

# **Design implementation and testing of XHTML 5 based solutions for interactive on-line inspection of XMCD 2.0 encoded outranking graphs**



**Gary Philippe Cornelius**

Faculty of Science, Technology and Communication

University of Luxembourg

This dissertation is submitted for the degree of  
*Bachelor of Engineering in Computer Science*

Academic Supervisor:  
Prof. Raymond Bisdorff

2014



## **Abstract**

In this report I am going to describe the development of the project that I was doing during my last semester of the Bachelor of Engineering in Computer Science at the University of Luxembourg.

The goal of this thesis is to find a way to export and visualize XMCD-2.0 encoded graphs in a way that allows users to manually explore graphs and show informations about their nodes and edges. The current solution to visualize graphs generated by Digraph3 only allows the export of a static picture. Unfortunately with static pictures it is not possible to get any additional information about certain edges or nodes and also analysing the graph itself becomes quite tedious as soon as the node number grows.

Therefore I am looking for a way to implement these outranking graphs in a dynamic way by using XHTML5 technology and I also analyse which solutions and libraries are existing on the market today, to see if we can use an already existing library in order to help us.

Dans ce rapport, je vais décrire le développement du projet que j'ai fait lors de mon dernier semestre dans le Bachelor en Informatique à l'Université du Luxembourg.

L'objectif de cette thèse est de trouver un moyen d'exporter et de visualiser des graphes encodés dans le format XMCD-2.0 de façon à permettre aux utilisateurs d'explorer manuellement ces graphes et de permettre aussi de montrer des informations supplémentaires sur leurs noeuds et les connections entre ces noeuds. La solution actuelle pour visualiser les graphes générés par Digraph3 ne permet que l'exportation d'une image statique. Malheureusement, avec des images statiques, il n'est pas possible d'obtenir des renseignements supplémentaires sur certaines connections ou noeuds. L'analyse de ces graphes devient donc très difficile quand le nombre des noeuds augmente.

Par conséquent, j'ai cherché un moyen d'afficher ces outranking graphes d'une manière dynamique en utilisant des nouvelles technologies comme XHTML5 et j'ai analysé également les solutions et les bibliothèques qui sont disponibles sur le marché aujourd'hui, pour voir s'il est possible d'utiliser une bibliothèque déjà existante afin de nous aider pour le développement.



## **Declaration**

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Gary Philippe Cornelius

2014



## **Acknowledgements**

First I would like to thank Prof. Raymond Bisdorff, my academic supervisor, for the provided help and supervision throughout the project.

I would also like to thank my family and Simona for the support and motivation that they gave me through my studies, especially in stressful times.

Finally, I would like to thank Mr. Christian Muller, Mr. Giorgos Ziazopoulos and all my other friends studying Computer Science at the University of Luxembourg. Without you it would not have been the same.





# Contents

<b>Contents</b>	<b>ix</b>
<b>List of Figures</b>	<b>xi</b>
<b>Nomenclature</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Introduction . . . . .	1
<b>2 Analysis</b>	<b>3</b>
2.1 Short title . . . . .	3
2.1.1 Small Bipolar Outranking Digraph . . . . .	4
2.1.2 Medium Bipolar Outranking Digraph . . . . .	5
2.1.3 Big Bipolar Outranking Digraph . . . . .	6
2.2 Analysis of existing libraries . . . . .	7
2.2.1 Springy.js . . . . .	7
2.2.2 InfoVis . . . . .	8
2.2.3 Cytoscape.js . . . . .	9
2.2.4 D3 - Data-Driven Documents . . . . .	10
2.3 Conclusion . . . . .	10
<b>3 Development - Introduction</b>	<b>11</b>
3.1 Kick start . . . . .	11
3.1.1 How does it work? . . . . .	11
3.2 The generated files . . . . .	12
3.3 Possible interactions . . . . .	14

<b>4</b>	<b>Development - Data Structures</b>	<b>17</b>
4.1	XMCD-2.0 . . . . .	17
4.2	Data-structure . . . . .	17
4.3	Import . . . . .	19
4.3.1	Graph type . . . . .	19
4.3.2	How does it work? . . . . .	19
4.3.3	Sequence Diagram . . . . .	23
4.4	Export . . . . .	24
4.4.1	How does it work? . . . . .	24
4.4.2	Problem . . . . .	24
4.5	Conclusion . . . . .	25
<b>5</b>	<b>Development - Visualization &amp; Control</b>	<b>27</b>
5.1	Visualization . . . . .	27
5.1.1	Edges . . . . .	27
5.1.2	Arrows . . . . .	28
5.2	Control . . . . .	29
5.2.1	Background . . . . .	29
5.2.2	Nodes . . . . .	30
5.2.3	Edges . . . . .	31
5.3	Conclusion . . . . .	33
<b>6</b>	<b>Conclusion</b>	<b>35</b>
6.1	Achievements . . . . .	35
6.2	Known limitations . . . . .	37
6.2.1	Browser Support . . . . .	37
6.2.2	Performance big graphs . . . . .	37
6.3	Further developments . . . . .	38
6.4	Experience . . . . .	38
	<b>References</b>	<b>39</b>
	<b>Appendix A Example Collection</b>	<b>41</b>

# List of Figures

2.1	GraphViz: Small BipolarOutrankingDigraph . . . . .	4
2.2	GraphViz: Medium BipolarOutrankingDigraph . . . . .	5
2.3	GraphViz: Big BipolarOutrankingDigraph . . . . .	6
2.4	Springy.js Basic . . . . .	7
2.5	InfoViz Example . . . . .	8
2.6	Cytoscape Example . . . . .	9
4.1	Sequence Diagram - importJSON() . . . . .	23
5.1	Graph: Curved edges . . . . .	28
5.2	Graph: Straight edges . . . . .	28
5.3	Graph: Arrow and edge types . . . . .	29
5.4	Graph: Background Context-menu . . . . .	29
5.5	Graph: Node Context-menu . . . . .	30
5.6	Graph: Edge Context-menu . . . . .	32
6.1	Graph: Outranking Example . . . . .	36
6.2	Graph: Outranking Inspect Example . . . . .	36
6.3	Graph: Performance on load . . . . .	37
6.4	Graph: Performance on idle . . . . .	38



# Chapter 1

## Introduction

### 1.1 Introduction

„Digraph3 is a Python-3 implementation of generic resources implementing decision aid algorithms in the context of bipolar-valued outranking digraphs. This computing resource is useful in the field of algorithmic decision theory and more specifically in outranking based multiple criteria decision aid.” [2]

The current implementation allows the export of GraphViz<sup>1</sup> (*see Chapter 1*) generated graphs in a graphics format like *.png* or *.pdf* and it is also possible to export those graphs as a *.html* file which embeds those generated graphical representations and other informations about the graphs. These graphical representations are static and do not allow any user interaction or editing. Also the amount of information provided by those static graphs about certain nodes or edges is very limited. Therefore the exploration and analysis of such graphs is tedious and leaves a lot of room for changes and improvements.

An important point that we have to keep in mind during our work is the size of these graphs. Meaning that small sized graphs have +-10 nodes, medium sized graphs have +-20 nodes and big sized graphs have more than 30 nodes. With GraphViz the visualization of medium sized graphs becomes already quiet confusing in case the graph is highly connected, which is mostly the case for outranking digraphs.

