GEXF 1.1draft Primer

Gephi project

May 31, 2010

Abstract

GEXF Primer is a non-normative document intended to provide an easily readable description of the GEXF facilities, and is oriented towards quickly understanding how to create GEXF documents. This primer describes the language features through examples which are complemented by references to normative texts. Specification is in RelaxNG Compact grammar.

Contents

1 Introduction

This document, GEXF Primer, provides an description of GEXF, and should be used alongside the formal descriptions of the language contained in the GEXF specification. The intended audience of this document includes application developers whose programs read and write GEXF files, and users who want to communicate with programs using GEXF import/export. The text assumes that you have a basic understanding of XML 1.0 and XML-Namespaces. Basic knowledge of XML Schema is also assumed for some parts of this document. Each major section of the primer introduces new features of the language, and describes those features in the context of concrete examples.

Section 2 covers the basic mechanisms of GEXF. It describes how to declare a simple graph by defining its nodes and edges and how to add simple user data to the graph.

Section 3 describes dynamic graph model.

Section 4 describes mechanisms for extending GEXF to add specific data with the Visualization module in example.

The primer is a non-normative document, which means that it does not provide a definitive specification of the GEXF language. The examples and other explanatory material in this document are provided to help you understand GEXF, but they may not always provide definitive answers. In such cases, you will need to refer to the GEXF specification, and to help you do this, we provide many links pointing to the relevant parts of the specification.

2 Basic Concepts

The purpose of a GEXF document is to define a graph representing a network. Let us start by considering the minimal graph shown in the figure below. It contains 2 nodes and 1 edge.

2.1 A Simple Graph

This is a dummy graph:

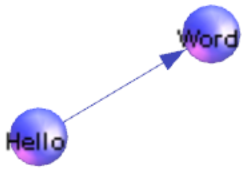

Figure 1: Hello-world graph

Listing 1: Hello world!

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.1draft"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.gexf.net/1.1draft
                            http://www.gexf.net/1.1draft/gexf.xsd"
      version="1.1">
  <meta lastmodifieddate="2009-03-20">
    <creator>Gephi.org</creator>
    <description>A hello world! file</description>
  </meta>
  <graph defaultedgetype="directed">
    <nodes>
      <node id="0" label="Hello"/>
      <node id="1" label="Word"/>
    </nodes>
    <edges>
      <edge id="0" source="0" target="1"/>
    </edges>
  </graph>
</gexf>
```

The GEXF document consists of a gexf element and a variety of subelements: graph, node, edge. In the remainder of this section we will discuss these elements in detail and show how they define a graph.

2.2 Header

In this section we discuss the parts of the document which are common to all GEXF documents, basically the gexf element and the meta declaration.

Listing 2: Header

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.1draft"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.gexf.net/1.1draft
                            http://www.gexf.net/1.1draft/gexf.xsd"
      version="1.1">
  <meta lastmodifieddate="2009-03-20">
    <creator>Gephi.org</creator>
    <description>A hello world! file</description>
    <keywords>basic, web</keywords>
  </meta>
  ...
</gexf>
```

The first line of the document is an XML process instruction which defines that the document adheres to the XML 1.0 standard and that the encoding of the document is UTF-8, the standard encoding for XML documents. Of course other encodings can be chosen for GEXF documents.

The second line contains the root-element element of a GEXF document: the gexf element. The gexf element, like all other GEXF elements, belongs to the namespace http://www.gexf.net/1.1draft. For this reason we define this namespace as the default namespace in the document by adding the XML Attribute xmlns=”http://www.gexf.net/1.1draft” to it. The two other XML Attributes are needed to specify the XML Schema for this document. In our example we use the standard schema for GEXF documents located on the gexf.net server. The first attribute, xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”, defines xsi as the XML Schema namespace. The second attribute, xsi:schemaLocation =”http://www.gexf.net/1.1draft http://www.gexf.net/1.1draft/gexf.xsd”, defines the XML Schema location for all elements in the GEXF namespace.

The XML Schema reference is not required but it provides means to validate the document and is therefore strongly recommended.

The meta element contains additionnal information about the network. Element leafs are assumed to be text, and lastmodifieddate is an international standard date (yyyy-mm-dd).

