

# GEXF 1.2draft Primer

GEXF Working Group

March 28, 2012

## 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

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Basic Concepts</b>	<b>2</b>
2.1	A Simple Graph . . . . .	3
2.2	Header . . . . .	3
2.3	Network Topology . . . . .	5
2.3.1	Declaring a Graph . . . . .	5
2.3.2	Declaring a Node . . . . .	6
2.3.3	Declaring an Edge . . . . .	7
2.4	Network Data . . . . .	8
2.4.1	Data types . . . . .	8
2.4.2	Attributes Example . . . . .	8
2.4.3	Declaring Attributes . . . . .	9
2.4.4	Defining Attribute Values . . . . .	10
<b>3</b>	<b>Advanced Concepts I: Hierarchy structure</b>	<b>12</b>
3.1	Introduction . . . . .	12
3.2	Sequential-safe Reading . . . . .	13
3.3	Random Writing . . . . .	14
<b>4</b>	<b>Advanced Concepts II: Phylogeny structure</b>	<b>15</b>
<b>5</b>	<b>Advanced Concepts III: Dynamics</b>	<b>15</b>
5.1	Example . . . . .	16
5.2	Dynamic Topology . . . . .	17
5.3	Dynamic Data . . . . .	18
5.3.1	Declaring Dynamic Attributes . . . . .	18
5.3.2	Defining Dynamic Values . . . . .	19
5.3.3	Dynamic Values and Spells . . . . .	19

<b>6</b>	<b>Advanced Concepts IV: Extending GEXF</b>	<b>21</b>
6.1	VIZ module . . . . .	21
6.1.1	Node Example . . . . .	21
6.1.2	Edge Example . . . . .	21
6.1.3	Colors . . . . .	22
6.1.4	Position . . . . .	22
6.1.5	Size . . . . .	23
6.1.6	Thickness . . . . .	23
6.1.7	Node Shape . . . . .	24
6.1.8	Edge Shape . . . . .	24
<b>7</b>	<b>Advices: Parser optimization</b>	<b>24</b>
<b>8</b>	<b>Web Services</b>	<b>25</b>
<b>9</b>	<b>Changelog</b>	<b>25</b>

## 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.

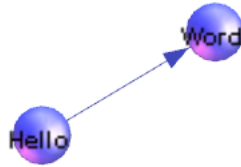


Figure 1: Hello-world graph

## 2.1 A Simple Graph

This is a dummy graph:

Listing 1: Hello world!

```

3  <?xml version="1.0" encoding="UTF-8"?>
4  <gexf xmlns="http://www.gexf.net/1.2draft"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://www.gexf.net/1.2draft
7         http://www.gexf.net/1.2draft/gexf.xsd"
8     version="1.2">
9  <meta lastmodifieddate="2009-03-20">
10 <creator>Gephi.org</creator>
11 <description>A hello world! file</description>
12 </meta>
13 <graph defaultedgetype="directed">
14 <nodes>
15 <node id="0" label="Hello"/>
16 <node id="1" label="Word"/>
17 </nodes>
18 <edges>
19 <edge id="0" source="0" target="1"/>
20 </edges>
21 </graph>
22 </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

```

3  <?xml version="1.0" encoding="UTF-8"?>
4  <gexf xmlns="http://www.gexf.net/1.2draft"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://www.gexf.net/1.2draft
7         http://www.gexf.net/1.2draft/gexf.xsd"
8     version="1.2">
9  <meta lastmodifieddate="2009-03-20">
10 <creator>Gephi.org</creator>

```

```

9   <description>A hello world! file</description>
    <keywords>basic, web</keywords>
    </meta>
12  ...
    </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/1.2draft`. For this reason we define this namespace as the default namespace in the document by adding the XML Attribute `xmlns="http://www.gexf.net/1.2draft"` 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/1.2draft http://www.gexf.net/1.2draft/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 additional information about the network. Element leafs are assumed to be text, and lastmodifieddate is an international standard date (yyyy-mm-dd). The graph element must be declared after the meta element.