---

<sup>1</sup><http://www.graphviz.org/>

Therefore the idea of my thesis is to find a way to make the Digraph3 generated graphs dynamically explorable. This would allow the user to easily analyse and explore a complicated graph. Furthermore, the user is able to concentrate on important things such as the values of the edges between nodes. Although the visualization of those graphs seems not to be easy, the goal will be to make them as explorable as possible while keeping the design of a standard digraph.

With explorable, we understand that the user should be able to explore these visualized digraphs in an as easy as possible way. It should therefore be possible to turn these graphs around, i.e. drag nodes in the graph in order to be able to manually explore the graph. Also by selecting a node, it should be possible to gather information about this node, leave comments to certain nodes or give them a name.

It should also be possible to visualize the pairwise comparison of certain nodes, which is the result of a long computation inside the Digraph3 program.

Another important point was further reuse of the library and the ability to be easily extendable and understandable by future developers for further improvements.

As a result we decided to use HTML5 and other web technologies that are heavily used today. Furthermore I tried to make the code as modular as possible, such that a future developer is able to find a fast way into the code and to make the single functions reusable. One limitation here was that since we are exporting all the files with the help of Python, we should keep everything as compact as possible and should not generate too many files in order to not confuse future users.

In this thesis, I will first give you an overview of why we need a dynamic way to explore graphs, existing solutions and finally my solution with the needed explanations of why I used this technology and implementation.

# Chapter 2

## Analysis

In this chapter, I am going to analyse the current solution implemented in Digraph3 which uses GraphViz for the generation of static images in common a graphic format. Furthermore, I will explore which libraries exist on the market today that allow the visualization of dynamic graphs with help of XHTML5 inside the web browser.

### 2.1 Analysis GraphViz

The current implementation of Digraph3 contains an export function named *exportGraphViz()*, which is used for the export of static images generated by GraphViz. GraphViz is an open sourced graph visualization software.

The main reason why we are looking to find a better way to visualize the Digraph3 created graphs is because GraphViz only allows us to store static image, for example in the well known PNG (Portable Network Graphics) format, or other static image formats. In order to be able to work with these graphs in the most productive way, we need to find a more dynamic way to explore the graphs and therefore we want to visualize them in a way that makes them easily understandable and analysable.

The following code snippet generates an outranking graph with 10 nodes and exports the GraphViz generated graph in a PDF (Portable Document Format) file.

```
> from outrankingDigraphs import *  
> t = RandomCBPerformanceTableau(numberOfActions=10)  
> g = BipolarOutrankingDigraph(t)  
> g.exportGraphViz(graphType="pdf")
```

In the following lines I will briefly describe the limitations that we encounter using GraphViz, explain the weak points of the current implementation and the changes that need to be done in the new implementation.

### 2.1.1 Small Bipolar Outranking Digraph

This first example is a small Bipolar Outranking Digraph which consists of 10 nodes.

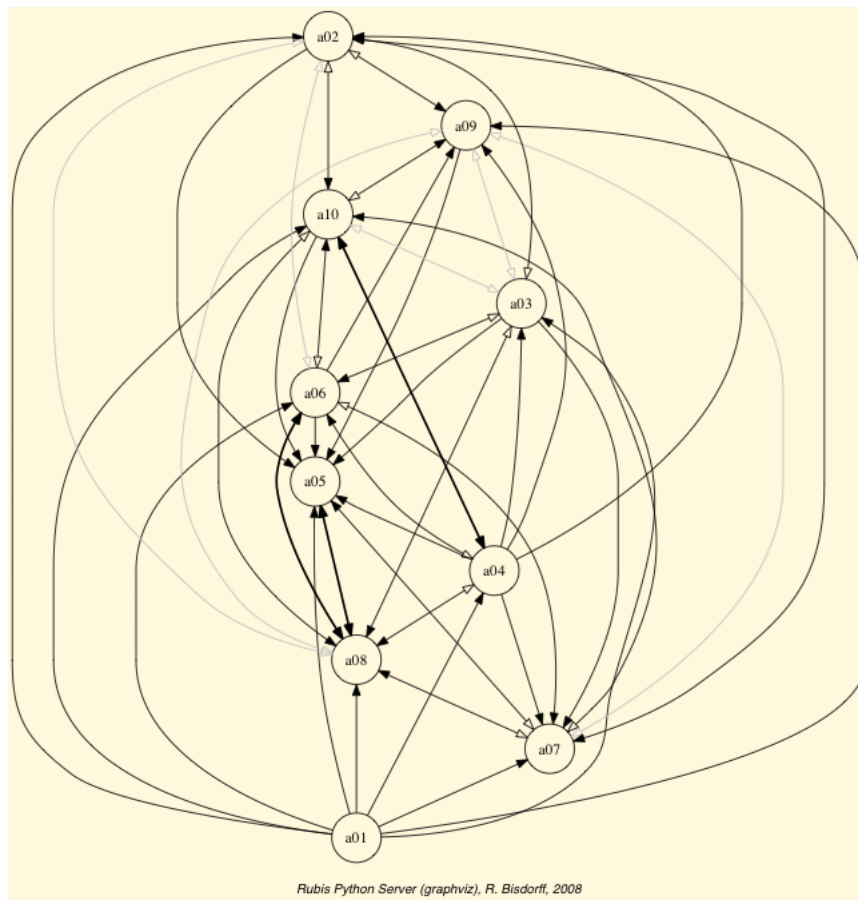


Fig. 2.1 GraphViz: Small BipolarOutrankingDigraph

As you can see, the overview of small graphs generated with GraphViz is quite good. If you focus on the nodes and edges you can still find out their connections quite fast. So that we could say that, the only limitation for small graphs is that we cannot know what exactly the attributes of the different nodes are, or what the connections between those nodes mean with respect to their value.



### 2.1.2 Medium Bipolar Outranking Digraph

This is an example of a medium sized Bipolar Outranking Digraph of 20 nodes. Here you will already be able to see the problems that we encounter visualizing this type of graphs.