GEXF document is specified in the RelaxNG Compact file gexf.rc. Header is ruled by the following declaration :

Listing 3: Header Specification

```
default namespace = "http://www.gexf.net/1.1draft"
namespace rng = "http://relaxng.org/ns/structure/1.0"
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"

## Grammar root
start = GEXF
Version = attribute version { string "1.1" }

## Tree
GEXF = element gexf { Meta?, Graph, Version, Variant? }
Meta = element meta { Creator?, Keywords?, Description?, LastModifiedDate? }

## Element leafs
Creator = element creator { text }
Description = element description { text }
Keywords = element keywords { text }

## Attribute leafs
LastModifiedDate = attribute lastmodifieddate { xsd:date }
```

2.3 Network Topology

The network topology structure containing nodes and edges is called the graph. A graph is, not surprisingly, denoted by a graph element. Nested inside a graph element are the declarations of nodes and edges. A node is declared with the node element inside a nodes element, and an egde with the edge element inside an edges element. Nodes and edges order doesn't matter.

Listing 4: The definition of the graph

```
<graph defaultedgetype="directed">
  <nodes>
    <node id="0" label="Hello" />
    <node id="1" label="Word" />
    ...
  </nodes>
  <edges>
    <edge id="0" source="0" target="1" weight="3.167" />
    ...
  </edges>
</graph>
```

2.3.1 Declaring a Graph

Graphs in GEXF are mixed, in other words, they can contain directed and undirected edges at the same time. If no direction is specified when an edge is declared, the default direction defaultedgetype is applied to the edge. If you know what kind of edges are stored, you may interpret the mixed graph as a directed or an undirected graph at your own risks.

The default direction is declared as the optional XML-attribute defaultedgetype of the graph element. The three possible values for this XML-attribute are *directed*, *undirected* and *mutual*. Note that the default direction is optional and would be assumed as *undirected*.

The optional XML-attribute mode set the kind of network: static or dynamic. Last one provides time support (see the section 5 on Dynamics). Static mode is assumed by default.

Listing 5: An empty graph!

```
<graph>
  <nodes>
  </nodes>
  <edges>
  </edges>
</graph>
```

Listing 6: Topology Specification

```
Graph = element graph { Nodes, Edges, Mode?, DefaultEdgeType? }
Node = element node { ID, Label, ... }
Edge = element edge { ID, Source, Target, Label?, Cardinal?, EdgeType? }

## Attribute leafs
DefaultEdgeType = [ a:defaultValue = "undirected" ] attribute defaultedgetype {
    string "directed" |
    string "undirected" |
% string "mutual"
}
Mode = [ a:defaultValue = "static" ] attribute mode {
    string "static" |
    string "dynamic"
}
...
```

2.3.2 Declaring a Node

Nodes in the graph are declared by the node element. Each node has an identifier, which must be unique within the entire document, i.e., in a document there must be no two nodes with the same identifier. The identifier of a node is defined by the XML-attribute id, which is a string. Each node must have a XML-attribute label, which is a string.

Listing 7: A node!

```
<node id="0" label="Hello world" />
```

Listing 8: Node Specification

```
Node = element node { ID, Label }

## Attribute leafs
ID = attribute id { xsd:string }
Label = attribute label { xsd:token }
```

2.3.3 Declaring an Edge

Edges in the graph are declared by the edge element. Each edge must define its two endpoints with the XML-Attributes source and target. The value of the source, resp. target, must be the identifier of a node in the same document. The identifier of an edge is defined by the XML-Attribute id. There is no order notion applied to edges.

Edges with only one endpoint, also called loops, selfloops, or reflexive edges, are defined by having the same value for source and target.

Each edge can have a optional XML-attribute label, which is a string.

The optional XML-attribute type declares if the edge is *directed*, *undirected* or *mutual* (directed *from source to target and from target to source*). If the direction is not explicitely defined, the default direction is applied to this edge as defined in the enclosing graph.

The weight of the edge is set by the optional XML-attribute weight and is a float.

Assuming two nodes having respectively the id value set to *0* and *1*:

Listing 9: An edge!

```
<edge id="0" source="0" target="1"/>
```

Listing 10: A more complete edge

```
<edge id="0" source="0" target="1" type="directed" weight="2.4" />
```

Listing 11: Edge Specification

```
Edge = element edge { ID, Source, Target, Label?, Weight?, EdgeType? }

## Attribute leafs
EdgeType = [ a:defaultValue = "undirected" ] attribute type {
   string "directed" |
```

```
    string "undirected" |
    string "mutual"
}
ID = attribute id { xsd:string }
Label = attribute label { xsd:token }
Source = attribute source { xsd:string } # ref on ID
Target = attribute target { xsd:string } # ref on ID
Weight = [ a:defaultValue = "1.0" ] attribute weight { xsd:float }
```

2.4 Network Data

In the previous section we discussed how to describe the topology of a graph in GEXF. While pure topological information may be sufficient for some applications, these days focus is made on network analysis based on data attributes. Data are everywhere.

