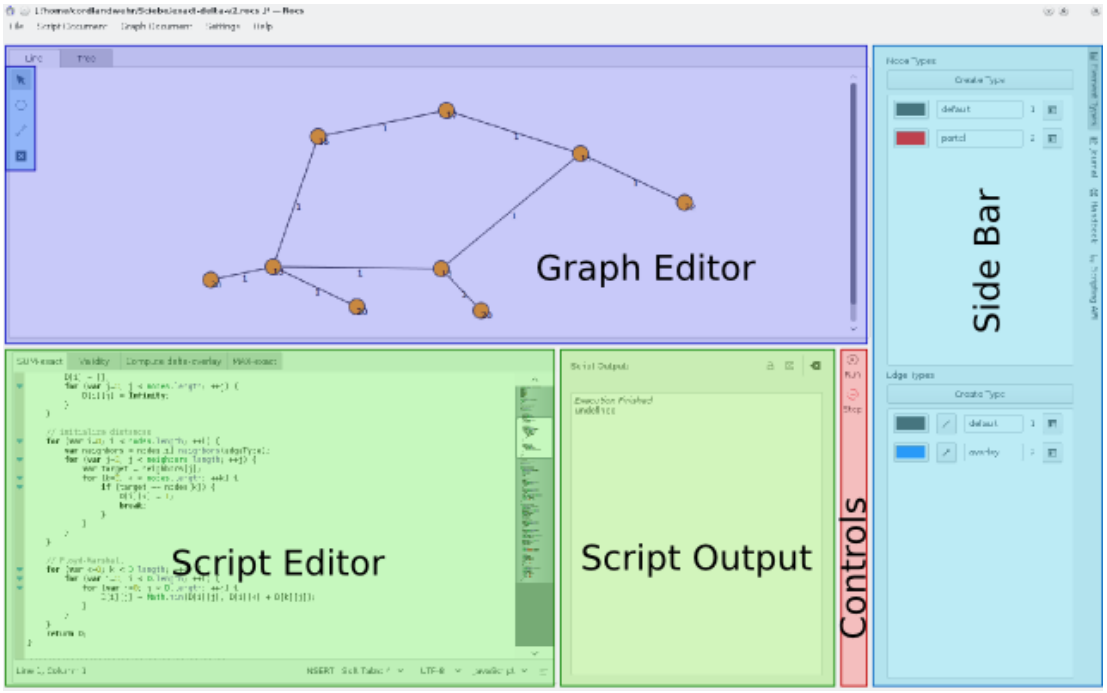
The algorithm can be executed by the **Run** button at the script control panel.

Chapter 2

The Rocs User Interface

2.1 Main Elements of the User Interface

The user interface is divided into several logical parts as presented at the screenshot below.



Graph Editor

The editor provides a whiteboard at that nodes and edges can be placed. Double-clicking at any of its elements opens a corresponding property menu. You can use the tools from the *Graph Editor Toolbar* to create and modify graphs.

Graph Editor Toolbar

The toolbar provides the **Create Node** or **Create Edge** tools, for creating new elements on the whiteboard. Note the extra-toolbar for selecting the respective node or edge type that becomes visible of one of these tools is selected. Also tools for selecting and moving as well as deleting elements are available here. For details see Section 2.2.

Side Bar

At the right, you can find the side bar that provides several tools for your workflow:

- Element Types: This widget gives you direct access to the available edge and node types.
- Journal: Each project has its own journal that can be used to, e.g. note tasks, results, or observations.
- Scripting API: To get direct access to the script documentation, you can open this widget.

Script Editor

In this text editor you can write algorithms as explained in detail in chapter 3. You can work on several script documents simultaneously by using several tabs.

Script Output

This text area either shows debug information or the script output of your algorithm, depending on the toggled setting at the top of this widget. If the script throws an error, automatically the debug output is presented.

Controls

Here you can find the controls for executing scripts. You can execute the script that is currently open at the script editor by pressing **Run**. While the script is executed, it is possible to stop execution by pressing the **Stop** button.

2.2 Graph Editor Toolbar

This toolbar consists of the following actions. Clicking at an action means that your mouse pointer applies this action at the graph editor whiteboard:

-  **Select and Move**: To select elements, either click at unused space at the whiteboard, keep the mouse pressed and draw a rectangle that contains some data elements and/or edges to select these elements or otherwise directly click at an unselected element to select this element. If you click at a selected element or a set of selected elements, respectively, by keeping the mouse pressed and moving around you can move these elements. Moving selected elements is also possible with the arrow keys.

-  **Create Node**: Click at an arbitrary position at the graph editor whiteboard to create a new data element that belongs to the currently selected data structure. By keeping the mouse pointer pressed at the button, a menu shows up at which the data type of the new created data elements can be selected (only if different data types exist).