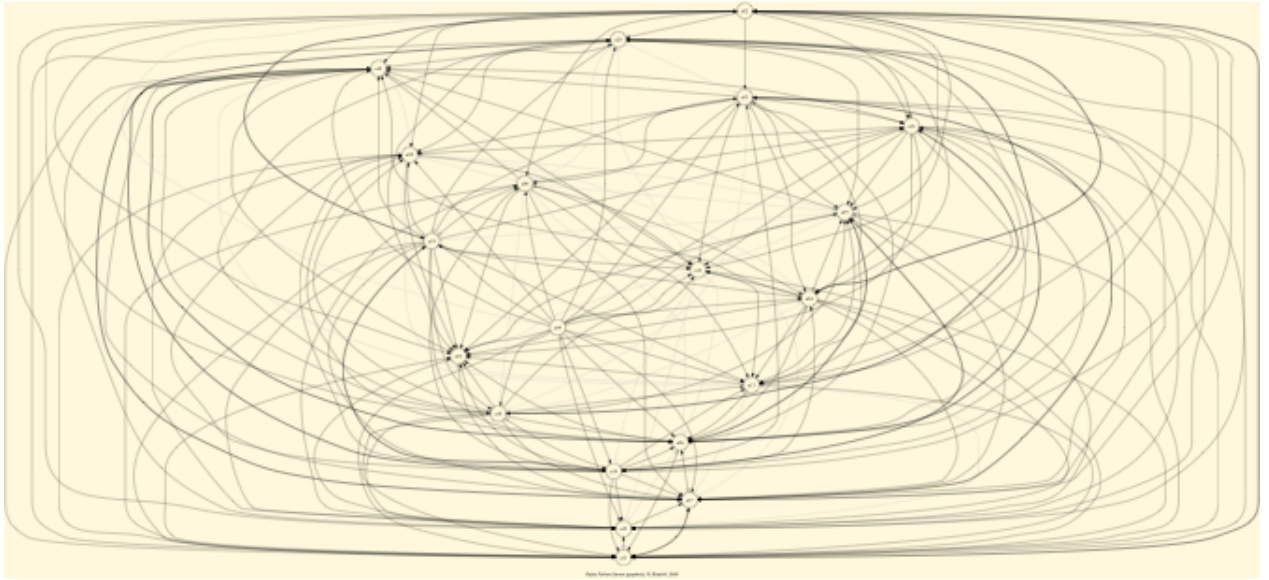


Fig. 2.2 GraphViz: Medium BipolarOutrankingDigraph

As already said, on a medium sized graph the overview that static images provide is already very limited. Edges are building a jungle of connections and we are not able to follow easily which edge is connected with which node, because for this we would have to follow the edge from start to end which is a very time consuming job. Even more if you want to compare several nodes. An analysis of these graphs becomes really tedious.

### 2.1.3 Big Bipolar Outranking Digraph

This is an example of a big sized BipolarOutrankingDigraph of 30 nodes.



Fig. 2.3 GraphViz: Big BipolarOutrankingDigraph

As you can see, you cannot see a lot. The graph is scaled 0.35% and it was technically not possible to include a better picture of an outranking digraph having this many nodes. What you can see on the picture is the immense number of edges. The visualisation of big graphs is a proof for the problems that we already encountered with medium-sized graphs and it becomes clear that the GraphViz solution is not optimal, if you want to inspect the graph graphically. Therefore we need to find a better solution to visualize and inspect these graphs.

I will also use the example of a big sized graph later in *Chapter 6* to proof that I was able to find a method which allows you to analyse also these big graphs in an effective way.

## 2.2 Analysis of existing libraries

### 2.2.1 Springy.js

Springy.js<sup>1</sup> is a JavaScript based library that uses a force directed graph algorithm based in real world physics in order to visualize directed graphs in a way that they look good. Springy is a quite basic library and doesn't seem to be widely used.

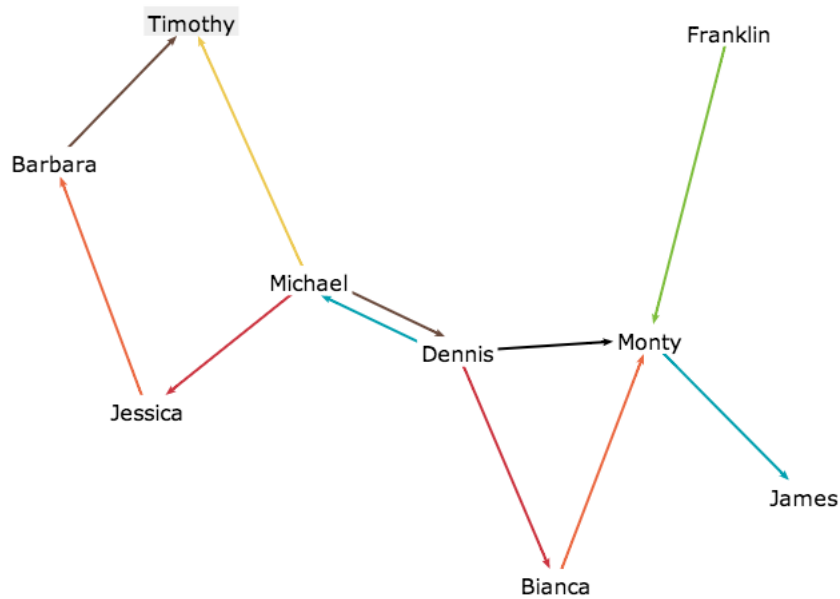


Fig. 2.4 Springy.js Basic

---

<sup>1</sup><http://getspringy.com/>

### 2.2.2 InfoVis

InfoViz<sup>2</sup> is a powerful JavaScript library that allows the creation of nearly every type of graph. Unfortunately the development activity seems not to be continued. At the time of writing, the last count of commits in one year was at around 10. Furthermore, the code examples provided seemed quiet complicated and not easy to use.

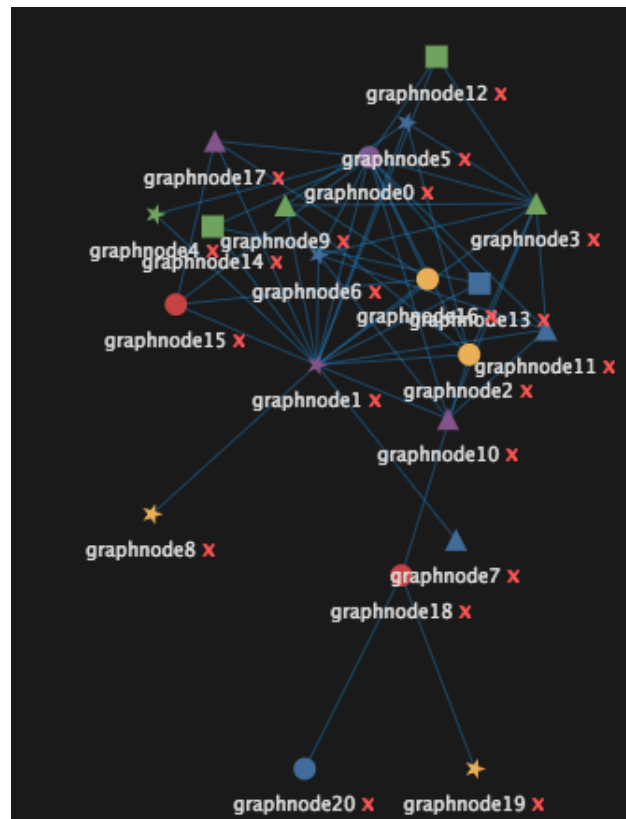


Fig. 2.5 InfoViz Example

---

<sup>2</sup><http://phillogb.github.io/jit/>

### 2.2.3 Cytoscape.js

Not to confuse with the Cytoscape desktop client, `cytoscape.js`<sup>3</sup> is a JavaScript library that can be used with HTML5. Cytoscape delivers us with nearly everything we need to create and analyse graphs in an effective way. The only reason why we did not choose Cytoscape was the number of references in forums and also the library seemed not so well developed in terms of examples and user friendliness for new developers.

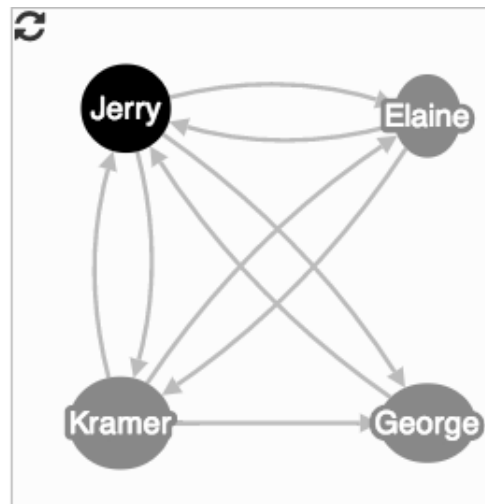


Fig. 2.6 Cytoscape Example

---

<sup>3</sup><http://cytoscape.github.io/cytoscape.js/>

### 2.2.4 D3 - Data-Driven Documents

I decided to use D3<sup>4</sup> for the following development. I took this decision because of the reasons already described in the Cytoscape paragraph, but also because of the feeling that I had reading through some code snippets of both libraries. The code examples that were used for D3 seemed much cleaner and easier to understand. Furthermore, the D3 GitHub repository was the most active and up to date.