A bunch of data can be stored within attributes. The concept is the same as table data or SQL. An attribute has a title/name and a value. Attribute's name/title must be declared for the whole graph. It could be for instance "degree", "valid" or "url". Besides the name of the attribute a column also contains the type.

2.4.1 Data types

GEXF uses the XML Schema Data Types (XSD 1.1) for the following primitives: string, integer, float, double, boolean, date, and anyURI.

2.4.2 Attributes Example

Each Node of this graph has three attributes : an url, an indegree value and a boolean for french websites which is set to *true* by default.

Listing 12: A (small) Web Graph

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.1draft"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.gexf.net/1.1draft
                           http://www.gexf.net/1.1draft/gexf.xsd"
      version="1.1">
  <meta lastmodifieddate="2009-03-20">
    <creator>Gephi.org</creator>
    <description>A Web network</description>
  </meta>
  <graph defaultedgetype="directed">
    <attributes class="node">
      <attribute id="0" title="url" type="string"/>
      <attribute id="1" title="indegree" type="float"/>
      <attribute id="2" title="frog" type="boolean">
        <default>true</default>
      </attribute>
    </attributes>
    <nodes>
      <node id="0" label="Gephi">
        <attvalues>
          <attvalue for="0" value="http://gephi.org"/>
          <attvalue for="1" value="1"/>
        </attvalues>
      </node>
      <node id="1" label="Webatlas">
        <attvalues>
```

```
          <attvalue for="0" value="http://webatlas.fr"/>
          <attvalue for="1" value="2"/>
        </attvalues>
      </node>
      <node id="2" label="RTGI">
        <attvalues>
          <attvalue for="0" value="http://rtgi.fr"/>
          <attvalue for="1" value="1"/>
        </attvalues>
      </node>
      <node id="3" label="BarabasiLab">
        <attvalues>
          <attvalue for="0" value="http://barabasilab.com"/>
          <attvalue for="1" value="1"/>
          <attvalue for="2" value="false"/>
        </attvalues>
      </node>
    </nodes>
    <edges>
      <edge id="0" source="0" target="1"/>
      <edge id="1" source="0" target="2"/>
      <edge id="2" source="1" target="0"/>
      <edge id="3" source="2" target="1"/>
      <edge id="4" source="0" target="3"/>
    </edges>
  </graph>
</gexf>
```

2.4.3 Declaring Attributes

Attributes are declared inside an attributes element. The XML-attribute class apply nested attributes on nodes (*node* value) or edges (*edge* value). You may specify the data type between *integer, double, float, boolean, string and list-string*, and specify a default value.

Listing 13: Attributes Definition

```
<graph mode="static">
  <attributes class="node">
    <attribute id="0" title="my-text-attribute" type="string"/>
    <attribute id="1" title="my-int-attribute" type="integer"/>
    <attribute id="2" title="my-bool-attribute" type="boolean"/>
  </attributes>
  <attributes class="edge">
    <attribute id="0" title="my-float-attribute" type="float">
      <default>2.0</default>
    </attribute>
  </attributes>
  ...
</graph>
```

Listing 14: Attributes Specification

```
Attributes = element attributes { Attribute*, Class, Mode? }
Attribute = element attribute { ID, Title, AttrType, Default?, Options? }

Class = attribute class {
    string "node" |
    string "edge"
}
Mode = [ a:defaultValue = "static" ] attribute mode {
    string "static" |
    string "dynamic"
}
AttrType = attribute type {
    string "integer" |
```

```
    string "long" |
    string "double" |
    string "float" |
    string "boolean" |
    string "liststring" |
    string "string" |
    string "anyURI"
}
Default = element default { text }
Options = element options { text }
```

2.4.4 Defining Attribute Values

You may understand attributes while looking at this node definition. Besides native fields (id, label), node values are set for three attributes. Omitting an attribute will set the default value as its value. If no default value is set, this is an error.

Listing 15: Node Attributes

```
<node id="0" label="Hello world">
  <attvalues>
    <attvalue for="0" value="samplevalue"/>
    <attvalue for="1" value="1831"/>
    <attvalue for="2" value="true"/>
  </attvalues>
</node>
```

Listing 16: Edge Attributes

```
<edge id="0" source="0" target="1">
  <attvalues>
    <attvalue for="0" value="1.5"/>
  </attvalues>
</edge>
```

Listing 17: Attribute Values Specification

```
Attvalues = element attvalues { Attvalue* }
Attvalue = element attvalue { FOR, Value }