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/1.2draft"
namespace rng = "http://relaxng.org/ns/structure/1.0"
3  datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"

# Grammar root
6  start = element gexf { gexf-content }

# Tree
9  gexf-content =
    attribute version { string "1.2" }
    & attribute variant { xsd:string }?
12  & (element meta { meta-content }?
    , element graph { graph-content })

15 # Attributes & Leafs
meta-content =
18  attribute lastmodifieddate { xsd:date }?
    & element creator { text }?
    & element keywords { text }?
    & element description { text }?

```

## 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 edge 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>
3   <node id="0" label="Hello" />
   <node id="1" label="Word" />
   ...
6  </nodes>
  <edges>
   <edge id="0" source="0" target="1" weight="3.167" />
9   ...
  </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.

The edges element must be declared after the `nodes` element.

Listing 5: An empty graph!

```
<graph>
  <nodes>
3  </nodes>
  <edges>
6  </edges>
</graph>
```

Listing 6: Topology Specification

```
graph-content =
  attribute defaultedgetype { defaultedgetype-type }?
3  & attribute idtype { idtype-type }?
  & attribute mode { mode-type }?
  & (element nodes { nodes-content }
6  , element edges { edges-content })
9  # Nodes
```

```

nodes-content =
12   attribute count { xsd:nonNegativeInteger }?
   & element node { node-content }*

15 node-content =
   attribute id { id-type }
   & attribute label { xsd:token }?

18

21 # Edges

edges-content =
24   attribute count { xsd:nonNegativeInteger }?
   & element edge { edge-content }*

edge-content =
27   attribute id { id-type }
   & attribute type { edgetype-type }?
   & attribute label { xsd:token }?
30   & attribute source { id-type }
   & attribute target { id-type }
   & attribute weight { weight-type }?

33

36 # Datatypes

defaultedgetype-type = [ a:defaultValue = "undirected" ]
39   string "directed" |
   string "undirected" |
   string "mutual"

42 edgetype-type = [ a:defaultValue = "undirected" ]
   string "directed" |
45   string "undirected" |
   string "mutual"

48 id-type =
   xsd:string | xsd:integer

51 idtype-type = [ a:defaultValue = "string" ]
   string "integer" |
   string "string"

54 mode-type = [ a:defaultValue = "static" ]
   string "static" |
   string "dynamic"

57 weight-type = [ a:defaultValue = "1.0" ]
   xsd:float

```

### 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-content =
   attribute id { id-type }
```

```

3   & attribute label { xsd:token }?
   id-type =
6   xsd:string | xsd:integer

```

### 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 explicitly 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-content =
  attribute id { id-type }
3  & attribute type { edgetype-type }?
  & attribute label { xsd:token }?
  & attribute source { id-type }
6  & attribute target { id-type }
  & attribute weight { weight-type }?

9  # Datatypes

id-type =
12  xsd:string | xsd:integer

edgetype-type = [ a:defaultValue = "undirected" ]
15  string "directed" |
  string "undirected" |
  string "mutual"

18  weight-type = [ a:defaultValue = "1.0" ]
  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/1.2draft"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.gexf.net/1.2draft
                           http://www.gexf.net/1.2draft/gexf.xsd"
6     version="1.2">
  <meta lastmodifieddate="2009-03-20">
    <creator>Gephi.org</creator>
9    <description>A Web network</description>
  </meta>
  <graph defaultedgetype="directed">
12    <attributes class="node">
      <attribute id="0" title="url" type="string"/>
      <attribute id="1" title="indegree" type="float"/>
15     <attribute id="2" title="frog" type="boolean">
        <default>true</default>
      </attribute>
18    </attributes>
    <nodes>
      <node id="0" label="Gephi">
21        <attvalues>
          <attvalue for="0" value="http://gephi.org"/>
          <attvalue for="1" value="1"/>
24        </attvalues>
      </node>
      <node id="1" label="Webatlas">
27        <attvalues>
          <attvalue for="0" value="http://webatlas.fr"/>
          <attvalue for="1" value="2"/>
30        </attvalues>
      </node>
      <node id="2" label="RTGI">
33        <attvalues>
          <attvalue for="0" value="http://rtgi.fr"/>
          <attvalue for="1" value="1"/>
36        </attvalues>
      </node>
      <node id="3" label="BarabasiLab">
39        <attvalues>
```