D3 is a JavaScript library that allows you to manipulate documents based on data and visualize this data in nearly any possible form. Considering the rich feature list of D3, we will never be limited for any further extension or change in the Digraph3 visualization library. Furthermore, D3 supports force directed graphs, which I thought would be the right way to visualize our graphs in an easy and understandable way. In force directed graphs, the rearrangement of the nodes is done automatically by a force algorithm.

I will not give an example of a D3 graph in this part of my thesis, because you are going to see enough of them in the following Chapters where I describe the development phase of the project.

## 2.3 Conclusion

One of the hardest things was to decide which library to use, since there are a lot of graph drawing libraries on the market that would do the job just fine.

At the end I was able to take the decision to use the D3 - Data Driven Documents library. It was the best documented and seems to be the most used library according to forum posts and suggestions that people made. It also seems easy to import, manipulate and visualize data. But the deciding factor for me was that there are no limitations for the type of graph that we can create and further development will not be limited in any way.

Furthermore, D3 makes use of new technologies like CSS3, HTML5 and JavaScript, and is also widely used today, which were our main goals.

As a conclusion I would like to say that the decision for a library is a very hard choice and sometimes there is no right or wrong, but you just have to choose the one that according to your intuition seems to fit best.

---

<sup>4</sup><http://d3js.org/>

# Chapter 3

## Development - Introduction

In this chapter I give you a short introduction into the Digraph3 program, explain the generated files (*Chapter 3.2*) and the possible interaction with our editor(*Chapter 3.3*). The *Chapter 3.1 Kick start* can be seen as an introduction chapter into my work and the Digraph3 program. For the development details and explanations please consult the Chapters #4 and #5 which are dedicated to the implementation part of this project.

### 3.1 Kick start

In this subsection, I want to give you a brief overview of how exactly everything is working together and also give you a quick crash course on how to use Digraph3 together with my implementation of an interactive graph. At the end you will be able to construct a graph by yourself.

#### 3.1.1 How does it work?

First of all we need the Digraph3 Python-3.4[3] program to generate a graph. In the first example on the following page I show you how to generate a *RandomCBPerformanceTableau* with 10 actions and then transform it into a Bipolar Outranking Digraph with 10 nodes. These graphs are non editable, since they allow us to visualize the pairwise comparisons between nodes which are the result of long computations.

The second example shows how to generate a general graph that can be manipulated in every possible way.

Example 1:

```
> from outrankingDigraphs import *
> t = RandomCBPerformanceTableau(numberOfActions=10)
> g = BipolarOutrankingDigraph(t)
> g.exportD3()
or
> dg.showInteractiveGraph()
```

Example 2:

```
> from digraphs import *
> dg = RandomValuationDigraph(order=10)
> dg.exportD3()
or
> dg.showInteractiveGraph()
```

## 3.2 The generated files

As you can see on the example of the code snipped above, a Bipolar Outranking Digraph can be exported with the help of the new *exportD3()* function or with the *showInteractiveGraph()* function, which automatically opens a new browser window with the newly generated files. Both functions generate these 4 files, which I am going to describe directly afterwards:

- d3export.json [outranking\_<graph name>.json]
- index.html [<graph name>.json]
- d3.v3.js
- digraph3lib.js

Where <graph name> is the name of the graph defined by Digraph3. The file name might also be followed by a ticket number in case a graph with the same name was already saved before inside the working directory.



**d3export.json**

If the datafile is generated by our *exportD3()* function, it contains all the data of the exported graph encoded in XMCD-2.0 format inside a JSON dictionary together with its pairwise comparisons. You will find more about the data structure in *Chapter 4*.

**index.html**

Contains all the HTML code needed to visualize the graph together with its special tools. In order to make our tools as pretty as possible I made use of Twitter Bootstrap v3.1.1<sup>1</sup>, an open-sourced front-end framework that contains a free collection of tools for creating web-sites and web applications, which makes it easy to produce good looking web applications.

**d3.v3.js - Changed**

Contains our modified D3 - Data driven Documents library. The only change that was necessary, was the one allowing us to the import of graphs in our own format, because the original D3 library only allows to import graphs with the following format:

```
{"source": 1 , "target" : 2}
```

Where the numbers 1 and 2 are the indexes of the nodes inside the JSON dictionary that is given to the D3 library in order to create the needed nodes and edges. I found out that this can be easily done modifying the *force.start()* function by adding the following lines:

```
if(typeof o.source == "string" || typeof o.target == "string") {  
    for (var k = 0; k < n; ++k) {  
if (nodes[k].name === o.source) links[i].source = nodes[k].index;  
        if (nodes[k].name === o.target) links[i].target = nodes[k].index;  
    }  
}
```

With the help of this small change we are now able to export a link directly from Digraph3 with the correct format, and do not have to care any more about indexes. The correct indexing is now done automatically in the background. This was the easiest and cleanest way possible.

```
{ "source": "node1" , "target" : "node2" }
```

---

<sup>1</sup><http://getbootstrap.com/>

### **digraph3lib.js**

Is the core file of our graph visualization, it contains all the needed code from import over graph editing to export of the edited graph. We will analyse it's implementation decisions later in this thesis in *Chapter 4 & 5*.

## **3.3 Possible interactions**

First versions of my project only allowed interactive inspection of XMCD-2.0 encoded graphs, but at the end it became clear that implementing a complete editor makes much more sense and would not be too much further work.

To better understand the concept of possible interactions, one has to know that there exist two types of graphs:

- outranking
- general

To see in which mode your editor currently, is you can use the '*Mode:*' label on the top-right corner of the canvas. Our editor allows you to create new graphs of type 'general' from scratch, import graphs from Digraph3 and export them as XMCD-2.0 encoded graphs of the type 'general'. You can import these graphs again into Digraph3 with the following commands:

```
> from digraphs import *  
> dg = XMCD2Digraph(fileName= "filename")
```

Both modes are quite different but have a few things in common.

In any case it is possible to drag nodes to a certain position where they are frozen (taken out of the physics engine) and can be released again. Also the inspection of single nodes together with their neighbours is possible.

### **Mode : 'general'**

If the graph is of type 'general', it is possible to add/edit/delete nodes and to add/edit/delete edges. It is not possible though to inspect an edge, since the pairwise comparisons table is generated by our Python-3.4 program and can only be calculated for graphs of the type 'outranking'.

**Mode : 'outranking'**

The actions that can be performed on an outranking graph are very limited, since these graphs are the result of long computations. As a consequence, it is not possible to add/edit/delete nodes or add/edit/delete/invert edges, since any change in the graph would result in another performance table and the values of the pairwise comparison would not be correct.



# Chapter 4

## Development - Data Structures

In this Chapter I explain the main decisions that we took during the development of our library, especially the import and export functions, as well as some other important details, like i.e. the data structure that I use.

### 4.1 XMCD-2.0

„XMCD is a data standard which allows to represent MultiCriteria Decision Analysis (MCD) data elements in XML according to a clearly defined grammar.” [4]

The logical Diagram of the XMCD-2.0 standard can be found on the web page of the Digraph3 project[1].