FOR = attribute for { xsd:string }
Value = element value { text }
```

Note about the *liststring* type: A liststring replaces the usage of multiple boolean attributes. Instead of declaring the attributes *foo*, *bar* and *foobar*, you just only have to declare *my-foobar*. *my-foobar* may takes the values *foo*, *bar*, *foobar*, *foo;bar*, *foobar;foo* etc. So the value *foobar;foo* is equivalent to an attribute *foobar=true* and *foo=true*.

Liststring gives the element values separated by a pipe, a comma or a semi-colon. This is an unsafe type! Liststring values are therefore parsed, and this parsing don't take any escape character like quotes or double-quotes into account. You have to check your data before making a GEXF file.

Listing 18: Liststring Definition

```
<graph mode="static">
  <attributes class="node">
```

```
    <attribute id="0" title="my-liststring-attribute" type="liststring">
      <default>foo|bar|foobar</default>
    </attribute>
  </attributes>
  ...
</graph>
```

Listing 19: Liststring usage

```
<node id="0" label="Hello world">
  <attvalues>
    <attvalue for="0" value="foobar|bar"/>
  </attvalues>
</node>
```

A complete example:

Listing 20: Boolean version

```
<attributes>
 <attribute id="0" title="hobby" type="liststring">
   <options>ski|dance|photo</options>
  </attribute>
</attributes>
<nodes>
 <node id="42" label="a node">
       <attvalues>
           <attvalue for="0" value="dance|ski">
       </attvalues>
   </node>
</nodes>
```

Listing 21: Liststring version

```
<attributes>
 <attribute id="0" title="hobby-ski" type="boolean" />
 <attribute id="1" title="hobby-dance" type="boolean" />
 <attribute id="2" title="hobby-photo" type="boolean" />
</attributes>
<nodes>
 <node id="42" label="a node">
       <attvalues>
           <attvalue for="0" value="true">
           <attvalue for="1" value="true">
           <attvalue for="2" value="false">
       </attvalues>
   </node>
</nodes>
```

Note about the attribute *options*: it defines the available values, separated by a pipe. It is both used as a type constraint and for parser optimization. The combined default value must be an available option, like the following example.

Listing 22: Options

```
<graph mode="static">
  <attributes class="node">
    <attribute id="0" title="my-string-attribute" type="string">
        <default>foo</default>
        <options>foo|bar|foobar</options>
    </attribute>
    <attribute id="1" title="my-integer-attribute" type="integer">
        <default>5</default>
        <options>1|2|5|6</options>
```

```
    </attribute>
  </attributes>
  ...
</graph>
```

When it is applied to a liststring attribute, it gives all possible elements of the list:

Listing 23: Valid values

```
<attributes>
 <attribute id="0" title="foo-attr" type="liststring">
   <options>foo1|foo2|foo3</options>
 </attribute>
</attributes>
<nodes>
 <node id="42" label="node A">
       <attvalues>
           <attvalue for="0" value="foo3|foo2">
       </attvalues>
   </node>
 <node id="43" label="node B">
       <attvalues>
           <attvalue for="0" value="">
       </attvalues>
   </node>
 <node id="44" label="node C">
       <attvalues>
           <attvalue for="0" value="foo1|foo2|foo3">
       </attvalues>
   </node>
</nodes>
```

Listing 24: Invalid values

```
...
<node id="42" label="node A">
   <attvalues>
       <attvalue for="0" value="foo1|foo4">
   </attvalues>
</node>
```

3 Advanced Concepts I: Hierarchy structure

3.1 Introduction

GEXF format allows creating hierarchical graph structure essentially for representing clustering. We modelize both a tree structure of ancestors and descendents, and a flat graph of nodes bound by edges.

Two ways are available:

1. Nodes can simply host other nodes and so on.
2. Each node refer to a parent node id with the XML-attribute pid.

The first style is preferred when the structure written is previously ordered. Sequential reading of this kind of GEXF is safe because no node reference is used. But in the case your program can't provide this, the second way allows

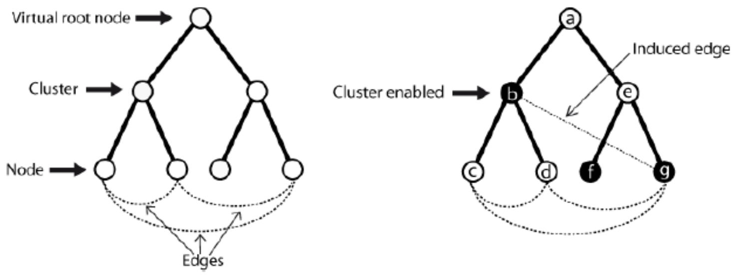

Figure 2: Graph tree with a virtual edge from a cluster to a leaf

writing (and then reading) nodes randomly, but linear reading is at your own risks.

3.2 Sequential-safe Reading

Listing 25: First way