```

    <attvalue for="0" value="http://barabasilab.com"/>
    <attvalue for="1" value="1"/>
42  <attvalue for="2" value="false"/>
    </attvalues>
  </node>
45 </nodes>
  <edges>
    <edge id="0" source="0" target="1"/>
48  <edge id="1" source="0" target="2"/>
    <edge id="2" source="1" target="0"/>
    <edge id="3" source="2" target="1"/>
51  <edge id="4" source="0" target="3"/>
  </edges>
</graph>
54 </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">
3   <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"/>
6   </attributes>
  <attributes class="edge">
   <attribute id="0" title="my-float-attribute" type="float">
9     <default>2.0</default>
   </attribute>
  </attributes>
12  ...
</graph>

```

Listing 14: Attributes Specification

```

attributes-content =
  attribute class { class-type }
3  & attribute mode { mode-type }?
  & element attribute { attribute-content }*

6  attribute-content =
  attribute id { id-type }
  & attribute title { xsd:string }
9  & attribute type { attrtype-type }
  & element default { text }?
  & element options { text }?

12 # Datatypes

15 class-type =
  string "node" |
18  string "edge"

mode-type = [ a:defaultValue = "static" ]
21  string "static" |
  string "dynamic"

24 attrtype-type =
  string "integer" |

```

```

27   string "long" |
    string "double" |
    string "float" |
    string "boolean" |
30   string "liststring" |
    string "string" |
    string "anyURI"

```

#### 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>
3   <attvalue for="0" value="samplevalue"/>
   <attvalue for="1" value="1831"/>
6   <attvalue for="2" value="true"/>
  </attvalues>
</node>

```

Listing 16: Edge Attributes

```

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

```

Listing 17: Attribute Values Specification

```

attvalues-content =
  element attvalue { attvalue-content }*
3
attvalue-content =
  attribute for { id-type }
6  & attribute value { xsd:string }

```

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">
3   <attribute id="0" title="my-liststring-attribute" type="liststring">
     <default>foo|bar|foobar</default>

```

```

6   </attribute>
   </attributes>
   ...
</graph>

```

Listing 19: Liststring usage

```

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

```

A complete example:

Listing 20: Boolean version

```

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

```

Listing 21: Liststring version

```

3 <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>
6 <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>
12 </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

```

3 <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>

```

```

12  ...
    </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">
3    <options>foo1|foo2|foo3</options>
  </attribute>
</attributes>
6 <nodes>
  <node id="42" label="node A">
    <attvalues>
9      <attvalue for="0" value="foo3|foo2">
    </attvalues>
  </node>
12 <node id="43" label="node B">
    <attvalues>
15    <attvalue for="0" value="">
    </attvalues>
  </node>
18 <node id="44" label="node C">
    <attvalues>
21    <attvalue for="0" value="foo1|foo2|foo3">
    </attvalues>
  </node>
</nodes>

```

Listing 24: Invalid values

```

...
3 <node id="42" label="node A">
  <attvalues>
    <attvalue for="0" value="foo1|foo4">
  </attvalues>
6 </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 descendants, 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 refers to a parent node id with the XML-attribute pid.

Listing 25: Hierarchy Specification

```

# Extension Point
node-content &=
3   attribute pid { id-type }?

```

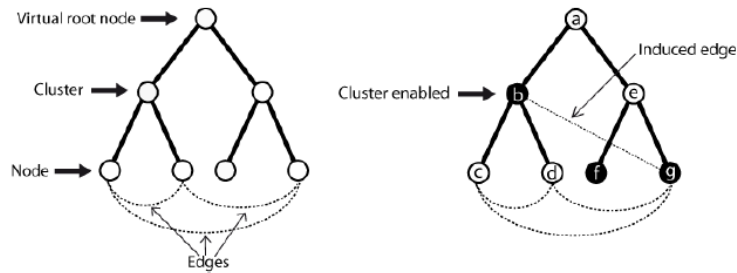


Figure 2: Graph tree with a virtual edge from a cluster to a leaf