Every generated graph can be exported from our Digraph3 program formatted as a JSON dictionary and then be imported into our editor. Graphs of the type "general" can be exported encoded in the XMCD-2.0 format and then be imported inside Digraph3 again.

Structure examples of XMCD-2.0 files can be found inside the appendix.

### 4.2 Data-structure

I had to find a good data structure to import XMCD 2.0 encoded graphs into our editor in order to visualize them with help of the D3 Library. First it was not clear which was the right way to do this, so I just implemented everything in Python and only exported the minimum of data needed to feed the D3 library with input. As I had to find out later this was not the optimal way to do it since changes inside the D3 library would make a change in the

Digraph3 Python code needed. Both implementations should be as separated as possible. Therefore another approach had to be found.

What was needed is a way to export the graph and also keep the code clean, which means that changes in the external D3 library would not make a change in the Digraph3 code needed.

Therefore, the decision has been made to simply use the XMCD A-2.0 format that is provided by the already existing function *saveXMCD A2()* inside the Digraph3 Python program and export this XMCD A-2.0 encoded graph together with the pairwise comparison table inside a JSON array which is then saved as a JSON file inside the working directory.

This file can then be uploaded into our graph editor. The editor, with the help of JavaScript then parses the JSON file and generates another JSON array with the format that is needed by D3. This implementation allows us to clearly separate the editor and the Digraph3 program. If something inside the D3 library changes, the only thing that has to be changed is the JavaScript code that generates the page.

```
{
  "xmcd a2":
    "<xmcd a2 encoded graph>",
  "pairwiseComparisions":
    { "a1" :
      { "a1" : "<html code>",
        "a2": "<html code>",
        ..
      },
      "a2" :
        { "a1" : "<html code>",
          "a2": "<html code>",
          ..
        }
    }
}
```

As you can see the JSON dictionary is composed of the two keys *xmcd a2* and *pairwiseComparisions*. The *xmcd a2* value field contains the string representation of the XMCD A-2.0 encoded graph and the *pairwiseComparisions* value field contains the string representation

of all the comparisons tables between nodes encoded in HTML format.

## 4.3 Import

The import and export functions are very important points, because they allow us to transfer XMCD-2.0 encoded graphs from Digraph3 into the editor and vice versa. The import works if and only if the imported file is in the format that I described in the subsection Data Structure at the beginning of *Chapter 4*.

### 4.3.1 Graph type

In order to decide the current mode of our editor, we had to find a solution that includes the graph type inside our XMCD-2.0 encoded file. Since some fields have different meanings throughout Digraph3 and because I had to follow the XMCD-2.0 format specification, the decision was taken to use the field 'id=' of the element *projectReference*, which is usually the file name. Meaning that the editor is always initialized in general mode, except if the imported XMCD-2.0 file has the word "outranking" inside its *id* field.

```
<projectReference id="general_graphName" name="general">
```

### 4.3.2 How does it work?

The *importJSON()* function is called after selecting a file from an input field inside the import Modal and pressing the *Save* button.

The import consists of the following functions:

- *importJSON()*
- *parseXMCD2(xmlinput:String)*
- *buildD3Json(actions:Dictionary, relation :Dictionary)*
- *load()*
- *initialize()*
- *start()*

A sequence diagram has been added behind the now following textual description, at the end of this subsection.

**importJSON()**

This is the core function of our import procedure. It loads the file that was selected from the HTML input element and controls the whole process of parsing, building of the correct dictionary and loading the final graph. On load end, which is a function of the FileReader API that is included in every new browser, the function *parseXMCD2(xmlinput : String)* is called.

**parseXMCD2(xmlinput : String)**

Parses the XML input String and transforms it into an XML document that can be parsed, by already existing functions in JavaScript or JQuery. We parse this XML document for the important fields like the valuation domain, the graph type and the nodes and edges. The return value of the function *parseXMCD2(xmlinput : String)* is an array of the format *[actions,relation]*. Where *actions* is the array of nodes and *relation* is the array of relations between nodes.

```
actions = {
  "node id1" : { "name" : "node name" , "comment" : "node comment" } ,
  "node id2" : . . .
}

relation = {
  "node source1" : { "node target1" : value2 ,
                    "node target2" : value2 , ..},
  "node source2" : ...}
```

Those two variables are then forwarded to the *buildD3Json(actions,relation)* function.

**buildD3Json(actions : Dictionary, relation : Dictionary, hide : Boolean )**

Generates the right format to import nodes and edges into *D3.v3.js*.

The variable *hide* is defined by default *False*, but in case you want to hide the undefined connections *hide* needs to be set to *True*. To keep track of the status *hide* inside our graph I introduced the global variable *hide\_graph* which indicates if hiding is turned on or off.

By analysing the different values of the edges, every edge that connects two nodes has a certain type, which is defined as follows.



```

r(a,b) < Med & r(b,a) < Med  a      b : skip
r(a,b) > MAX & r(b,a) > MAX  a $    b :-1
r(a,b) > Med & r(b,a) < Med  a  --> b :0
r(a,b) < Med & r(b,a) > Med  a  <-- b :1
r(a,b) > Med & r(b,a) > Med  a <--> b :2
r(a,b) > Med & r(b,a) = Med  a o--> b :3
r(a,b) = Med & r(b,a) > Med  a <--o b :4
r(a,b) < Med & r(b,a) = Med  a o..  b :5
r(a,b) = Med & r(b,a) < Med  a  ..o b :6
r(a,b) = Med = r(b,a)        a o..o b :7

```

With:

```

r(a,b) : the relation from node a to node b
Med : the Median value of the valuation domain.
-- : Representantation of normal edge
... : Representation of stroked edge
o : Undefined arrow head
< : Defined arrow head
a $ b: The type used for new edges that have no value

```

Defining types for different edges made it easy to decide which edge type is needed and should be drawn.

### **load(hide : Boolean)**

Initializes the graph with help of the *initialize()* function and then calling the *buildD3Json(actions : Dictionary, relation : Dictionary ,hide : Boolean)* function and taking it's output, which is correctly D3 formatted, to replace the values of *force.links()* and *force.nodes()* needed by D3 to generate the graph. We do this since we are calling *load()* every time the graph might have changed.

### **initialize()**

Deletes the old SVG element from the HTML Dom and initializes a new empty graph editor window.

**start()**

Takes the data that we put inside the D3 *force.links()* and *force.nodes* and adds the according nodes and edges to the graph. It sets all the labels and at the the end of the start-up we call the D3 function *force.start()* which is used to start the D3 graph physics and starts to order the nodes in the best possible way.

### 4.3.3 Sequence Diagram

Here you can see the sequence diagram that has been made to better understand how the *importJSON()* function works.