```
<graph mode="static" defaultedgetype="directed">
    <nodes>
      <node id="a" label="Kevin Bacon">
        <nodes>
          <node id="b" label="God">
            <nodes>
              <node id="c" label="human1"/>
              <node id="d" label="human2"/>
            </nodes>
          </node>
          <node id="e" label="Me">
            <nodes>
              <node id="f" label="frog1"/>
              <node id="g" label="frog2"/>
            </nodes>
          </node>
        </nodes>
      </node>
    </nodes>
    <edges>
        <edge id="0" source="b" target="e" />
        <edge id="1" source="c" target="d" />
        <edge id="2" source="g" target="b" />
        <edge id="3" source="f" target="a" />
    </edges>
</graph>
```

Note that edges are not necessarily written at the end:

Listing 26: First way with edges inside clusters

```
<graph mode="static" defaultedgetype="directed">
    <nodes>
      <node id="a" label="Kevin Bacon">
        <nodes>
          <node id="b" label="God">
            <nodes>
              <node id="c" label="human1"/>
              <node id="d" label="human2"/>
            </nodes>
            <edges>
```

```
                <edge id=”0” source=”c” target=”d” />
              </edges>
            </node>
            <node id=”e” label=”Me”>
              <nodes>
                <node id=”f” label=”frog1”/>
                <node id=”g” label=”frog2”/>
              </nodes>
            </node>
          </nodes>
          <edges>
            <edge id=”1” source=”b” target=”e” />
            <edge id=”3” source=”f” target=”a” />
            <edge id=”2” source=”g” target=”b” />
          </edges>
        </node>
      </nodes>
      <edges />
    </graph>
```

3.3 Random Writing

If you can't structurize your graph topology before writing a GEXF file, you may use the second style. Nodes sent to Gephi from a live data source, i.e. a web crawler, are written like this. Note that edges are always written randomly.

Listing 27: Second way

```
<nodes>
  <node id=”a” label=”Kevin Bacon” />
  <node id=”b” label=”God” pid=”a” />
  <node id=”c” label=”human1” pid=”b” />
  <node id=”d” label=”human2” pid=”b” />
  <node id=”e” label=”Me” pid=”a” />
  <node id=”f” label=”frog1” pid=”e” />
  <node id=”g” label=”frog2” pid=”e” />
</nodes>
```

With using pid, node order doesn't matter. An implementation should manage the case when a node reference (pid) is used before the node declaration. This listings could also be:

Listing 28: Second way randomized

```
<nodes>
  <node id=”g” label=”frog2” pid=”e” />
  <node id=”a” label=”Kevin Bacon” />
  <node id=”c” label=”human1” pid=”b” />
  <node id=”b” label=”God” pid=”a” />
  <node id=”e” label=”Me” pid=”a” />
  <node id=”d” label=”human2” pid=”b” />
  <node id=”f” label=”frog1” pid=”e” />
</nodes>
```

4 Advanced Concepts II: Phylogeny structure

Multiple parents can be adressed with the following syntax, where a and b are c's parents:

Listing 29: Multiple parents

```
<nodes>
  <node id="a" label="cheese">
  <node id="b" label="cherry">
  <node id="c" label="cake">
    <parents>
      <parent for="a" />
      <parent for="b" />
    </parents>
  </node>
</nodes>
```

5 Advanced Concepts III: Dynamics

As networks dynamics is a growing topic of research, GEXF format includes time support. Enable it by setting the mode attribute of the graph to *dynamic*.

Listing 30: Dynamic Enabled!