```
& element nodes { nodes-content }?
& element edges { edges-content }?
```

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

### 3.2 Sequential-safe Reading

Listing 26: First way

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

Note that edges are not necessarily written at the end:

Listing 27: First way with edges inside clusters

```
<graph mode="static" defaultedgetype="directed">
  <nodes>
```

```

3   <node id="a" label="Kevin Bacon">
      <nodes>
6     <node id="b" label="God">
          <nodes>
            <node id="c" label="human1"/>
            <node id="d" label="human2"/>
          </nodes>
9     <edges>
        <edge id="0" source="c" target="d" />
      </edges>
    </node>
    <node id="e" label="Me">
15     <nodes>
        <node id="f" label="frog1"/>
        <node id="g" label="frog2"/>
      </nodes>
    </node>
  </nodes>
21  <edges>
    <edge id="1" source="b" target="e" />
    <edge id="2" source="f" target="a" />
24  <edge id="2" source="g" target="b" />
  </edges>
</node>
27 </nodes>
    <edges />
  </graph>

```

### 3.3 Random Writing

If you can't structure 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 28: Second way

```

<nodes>
3  <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" />
6  <node id="e" label="Me" pid="a" />
   <node id="f" label="frog1" pid="e" />
   <node id="g" label="frog2" pid="e" />
9  </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 29: Second way randomized

```

<nodes>
3  <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" />
6  <node id="e" label="Me" pid="a" />
   <node id="d" label="human2" pid="b" />
   <node id="f" label="frog1" pid="e" />
9  </nodes>

```

## 4 Advanced Concepts II: Phylogeny structure

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

Listing 30: Multiple parents

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

Listing 31: Phylogeny Specification

```
# Extension Point
node-content &=
3   element parents { parents-content }?

# New Point
6 parents-content =
   element parent { parent-content }*

9 # New Point
parent-content =
   attribute for { id-type }
```

## 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 32: Dynamic Enabled!

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

Time in GEXF is encoded in two ways, discrete or continuous.

Discrete, it is an *integer* or a *double*. Continuous, it is encoded as an international standard *date* (yyyy-mm-dd) or a *dateTime* defined by the corresponding XSD Datatype. If omitted, the default type is *double*. Use the the XML-attribute `timeformat` of the graph element to explicitly declare the type.

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.

Non-inclusive intervals are available as well: `startopen` and `endopen`. One can mix an inclusive and a non-inclusive boundary, but declaring both `start` and `startopen`, or both `end` and `endopen` for the same element is forbidden.

## 5.1 Example

Listing 33: A (small) Dynamic Web Graph with continuous time

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.gexf.net/1.2draft
                       http://www.gexf.net/1.2draft/gexf.xsd"
6   version="1.2">
  <meta lastmodifieddate="2009-03-20">
    <creator>Gephi.org</creator>
9   <description>A Web network changing over time</description>
  </meta>
  <graph mode="dynamic" defaultedgetype="directed" timeformat="date"
12   start="2009-01-01" end="2009-03-20">
    <attributes class="node" mode="static">
      <attribute id="0" title="url" type="string"/>
15     <attribute id="1" title="frog" type="boolean">
        <default>true</default>
      </attribute>
    </attributes>
18   <attributes class="node" mode="dynamic">
      <attribute id="2" title="indegree" type="float"/>
    </attributes>
21   <nodes>
     <node id="0" label="Gephi" start="2009-03-01">
24     <attvalues>
        <attvalue for="0" value="http://gephi.org"/>
        <attvalue for="2" value="1"/>
27     </attvalues>
     </node>
     <node id="1" label="Webatlas">
30     <attvalues>
        <attvalue for="0" value="http://webatlas.fr"/>
        <attvalue for="2" value="1" end="2009-03-01"/>
33     <attvalue for="2" value="2" start="2009-03-01" end="2009-03-10"/>
        <attvalue for="2" value="1" start="2009-03-11"/>
     </attvalues>
36     </node>
     <node id="2" label="RTGI" end="2009-03-10">
39     <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"/>
42     </attvalues>
     </node>
     <node id="3" label="BarabasiLab">
45     <attvalues>
        <attvalue for="0" value="http://barabasilab.com"/>
        <attvalue for="1" value="false"/>
48     <attvalue for="2" value="0" end="2009-03-01"/>
        <attvalue for="2" value="1" start="2009-03-01"/>
     </attvalues>
51     </node>
   </nodes>
   <edges>
54     <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"/>
57     <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"/>
60   </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 34: 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:

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

Listing 35: 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 36: 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. Using a non-inclusive endopen value implies to keep it strictly greater than the start value.