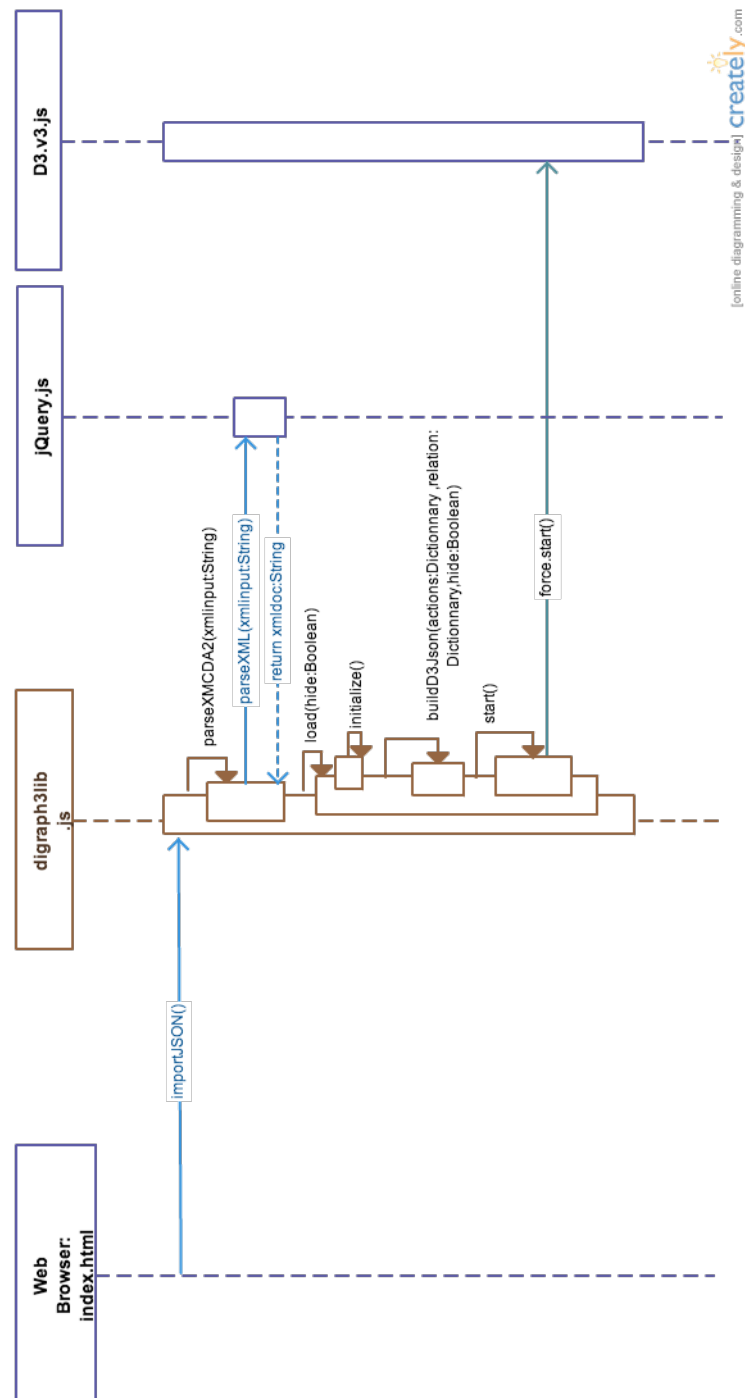


Fig. 4.1 Sequence Diagram - *importJSON()*

## 4.4 Export

### 4.4.1 How does it work?

The export can always be done and the graph can be imported into Digraph3 as a digraph with the function *XMCDa2Digraph()*.

When exporting a graph from our editor, we call the *exportXMCDa2(xmlinput : String)* function. The input of the export function is the output of *buildXMCDa2()* which takes both Dictionary's actions and relation and generates the last up to date version of the graph encoded as XMCDa-2.0.

To proof that the import inside Digraph3 is working, you can test the import function with the following code.

```
> from digraphs import *
> dg = XMCDa2Digraph(fileName = "yourfilename")
> dg.showAll()
```

The *showAll()* function displays all important details of the graph, i.e. the relation table or the neighbourhood. An example of the smallest possible non empty graph with one node can be found within the Appendix A.

All exported XMCDa-2.0 formats by my script have valid XML format, which can be checked with a XML validator, i.e. the W3 Validator<sup>1</sup>.

### 4.4.2 Problem

The action of saving a JavaScript variable to the hard drive is forbidden by most of the web browsers. The reason is that they are not allowing JavaScript to manipulate the file-system for security reasons. A workaround for this was to create a Blob object, which can be downloaded from the newly created *<a>* element inside the HTML DOM, which is initialized with the Blob object and other needed attributes. In order to keep the HTML Dom clean, we delete the *<a>* element directly after the download was initiated.

Since this kind of trick is also used by drive by attacks it might not be the cleanest solution, but it is the only working solution that works on all our supported web-browsers(*see known limitations*).

---

<sup>1</sup>W3 XML Validator: [http://www.w3schools.com/xml/xml\\_validator.asp](http://www.w3schools.com/xml/xml_validator.asp)

## 4.5 Conclusion

In order to separate the two implementations `digraph3lib.js` and `Digraph3` completely from each other, we decided to use the XMCD A-2.0 encoding of graphs that is provided by the `Digraph3` program with the function `saveXMCD A2()` and put this data together with the pairwise comparison tables that are generated by `Digraph3` into a JSON dictionary.

This way both the editor and `Digraph3` are completely separated and changes will only have impact on the changed program itself, since the XMCD A-2.0 is a standard for MCDA data elements.

Furthermore, a way has been found to import the graph and parse the XMCD A-2.0 string that we get from our JSON array. Therefore it is possible to visualize the graph together with its pairwise comparison tables that were generated by `Digraph3`, but only in case the graph is of the type outranking.

For the export we use a method that is also used by drive-by download attacks. Attackers are using this method to download malicious code onto the system without user interaction. This had to be done, because JavaScript is not allowed to manipulate the file system for security reasons. But as a positive result the export works in every common web browser.



# **Chapter 5**

## **Development - Visualization & Control**

In this Chapter I am going to describe the decision that were taken for the visualization and the control of the graph. The goal was to keep the graph editor as simple as possible, but on the other hand still provide all the needed features and keep the goal of better visualization in mind. The development of my editor is based partly on a code example that I found on a tutorial page[8] in order to learn how to use the D3 library and on the things that I learned from the W3Schools[9] tutorial. For the context-menu a jQuery plugin was used[5].

### **5.1 Visualization**

In this subsection I describe you the decisions that were taken for the design of nodes/edges which were done with the help of SVG.

#### **5.1.1 Edges**

First of all, a decision had to be made about the best possible way to represent edges and nodes in the most simplistic and user friendly way, in order to guarantee an overview over the graph at any moment in time. Therefore two implementations were tested, in the first one the edges had a curved form[6].

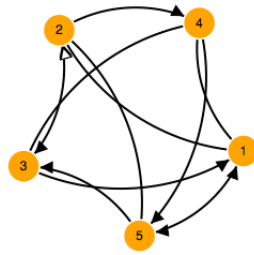


Fig. 5.1 Graph: Curved edges

The problem that I encountered with this implementation was that the graph force algorithms are not working very nicely in case of curved edges. And therefore the decision has been made to use the other implementation of straight edges. In which the overlapping of edges is not so frequent.

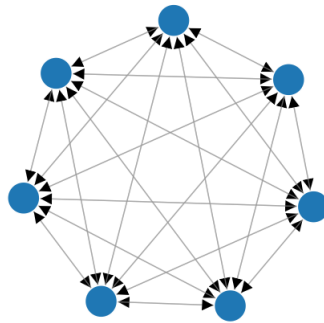


Fig. 5.2 Graph: Straight edges

Also the separation of edges and nodes done by the force algorithm is in the case of straight edges done in a more efficient way. Therefore, the decision has been made to use straight edges for the implementation of our editor.

### 5.1.2 Arrows

With the help of SVG, I also implemented all possible arrow/edge combinations. As described in the paragraph import, the type of the edge describes its later looks. Adding an arrow head to the edge was as easy as adding a marker element to the path element which represents the edge.