```
<graph mode="dynamic">
  ...
</graph>
```

Time in GEXF is encoded in two ways. Continuous by default, it is encoded as an international standard date (yyyy-mm-dd). Discrete, it is a float. Use the the XML-attribute *timetype* of the graph element to explicitly declare the type: *float* or *date*.

Both network topology and data have a lifetime. The hole graph, each node, each edge and their respective data values may have time limits, beginning with an XML-attribute start and ending with a end. Omitting start set the past of the thing to infinity, so as for end. The file creator is responsible for the dates consistency.

5.1 Example

Listing 31: A (small) Dynamic Web Graph with continuous time

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.1draft"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.gexf.net/1.1draft
                           http://www.gexf.net/1.1draft/gexf.xsd"
      version="1.1">
  <meta lastmodifieddate="2009-03-20">
    <creator>Gephi.org</creator>
    <description>A Web network changing over time</description>
  </meta>
  <graph mode="dynamic" defaultedgetype="directed"
         start="2009-01-01" end="2009-03-20">
    <attributes class="node" mode="static">
      <attribute id="0" title="url" type="string"/>
      <attribute id="1" title="frog" type="boolean">
        <default>true</default>
      </attribute>
    </attributes>
    <attributes class="node" mode="dynamic">
      <attribute id="2" title="indegree" type="float"/>
    </attributes>
    <nodes>
      <node id="0" label="Gephi" start="2009-03-01">
        <attvalues>
```

```
        <attvalue for="0" value="http://gephi.org"/>
        <attvalue for="2" value="1"/>
      </attvalues>
    </node>
    <node id="1" label="Webatlas">
      <attvalues>
        <attvalue for="0" value="http://webatlas.fr"/>
        <attvalue for="2" value="1" end="2009-03-01"/>
        <attvalue for="2" value="2" start="2009-03-01" end="2009-03-10"/>
        <attvalue for="2" value="1" start="2009-03-11"/>
      </attvalues>
    </node>
    <node id="2" label="RTGI" end="2009-03-10">
      <attvalues>
        <attvalue for="0" value="http://rtgi.fr"/>
        <attvalue for="2" value="0" end="2009-03-01"/>
        <attvalue for="2" value="1" start="2009-03-01"/>
      </attvalues>
    </node>
    <node id="3" label="BarabasiLab">
      <attvalues>
        <attvalue for="0" value="http://barabasilab.com"/>
        <attvalue for="1" value="false"/>
        <attvalue for="2" value="0" end="2009-03-01"/>
        <attvalue for="2" value="1" start="2009-03-01"/>
      </attvalues>
    </node>
  </nodes>
  <edges>
    <edge id="0" source="0" target="1" start="2009-03-01"/>
    <edge id="1" source="0" target="2"
          start="2009-03-01" end="2009-03-10"/>
    <edge id="2" source="1" target="0" start="2009-03-01"/>
    <edge id="3" source="2" target="1" end="2009-03-10"/>
    <edge id="4" source="0" target="3" start="2009-03-01"/>
  </edges>
 </graph>
</gexf>
```

5.2 Dynamic Topology

Time limits declared for a graph element are optional, however they could save pre-importing computation. Time limits of edges must be consistent with the related nodes'ones.

The graph scope is defined as follow for a network from 2009-01-01 to 2009-03-20:

Listing 32: Graph Scope Example

```
<graph mode="dynamic" start="2009-01-01" end="2009-03-20">
```

Each edge must declare time limits inside the join scope of source and target:

- edge.start $\leq$ (source.start and target.start)
- edge.end $\geq$ (source.end and target.end)

Listing 33: Edge Scope Example

```
<nodes>
  <node id="0" label="Hello" start="2009-01-01" end="2009-02-01" />
  <node id="1" label="World" start="2009-01-15" end="2009-03-20" />
  ...
</nodes>
```

```
<edges>
  <edge id="0" source="0" target="1" start="2009-01-20" end="2009-02-01"/>
</edges>
```

Important: start and end values are inclusive, i.e. the following line is allowed:

Listing 34: Smallest time scope

```
<edge id="0" source="0" target="1" start="2009-01-20" end="2009-01-20"/>
```

And of course the end value must be later than the start value.

If a node or an edge exists only at some timeranges, we use the concept of slices described by Skye Bender-deMoll and Daniel A. McFarland in The Art and Science of Dynamic Network Visualization. Slices are not provided for data values, which are only limited by one start and one end. Use the xml-element slices for topology like this:

Listing 35: Node with multiple slices

```
<gexf ...>
  ...
  <graph mode="dynamic">
    <node id="0" label="Hello">
      <slices>
        <slice start="2009-01-01" end="2009-01-15" />
        <slice start="2009-01-30" end="2009-02-01" />
      </slices>
    </node>
    ...
  </graph>
</gexf>
```

If the xml-attributes start and end are used in node like before, they should be ignored by parsers: if slices are provided, only their content are taken into account. If no start is provided, the slice begins with the network. If no end is provided, the slice ends with the network. If two slices are covering a same period of time, parsers should consider them as a unique slice.

About the weight: dynamic weight can be used with the reserved *title* "weight" in attributes. In dynamic mode, the static XML-attribute *weight* should be ignored if the dynamic one is provided.

5.3 Dynamic Data

Node and edges data can take different values over time. Attributes must be declared as dynamic, allowing values to exist in during a time scope.