If a node or an edge exists only at some timeranges, we use the concept of spells. Spells are not provided for data values, which are only limited by one start and one end. Use the xml-element spells for topology like this:

Listing 37: Node with multiple spells

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

If the xml-attributes start and end are used in node like before, they should be ignored by parsers: if spells are provided, only their content are taken into

account. If no start is provided, the spell begins with the network. If no end is provided, the spell ends with the network. If two spells are covering a same period of time, parsers should consider them as a unique spell.

Listing 38: Dynamic Topology Specification

```

# Extension Point
graph-content &=
3   attribute timeformat { timeformat-type }?
   & (
6     ( attribute start { time-type }?
       | attribute startopen { time-type }?)
       &
9     ( attribute end { time-type }?
       & attribute endopen { time-type }?)
   )

12 # Extension Point
node-content &= (
15   ( attribute start { time-type }?
     | attribute startopen { time-type }?)
     &
18   ( attribute end { time-type }?
     & attribute endopen { time-type }?)
   )
   & element spells { spells-content }?

21 # Extension Point
edge-content &= (
24   ( attribute start { time-type }?
     | attribute startopen { time-type }?)
     &
27   ( attribute end { time-type }?
     & attribute endopen { time-type }?)
   )
30   & element spells { spells-content }?

```

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 39: Indegree may change over time!

```

<gexf ...>
...
3  <graph mode="dynamic" defaultedgetype="directed">
  <attributes class="node" mode="dynamic">
    <attribute id="2" title="indegree" type="float"/>
    ...
6  </attributes>
  ...
9  </graph>
</gexf>

```

Listing 40: Dynamic Attributes Specification

```

# Extension Point
attributes-content &= (
3   ( attribute start { time-type }?
    | attribute startopen { time-type }?)
    &
6   ( attribute end { time-type }?
    & attribute endopen { time-type }?)
    )

```

### 5.3.2 Defining Dynamic Values

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

Listing 41: Data value changing over time

```

<node id="3" label="BarabasiLab">
  <attvalues>
3   <attvalue for="2" value="0" start="2009-01-01" end="2009-03-01"/>
    <attvalue for="2" value="1" start="2009-03-02" end="2009-03-10"/>
  </attvalues>
6 </node>

```

Listing 42: Using open and closed intervals

```

<node id="3" label="BarabasiLab">
  <attvalues>
3   <attvalue for="2" value="15.0" start="2000" end="2005" />
    <attvalue for="2" value="20.0" startopen="2005" endopen="2010" />
    <attvalue for="2" value="25.0" start="2010" endopen="2015" />
6  </attvalues>
  </node>

```

Listing 43: Dynamic Values Specification

```

# Extension Point
attributes-content &= (
3   ( attribute start { time-type }?
    | attribute startopen { time-type }?)
    &
6   ( attribute end { time-type }?
    & attribute endopen { time-type }?)
    )

```

### 5.3.3 Dynamic Values and Spells

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

Listing 44: Spells and attvalues

```

<gexf ...>
  ...
3  <graph mode="dynamic">
    <node id="0" label="Hello">
      <attvalues>
6     <attvalue for="0" value="1" start="2009-01-01" end="2009-01-05"/>
      </attvalues>
      <spells>
9     <spell start="2009-01-01" end="2009-01-02" />

```

```

    <spell start="2009-01-04" end="2009-01-05" />
    </spells>
12 </node>
    ...
    </graph>
15 </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 45: Value overlapping

```

<node id="0" label="Hello">
  <attvalues>
3   <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>
6 </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 46: Value transition

```

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