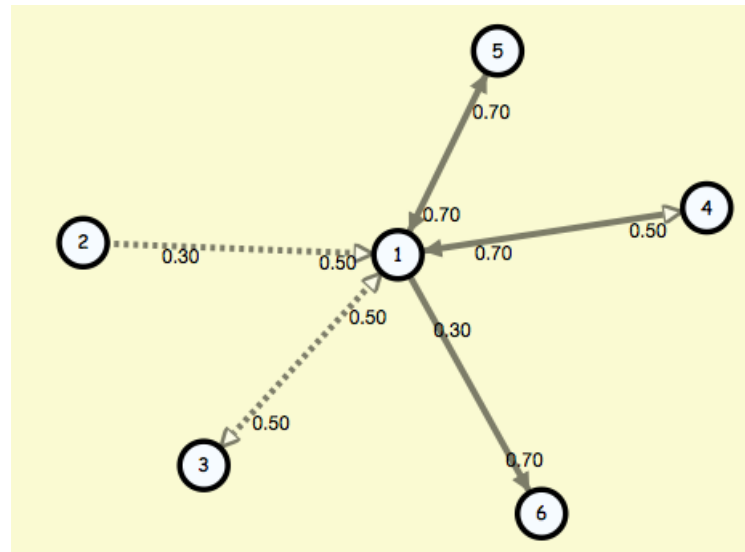


Fig. 5.3 Graph: Arrow and edge types

You can see here all possible edge and arrow combinations that are possible.

## 5.2 Control

In this section a short overview of the possible actions that you can perform on the graphs is given.

### 5.2.1 Background

The actions that can be performed with help of the context menu that appears by right click on the background are all the basic functions that one would try to perform inside an empty canvas or on the whole graph.

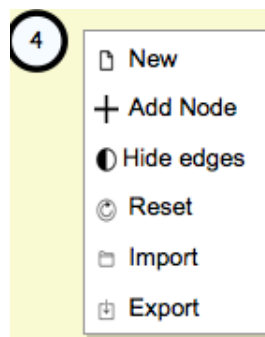


Fig. 5.4 Graph: Background Context-menu

### Hide edges

Choosing the function *Hide edges* from the background context menu will hide all undefined edges and redraw the graph in order to rearrange its nodes. You can make the edges visible again by choosing the *Unhide* function that becomes available in the background context-menu as soon as hiding was activated.

### Graph Freeze

In the latest version the graph is frozen after the tick count of 100 ticks (graph movement actions) was reached. The graph force can then be restarted by simple double clicking on the background. This was done because big graphs use nearly all resources that the browser offers on slower machines, and freezing the nodes frees those resources. The force algorithm can be restarted by a double-click on the background.

## 5.2.2 Nodes

The controls that are possible with a node reach from dragging to inspection. I will only give you a fast overview of the decisions that were taken and explain to you why I decided to use this kind of implementation. The other functions can be easily seen by using the editor. All the node actions can be done via the node context-menu that appears by a right-click on a node.

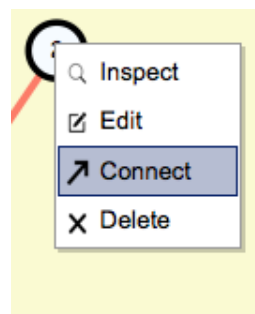


Fig. 5.5 Graph: Node Context-menu

### Dragging

Nodes can be dragged by a mouse-down command and a mouse-up freezes the node on the mouse-cursor position[7] until the node is released into the force again by a double-

click. This varies from the standard implementation where nodes are dragged and remain inside the force algorithm and thus are moved back again instantly after release. Like this it is possible to expand the graph and move nodes to a certain place. Dragging a node will freeze the whole graph, and thus makes it possible to rearrange the nodes as you want them to be.

### **Inspect**

It is also possible to inspect a node by right-clicking on it and selecting *Inspect* from the dialogue box. This enables the inspect view where non connected nodes and their edges are moved in the background and you are only able to see directly connected nodes, together with their connection to the inspected node.

Another way would have been to just visualize the node and it's neighbours in a "new" graph where all the other nodes are not shown, but this solution might lead to confusions since you are not able to see the other connections to the 2nd level connected nodes. An example for the Inspect function can be found in *Chapter 6*

### **Add/Delete/Edit**

If the graph is not of the type *outranking* then nodes can be added, deleted and edited.

Adding a node can be done by right clicking on the background and selecting *Add Node* from the context menu. The new node is then added to the force graph. Editing and deleting a node can be done by a right-click on the node itself. Editing only allows you to edit the *Name* and *Comment* value of a node.

## **5.2.3 Edges**

Edges can be controlled with the context-menu that is showing up after a right-click on the edge.

### **Add/Delete/Edit**

If the graph is not of the type *outranking* then edges can be added, deleted and edited.

Adding of an edge can be done by right clicking on a node and selecting the connect function from the context menu. The connect function has not been implemented by click and drag,

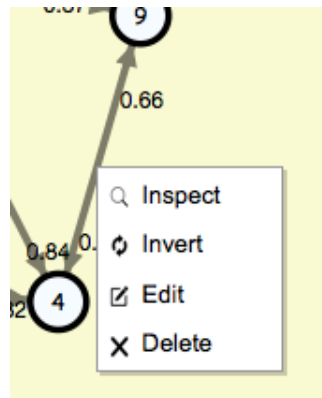


Fig. 5.6 Graph: Edge Context-menu

because on big graphs it will be easier to just click on a node and selecting the node that you want to connect to from a list, one by one.

### Inspect

If the graph is of the type *outranking*, an edge can be inspected by simple right clicking on the edge and selecting the *Inspect* function from the context menu. Inspecting an edge will show you the pairwise comparison table of it's source and target node.

### Invert

Choosing the *Invert* function from the context menu will provide you with the invert function. Inverting an edge means that you invert his value around the median. This is only possible for graphs of the type *general*. The formula to do this is the following:

```
valuationdomain["Max"] - relation[d.source.name][d.target.name]
+ valuationdomain["Min"]
```

The graph is afterwards redrawn with help of the *load()* function in order to rearrange the nodes in case the edge is not shown due to it's new values.

## 5.3 Conclusion

Making software interaction user friendly is not easy. Everyone has his own preferences and there are a lot of ways in which the interaction or visualization can be done. Throughout the project there were many changes in the design and also in the user interaction which cost a lot of development time, because the JavaScript code had to be rewritten nearly from scratch in order to test other designs or other interaction methods. But at the end I was able to consider the broad range of uses and allow users to interact in the most productive way with regard to graphs of 10-35 Nodes.



# Chapter 6

## Conclusion

In this chapter are the results and achievements throughout the project, some future extension that could be implemented and a small note about the personal experiences that I made during the project.

### 6.1 Achievements

The goal of this thesis was to find a better way to display outranking digraphs, meaning that the user should be able to interact with the graph and it should be easier for him to analyse the different value for the connections between edges. For this, functions have been developed that allow the export of the Digraph3 generated graphs in a special format that is needed by our graph editor in order to reproduce the graphs. Furthermore we are also able to export self made graphs and import them again into our Digraph3 program where we are able to continue working on them.

To proof the concept I generated an outranking digraph of 30 nodes, exactly like the one that I showed you in the Analysis Chapter. To compare both visualisations equally and fair, the case of big graphs is tested, because they are the hardest case and show perfectly what improvements were made to visualize those graphs.