5.3.1 Declaring Dynamic Attributes

Listing 36: Indegree may change over time!

```
<gexf ...>
  ...
  <graph mode="dynamic" defaultedgetype="directed">
```

```
    <attributes class="node" mode="dynamic">
      <attribute id="2" title="indegree" type="float"/>
      ...
    </attributes>
    ...
  </graph>
</gexf>
```

5.3.2 Defining Dynamic Values

Attvalues have their scopes limited by the xml-attributes start and end.

Listing 37: Data value changing over time

```
<node id="3" label="BarabasiLab">
  <attvalues>
    <attvalue for="2" value="0" start="2009-01-01" end="2009-03-01"/>
    <attvalue for="2" value="1" start="2009-03-01" end="2009-03-10"/>
  </attvalues>
</node>
```

5.3.3 Dynamic Values and slices

If an attvalue is covering a period out of any slice, this period should be ignored by parsers. In the following example, the day 2009-01-03 is ignored:

Listing 38: Slices and attvalues

```
<gexf ...>
  ...
  <graph mode="dynamic">
    <node id="0" label="Hello">
      <attvalues>
        <attvalue for="0" value="1" start="2009-01-01" end="2009-01-05"/>
      </attvalues>
      <slices>
        <slice start="2009-01-01" end="2009-01-02" />
        <slice start="2009-01-04" end="2009-01-05" />
      </slices>
    </node>
    ...
  </graph>
</gexf>
```

If a value 'B' is declared after a value 'A' of the same attribute and overlaps it, then 'A' is forced to end at 'B' start. In the following example, 'A' will effectively end at 2009-03-03, and the value at 2009-03-03 is 'B':

Listing 39: Value overlapping

```
<node id="0" label="Hello">
  <attvalues>
    <attvalue for="0" value="A" start="2009-03-01" end="2009-03-05"/>
    <attvalue for="0" value="B" start="2009-03-03" end="2009-03-10"/>
  </attvalues>
</node>
```

One of the consequences of this rule is the right to end 'A' exactly with the 'B' start. Attribute '0' takes then the value 'B' at 2009-03-01.

Listing 40: Value transition

```
<node id="0" label="Hello">
  <attvalues>
    <attvalue for="0" value="A" end="2009-03-01"/>
    <attvalue for="0" value="B" start="2009-03-01"/>
  </attvalues>
</node>
```

While each data value is encoded in one piece of time, topology slices can cut them into more pieces. In the following example, a node exists from the beginning of the graph to 2009-03-01, and re-appears from 2009-03-05 to 2009-03-10. The data values will then fit inside these scopes even if they initially have a larger scope. Value 'A' of attribute '2' will be effective in 2009-03-01 (dates are inclusive), and from 2009-03-05 to 2009-03-10.

Listing 41: Slicing attvalues

```
<node id="2" label="RTGI">
  <attvalues>
    <attvalue for="0" value="http://rtgi.fr"/>
    <attvalue for="2" value="X" end="2009-02-28"/>
    <attvalue for="2" value="A" start="2009-03-01"/>
  </attvalues>
  <slices>
    <slice end="2009-03-01">
    <slice start="2009-03-05" end="2009-03-10">
  </slices>
</node>
```

6 Advanced Concepts IV: Extending GEXF

GEXF is designed to be easily extensible. Additional namespaces are defined by an XML Schema. The default namespace is always the gexf namespace. Gephi team actually provides a module for storing visualization data called *viz*.

6.1 VIZ module

Using the visualization module must be declared by adding the XML Attribute xmlns:viz="http://www.gexf.net/1.1draft/viz" to the document namespaces. The xsi:schemaLocation attribute includes the XML-Schema declaration of the VIZ module. The RelaxNG Compact specification is available in viz.rnc, and independent XSD declaration in viz.xsd.

Color, position and size of nodes and edges are stored as attributes.

6.1.1 Node Example

The following gexf contains a node having a color, a position, a shape and a specified size.

Listing 42: VIZ attributes

```
<gexf xmlns="http://www.gexf.net/1.1draft"
      xmlns:viz="http://www.gexf.net/1.1draft/viz">
...
  <node ... >
```

```
    <viz:color r="239" g="173" b="66"/>
    <viz:position x="15.783598" y="40.109245" z="0.0"/>
    <viz:size value="2.0375757"/>
    <viz:shape value="disc"/>
  </node>
...
</gexf>
```

6.1.2 Edge Example

The following gexf contains an edge having a color, a thickness and a shape.