```

While each data value is encoded in one piece of time, spells 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 47: Spells

```

<node id="2" label="RTGI">
  <attvalues>
3   <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"/>
6  </attvalues>
  <spells>
9   <spell end="2009-03-01">
   <spell start="2009-03-05" end="2009-03-10">
  </spells>
</node>

```

Listing 48: Spells Specification

```

# New Point
spells-content =
3   element spell { spell-content }+

```

```

6  # New Point
   spell-content = (
       ( attribute start { time-type }?
         | attribute startopen { time-type }?)
9     &
       ( attribute end { time-type }?
         & attribute endopen { time-type }?)
12  )

```

## 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/1.2draft/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, size and shape are stored as attributes.

These elements are compatible with spells and intervals defined in previous section.

#### 6.1.1 Node Example

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

Listing 49: VIZ Attributes

```

<gexf xmlns="http://www.gexf.net/1.2draft"
      xmlns:viz="http://www.gexf.net/1.2draft/viz">
3  ...
   <node ... >
       <viz:color r="239" g="173" b="66" a="0.5"/>
6       <viz:position x="15.783598" y="40.109245" z="0.0"/>
       <viz:size value="2.0375757"/>
       <viz:shape value="disc"/>
9   </node>
   ...
</gexf>

```

#### 6.1.2 Edge Example

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

Listing 50: VIZ Attributes

```

<gexf xmlns="http://www.gexf.net/1.2draft"
      xmlns:viz="http://www.gexf.net/1.2draft/viz">
3  ...

```

```

6 <edge ... >
  <viz:color r="157" g="213" b="78"/>
  <viz:thickness value="5.124"/>
  <viz:shape value="solid"/>
  </edge>
9 ...
</gexf>

```

### 6.1.3 Colors

Colors are defined by the RGBA color model. Each XML-attribute value *r*, *g* or *b* is hence an integer from 0 to 255, and the alpha value *a* is a float from 0.0 to 1.0.

Listing 51: VIZ Color Declaration

```
<viz:color r="239" g="173" b="66" a="0.5"/>
```

Listing 52: Color Specification

```

3 # Extension Point
  node-content &=
    element color { color-content }?
6 # New Point
  color-content =
    attribute r { color-channel }
    & attribute g { color-channel }
    & attribute b { color-channel }
    & attribute a { alpha-channel }?
12 # Datatypes
  color-channel =
15   xsd:nonNegativeInteger { maxInclusive = "255" }
  alpha-channel = [ a:defaultValue = "1.0" ]
18   xsd:float { minInclusive = "0.0" maxInclusive = "1.0" }

```

### 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 53: VIZ Position Declaration

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

Listing 54: Position Specification

```

3 # Extension Point
  node-content &=
    element position { position-content }?
6 # New Point
  position-content =
    attribute x { space-point }
    & attribute y { space-point }

```

```

9   & attribute z { space-point }
12  # 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 55: VIZ Size Declaration

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

Listing 56: Size Specification

```

3  # Extension Point
    node-content &=
      element size { size-content }?
6  # New Point
    size-content =
      attribute value { size-type }
9  # 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 57: VIZ Thickness Declaration

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

Listing 58: Thickness Specification

```

3  # Extension Point
    edge-content &=
      element thickness { thickness-content }?
6  # New Point
    thickness-content =
      attribute value { thickness-type }
9  # 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 59: Node Shape Specification

```
# Extension Point
node-content &=
3   element shape { node-shape-content }?

# New Point
6   node-shape-content =
    attribute value { node-shape-type }
    & attribute uri { xsd:anyURI }?

9

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

Listing 60: 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 61: Edge Shape Specification

```
# Extension Point
edge-content &=
3   element shape { edge-shape-content }?

# New Point
6   edge-shape-content =
    attribute value { edge-shape-type }

9

# Datatype
12  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, as it is mandatory since version 1.2. 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 should allocate, 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“.

## 8 Web Services

The content-type for serving GEXF files via HTTP/REST is *application/gexf+xml*.

## 9 Changelog

This document has been edited on:

- 28 March 2012 – Add open interval example in section Defining Dynamic Values
- 29 Nov 2010 – Initial release