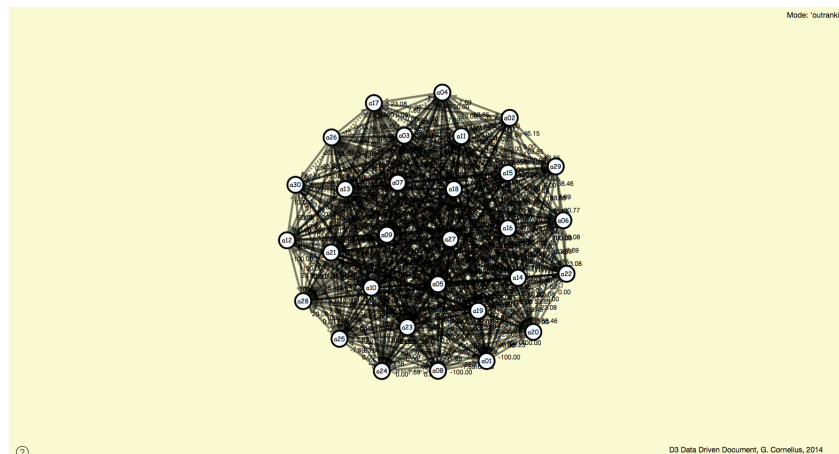


Fig. 6.1 Graph: Outranking Example

As you can see the problem that we encounter on big graphs is still the same one as described already in the Analysis part, this is due to fact that those examples are both examples of highly connected graphs with the limitations that will be described in the paragraph *Known limitations*. But with help of our visualisation tool we are able to inspect certain edges, which provides us with much more overview over the graph.

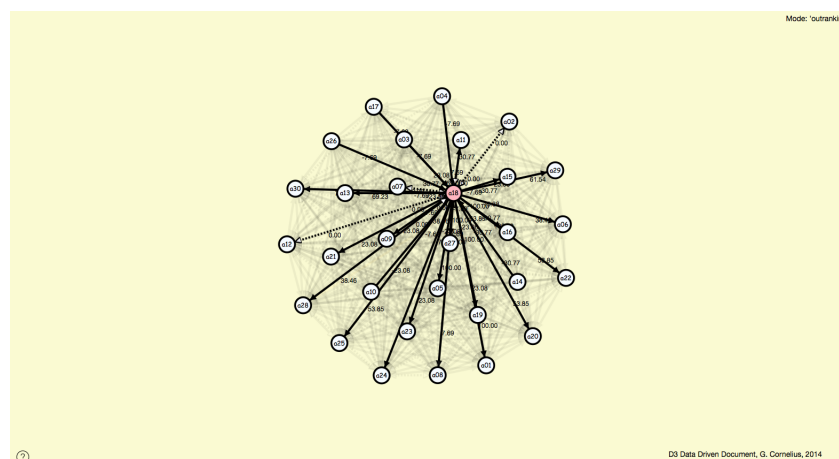


Fig. 6.2 Graph: Outranking Inspect Example

You can see that now the different connections become much easier to understand. Both examples were done without dragging nodes around or manipulating the graph manually, which would result in an even better view of the graph.



## 6.2 Known limitations

### 6.2.1 Browser Support

To support all the browsers was one of the hardest tasks, because there are big differences in their behaviours and also in the implementations of standards. Internet Explorer for example, is not supporting the marker-ends (Arrows) on lines, which destroys your graph completely. This is a bug inside Internet Explorer and cannot be fixed by my side in any way. Therefore it is highly recommended you not to use Internet Explorer if you want to work with our editor, but to consider using Chrome, FireFox, Safari or Opera.

Another point is that some browsers don't allow local files to be loaded by default from the browser if no header is send by a web-server for this file. If you are using Firefox, the automatic load of the web-page together with it's data file is working just fine(one warning). But if you are using another browser I highly recommend you to execute the following python command inside the working directory and access the page via the localhost.

```
python -m SimpleHTTPServer
```

Access: [http://0.0.0.0:8000/file\\_name.html](http://0.0.0.0:8000/file_name.html)

### 6.2.2 Performance big graphs

Another problem encountered at the end was that for graphs that are too connected, the browser reaches it's limits. If the force between the nodes is enabled, the editor can barely be used. This happens because all the coordinates of nodes, edges and labels are updated by the tick function.


Process Name	% CPU ▾	CPU Time	Threads	Idle Wake Ups	PID	User
 Firefox	93.5	4:11.29	47	7	42295	garycorn

Fig. 6.3 Graph: Performance on load

Therefore, the double-click function was implemented on the background in order to be able to freeze and unfreeze the graph manually. Furthermore, a tick limit has been implemented, which freezes the graph automatically after 100 ticks. Having as a result that after the graph is frozen, the editor can be used again without any performance issues.

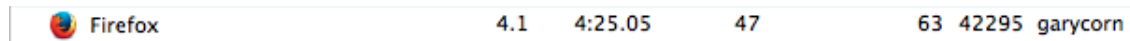


Fig. 6.4 Graph: Performance on idle

As you can see, after freezing the graph the CPU usage is dropping dramatically. Best performance was experienced with *Google Chrome*.

## 6.3 Further developments

Further developments could be the implementation of a better graph visualization algorithm and also to improve the look of our editor. Improving the graph visualization though will be a hard task, because our graphs are mostly heavily connected and visualizing graphs without edge crossing is an NP-Hard problem and I am not sure our browser can support such expensive calculations, regarding the known issue of performance for big graphs that we are already encountering.

## 6.4 Experience

During my project the most surprising experience that I was able to make was about time management. I learned that it takes some time to be able to handle a new library and programming language. In order to be able to program freely, following tutorials is not enough. The time factor was sometimes the thing that surprised me the most, because for really small problems I sometimes needed a few days to solve them. Whereas bigger problems, where I planned a few days, were sometimes done in a few hours.

Learning about Digraph3, reading code and understanding what the program is doing and what functions exist was another challenge that was very interesting for me.

# References

- [1] Bisdorff, R. (2009). Documentation for xmcda-2.0.0.  
URL: <http://ernst-schroeder.uni.lu/UMCDA-ML-2.0/doc/>.
- [2] Bisdorff, R. (2013-2014a). Digraph3 documentation.  
URL: <http://leopold-loewenheim.uni.lu/Digraph3/docSphinx/html/index.html>.
- [3] Bisdorff, R. (2013-2014b). Digraph3 project page.  
URL: <http://leopold-loewenheim.uni.lu/>.
- [4] DecisionDeck (2009-2013). Decision deck project.  
URL: <http://www.decision-deck.org/xmcda/index.html>.
- [5] Domigan, C. (2007). jquery context-menu.  
URL: <http://www.trendskitchens.co.nz/jquery/contextmenu/jquery.contextmenu.r2.js>.
- [6] mbostock (2013a). Directed graph curved edges.  
URL: <http://bl.ocks.org/mbostock/1153292>.
- [7] mbostock (2013b). Sticky force layout.  
URL: <http://bl.ocks.org/mbostock/3750558>.
- [8] rkirsling (2013). Directed graph editor.  
URL: <http://bl.ocks.org/rkirsling/5001347>.
- [9] W3Schools (1999-2014). Svg tutorial.  
URL: <http://www.w3schools.com/svg/default.asp>.



# **Appendix A**

## **Example Collection**

I collected a few examples on the web page that was given to me in order to upload needed resources. You can find everything following the URL

<http://leopold-loewenheim.uni.lu/WWGary/>