-  **Create Edge**: Click at one data element, keep the mouse pressed and draw a line to another data element to which the edge shall point. This action is only successful if the current graph allows to add this edge (e.g., in an undirected graph you are not allowed to add multiple edges between two data elements). By keeping the mouse pointer pressed at the button, a menu shows up at which the edge type of the new created edges can be selected (only if different edge types exist).

-  **Delete**: Click at an element to delete it. If you delete a node, all adjacent edges are also deleted.

Chapter 3

Scripting

3.1 Executing Algorithms in Rocs

Rocs internally uses the QtScript Java Script engine. This means, all algorithms that you implement must use Java Script. In the following, we explain how to access and change elements of a graph document from the scripting engine. It is important to note that changes done by the scripting engine are directly reflected at the properties at the graph editor elements.

3.1.1 Control Script Execution

There are different execution modes for your algorithms:

-  **Run**: Execute the script until it finishes.
-  **Stop**: Stop script execution (only available while a script is executed).

3.1.2 Script Output

During the execution of an algorithm, debug and program output is displayed in the *Debug & Script Output*. If the scripting engine detects a syntax error in your script, the error is also displayed as debug message. Note that all program messages are also displayed at the debug output (displayed as bold text).

You can control the text that is displayed at the script output by the following functions:

```
Console.log(string message);            // displays the message as ↩
    script output
Console.debug(string message);          // displays the message as ↩
    debug output
Console.error(string message);          // displays the message as ↩
    error output
```

3.1.3 Scripting Engine API

The different parts of Rocs each provide a static element that can be accessed by the scripting engine. These are:

- **`Document`** for the graph document
- **`Console`** for the console log output

For the explicit API use and for a method reference, please see the inline help at the Rocs side bar.

Chapter 4

Import and Export

4.1 Exchange Rocs Projects

Rocs projects can be imported and exported as archived `.tar.gz` files. These archives can be used to exchange projects. Import and Export can be done with **Graph Document** → **Import Graph** and **Graph Document** → **Export Graph as**, respectively.

4.1.1 Import and Export of Graph Documents

Rocs currently supports import and export of the following file formats:

- DOT files, also known as Graphviz files
- GML files
- Trivial Graph Format files
- Keyhole Markup Language Format

4.1.1.1 Trivial Graph File Format

The *Trivial Graph Format* (TGF) is a simple text-based file format for describing graphs. A TGF file consists of a list of node definitions, that map the node IDs to labels, followed by a list of the edges. In this format it is only possible to have one label per node and one value per edge. Rocs interprets imported graphs as undirected graphs. Exported graphs will contain two edges per connection if connections are bidirectional.

4.1.1.1.1 Format Specification

- The file starts with a list of nodes (one node per line), followed by a line with the only character ˝#˝, followed by a list of edges (one edge per line).
- A node consists of an integer (identifier), followed by a space, followed by an arbitrary string.
- An edge consists of two integers (identifiers) separated by a space, followed by a space, followed by an arbitrary string. It is assumed that the directed edge points from the first identifier to the second identifier.

4.1.1.1.2 Example

```
1 starting node
2 transmitter
3 sink
#
1 2 blue
2 1 red
2 3 green
```

4.1.1.2 DOT Language / Graphviz Graph File Format

The DOT language is a plain text graph description language that allows both,a good human readable representation of graphs as well as an efficient processing by graph layout programs. DOT is the default file format for the Graphviz graph visualization suite, but is also widely used by other graph tools. The usual file extensions for DOT are *.gv* and *.dot*.

4.1.1.2.1 Unsupported Features

Rocs can parse every graph file that contains a graph specified according to the DOT language specification[1]. The support of language features is complete, despite of the following exceptions:

- subgraph: Due to the lack of a subgraph concept in Rocs, subgraphs are only imported as a set of date elements and connections. Especially, connections to or from subgraphs are not imported.
- HTML and XML attributes: Attributes (like labels) that contain HTML or XML syntax are read unchanged. Especially, not adjustment of fonts and styles are read from those attributes.

4.1.1.2.2 Example

```
digraph myGraph {
    a -> b -> c;
    b -> d;
}
```

[1] http://www.graphviz.org/content/dot-language

Chapter 5

Credits and License

Rocs
Program Copyright:

- Copyright 2008 Ugo Sangiori (ugorox AT gmail.com)
- Copyright 2008-2012 Tomaz Canabrava (tcanabrava AT kde.org)
- Copyright 2008-2012 Wagner Reck (wagner.reck AT gmail.com)
- Copyright 2011-2015 Andreas Cord-Landwehr (cordlandwehr AT kde.org)

Documentation Copyright:

- Documentation copyright 2009 Anne-Marie Mahfouf annma@kde.org
- Documentation copyright 2009 Tomaz Canabrava (tcanabrava AT kde.org)
- Documentation copyright 2011-2015 Andreas Cord-Landwehr (cordlandwehr AT kde.org)