Listing 43: VIZ attributes

```
<gexf xmlns="http://www.gexf.net/1.1draft"
      xmlns:viz="http://www.gexf.net/1.1draft/viz">
...
  <edge ... >
    <viz:color r="157" g="213" b="78"/>
    <viz:thickness value="5.124"/>
    <viz:shape value="solid"/>
  </edge>
...
</gexf>
```

6.1.3 Colors

Colors are defined by the RGB color model. Each XML-attribute value r, g or b is then an integer from 0 to 255.

Listing 44: VIZ color declaration

```
<viz:color r="239" g="173" b="66"/>
```

Listing 45: Color Specification

```
# extension point
node-content &=
    element color { color-content }?

# new point
color-content =
    attribute r { color-channel }
  & attribute g { color-channel }
  & attribute b { color-channel }

# new datatype
color-channel =
    xsd:nonNegativeInteger { maxInclusive = "255" }
```

6.1.4 Position

Space positions are set in three dimensions called x, y and z. Note that Gephi associates z as the height, and most of spatialization algorithms only use x and y. They are floats.

Listing 46: VIZ position declaration

```
<viz:position x="15.783598" y="40.109245" z="0.0"/>
```

Listing 47: Position Specification

```
# extension point
node-content &=
  element position { position-content }?

# new point
position-content =
    attribute x { space-point }
  & attribute y { space-point }
  & attribute z { space-point }

# new datatype
space-point =
    xsd:float
```

6.1.5 Size

Node size is a scale. It is set to *1.0* by default and is a non-negative float. Network viz softwares assume that an object representing a node of size *2.0* is twice bigger as one of *1.0*.

Listing 48: VIZ size declaration

```
<viz:size value="2.0375757"/>
```

Listing 49: Size Specification

```
# extension point
node-content &=
  element size { size-content }?

# new point
size-content =
    attribute value { size-type }

# new datatype
size-type = [ a:defaultValue = "1.0" ]
    xsd:float { minInclusive = "0.0"}
```

6.1.6 Thickness

Edge thickness is a scale. It is set to *1.0* by default and is a non-negative float. Network viz softwares assume that an object representing an edge of thickness *2.0* is twice bigger as one of *1.0*.

Listing 50: VIZ thickness declaration

```
<viz:size value="2.0375757"/>
```

Listing 51: Thickness Specification

```
# extension point
edge-content &=
```

```
  element thickness { thickness-content }?

# new point
thickness-content =
    attribute value { thickness-type }

# new datatype
thickness-type = [ a:defaultValue = "1.0" ]
    xsd:float { minInclusive = "0.0"}
```

6.1.7 Node Shape

Default node is shaped as a disc. Four shapes are proposed: *disc*, *square*, *triangle* and *diamond*. Images require an additional xml-attribute to set their location: uri.

Listing 52: Node Shape Specification

```
# extension point
node-content &=
  element shape { node-shape-content }?

# new point
node-shape-content =
    attribute value { node-shape-type }
   & attribute uri { xsd:anyURI }?

# new datatype
node-shape-type = [ a:defaultValue = "disc" ]
    string "disc"  |
    string "square"  |
    string "triangle"  |
    string "diamond"  |
    string "image"
```

Listing 53: Image declaration

```
<viz:shape value="image" uri="http://my.image.us/blah.jpg"/>
```

6.1.8 Edge Shape

Default edge is shaped as solid. Four shapes are proposed: *solid*, *dotted*, *dashed* and *double*.

Listing 54: Edge Shape Specification

```
# extension point
edge-content &=
  element shape { edge-shape-content }?

# new point
edge-shape-content =
    attribute value { edge-shape-type }

# new datatype
edge-shape-type = [ a:defaultValue = "solid" ]
    string "solid"  |
    string "dotted"  |
    string "dashed"  |
    string "double"
```

7 Advices: Parser optimization

This section provides some good tips to write parser-friendly files.

- Always place the edges after the nodes. Some parsers, depending on their implementation, may reject an edge if its linked nodes haven't been declared before, due to conceptual or data integrity reason.
- Use the *count* XML-attribute in *nodes* and *edges* declaration: the parser will know how much memory it has to allow, and will speed up the file reading. Note that count only refers to direct children, not the whole sub-graph!
- Prefer *liststring* to *string* attributes if you can. A smart parser will store the strings in one place, and just set pointers to them from the related nodes/edges.
- Identifiers may be interpreted as integers if you only use numbers. We encourage this practice, as an integer takes much less size in memory than an equivalent string. Tell the parser to optimize IDs storage by filling the optional graph XML-attribute called *idtype* with "string" or "integer".