Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Overview of Standard Graph File Formats

A. Messina

Rapporto Tecnico N.:
RT-ICAR-PA-2018-06

Dicembre 2018

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sede di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sede di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Overview of Standard Graph File Formats

A. Messina[1]

Rapporto Tecnico N.:
RT-ICAR-PA-2018-06 **Dicembre 2018**

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Palermo, Via Ugo La Malfa n. 153, 90146 Palermo.

Index

1 Introduction

There are many different file formats for graphs. The capabilities of these file formats range from simple adjacency lists or coordinates to complex formats that can store arbitrary data. This has led to a situation where we have a large number of different, mostly incompatible formats.

Exchanging graphs between different programs is painful, and sometimes impossible. The obvious answer to this problem is the introduction of a common file format.

One reason is that exchange formats often do not support all product and platform specific features. This is inevitable, but should not exclude the exchange of platform independent parts, probably with a less-efficient, portable replacement for product specific features.

Another concern is efficiency. One should not expect a universal format to be more efficient than one that is designed for a specific purpose, but there is no reason that a common file format should be so inefficient that it cannot be used.

In the case of graphs, many file formats for graphs are not designed for efficiency, but for ease of use, so the overhead should be small. Furthermore, there is no reason that prevents the use of both an optimized native format, and a second interchange format.

In general, a common graph file format should have the following features:

1. The format must be platform independent, and easy to implement.

2. It must have the capability to represent arbitrary data structures, since advanced programs have the need to attach their specific data to nodes and edges.

3. It should be flexible enough that a specific order of declarations is not needed, and that any non-essential data may be omitted.

2 CSV Format

2.1 Introduction

A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format [1].

The term "CSV" also denotes some closely related delimiter-separated formats that use different field delimiters, for example, semicolons. These include tab-separated values and space-separated values. A delimiter that is not present in the field data (such as tab) keeps the format parsing simple. These alternate delimiter-separated files are often even given a .csv extension despite the use of a non-comma field separator. This loose terminology can cause problems in data exchange. Many applications that accept CSV files have options to select the delimiter character and the quotation character.

2.2 CSV Graphs

CSV files can simply represent relationships and, usually, graphs expressed in CSV format are directed graphs.

2.2.1 Edge list

A graph with two edges “a”->”b” and “b”->”c” can be written as follow:

```
a;b
b;c
```

2.2.2 Adjacency list

All edges can be written as node pairs. It’s also possible to write all node’s connection on the same line. The example below represents a graph with 3 edges:

“a” -> “b”, “b” -> “c” and “b” -> “d”.

```
a;b
b;c;d
```

2.2.3 Mixed

The following example shows various cases that CSV supports as well. Self-loops and mutual edges are supported. It’s also possible to repeat an edge, “D” -> “E” is repeated twice in this example. As a consequence, the edge weight is incremented. “D” -> “E” has a edge weight at two, whereas default value is one.

```
A,B
B,A
C,C
D,E
A,D
D,B,E
F,G,A,B
```

2.2.4 Matrix

The sample below shows a graph with 5 nodes. An edge is created when the cell is ‘1′.

```
;A;B;C;D;E
A;0;1;0;1;0
B;1;0;0;0;0
C;0;0;1;0;0
D;0;1;0;1;0
E;0;0;0;0;0
```

Different edges weights can be simply expressed replacing ‘1’ by a value formatted as a double.

3 GraphViz DOT Format

3.1 Introduction

DOT is the text file format of the suite GraphViz [2], an open source graph visualization software. It has a human-readable syntax that describes network data, including subgraphs and elements appearances (i.e. color, width, label).

3.2 The DOT Language

The following is an abstract grammar defining the DOT language, where:

- Terminals are shown in bold font and nonterminals in italics;
- Literal characters are given in single quotes;
- Parentheses (and) indicate grouping when needed;
- Square brackets [and] enclose optional items;
- Vertical bars | separate alternatives.

```
Graph      : [ strict ] (graph | digraph) [ ID ] '{' stmt_list '}'
stmt_list  : [ stmt [ ';' ] stmt_list ]
stmt       : node_stmt
           | edge_stmt
           | attr_stmt
           | ID '=' ID
           | subgraph
attr_stmt  : (graph | node | edge) attr_list
attr_list  : '[' [ a_list ] ']' [ attr_list ]
a_list     : ID '=' ID [ (';' | ',') ] [ a_list ]
edge_stmt  : (node_id | subgraph) edgeRHS [ attr_list ]
edgeRHS    : edgeop (node_id | subgraph) [ edgeRHS ]
node_stmt  : node_id [ attr_list ]
node_id    : ID [ port ]
port       : ':' ID [ ':' compass_pt ]
           | ':' compass_pt
Subgraph   : [ subgraph [ ID ] ] '{' stmt_list '}'
compass_pt : (n | ne | e | se | s | sw | w | nw | c | _)
```

The keywords node, edge, graph, digraph, subgraph, and strict are case-independent. Note also that the allowed compass point values are not keywords, so these strings can be used elsewhere as ordinary identifiers and, conversely, the parser will actually accept any identifier.

An ID is one of the following:

- Any string of alphabetic ([a-zA-Z\200-\377]) characters, underscores ('_') or digits ([0-9]), not beginning with a digit;
- a numeral [-]?(.[0-9]+ | [0-9]+(.[0-9]*)?);
- any double-quoted string ("...") possibly containing escaped quotes (\");
- an HTML string (<...>).

An ID is just a string; the lack of quote characters in the first two forms is just for simplicity. There is no semantic difference between abc_2 and "abc_2", or between 2.34 and "2.34".

Obviously, to use a keyword as an ID, it must be quoted. Note that, in HTML strings, angle brackets must occur in matched pairs, and newlines and other formatting whitespace characters are allowed.

In addition, the content must be legal XML, so that the special XML escape sequences for ", &, <, and > may be necessary in order to embed these characters in attribute values or raw text. As an ID, an HTML string can be any legal XML string. However, if used as a label attribute, it is interpreted specially and must follow the syntax for HTML-like labels.

Both quoted strings and HTML strings are scanned as a unit, so any embedded comments will be treated as part of the strings.

An edgeop is -> in directed graphs and -- in undirected graphs.

The language supports C++-style comments: /* */ and //. In addition, a line beginning with a '#' character is considered a line output from a C preprocessor (e.g., # 34 to indicate line 34) and discarded.

Semicolons and commas aid readability but are not required. Also, any amount of whitespace may be inserted between terminals.

As another aid for readability, dot allows double-quoted strings to span multiple physical lines using the standard C convention of a backslash immediately preceding a newline character. In addition, double-quoted strings can be concatenated using a '+' operator. As HTML strings can contain newline characters, which are used solely for formatting, the language does not allow escaped newlines or concatenation operators to be used within them.

3.2.1 Subgraphs and Clusters

Subgraphs play three roles in Graphviz. First, a subgraph can be used to represent graph structure, indicating that certain nodes and edges should be grouped together. This is the usual role for subgraphs and typically specifies semantic information about the graph components. It can also provide a convenient shorthand for edges. An edge statement allows a subgraph on both the left and right sides of the edge operator. When this occurs, an edge is created from every node on the left to every node on the right. For example, the specification

```
A -> {B C}
```

is equivalent to

```
A -> B
A -> C
```

In the second role, a subgraph can provide a context for setting attributes. For example, a subgraph could specify that blue is the default color for all nodes defined in it. In the context of graph drawing, a more interesting example is:

is equivalent to

```
subgraph {
    rank = same; A; B; C;
}
```

This (anonymous) subgraph specifies that the nodes A, B and C should all be placed on the same rank if drawn using dot.

The third role for subgraphs directly involves how the graph will be laid out by certain layout engines. If the name of the subgraph begins with cluster, Graphviz notes the subgraph as a special cluster subgraph. If supported, the layout engine will do the layout so that the nodes belonging to the cluster are drawn together, with the entire drawing of the cluster contained within a bounding rectangle. Note that, for good and bad, cluster subgraphs are not part of the DOT language, but solely a syntactic convention adhered to by certain of the layout engines.

3.2.2 Lexical and Semantic Notes

A graph must be specified as either a *digraph* or a *graph*. Semantically, this indicates whether or not there is a natural direction from one of the edge's nodes to the other.

Lexically, a digraph must specify an edge using the edge operator -> while a undirected graph must use --. Operationally, the distinction is used to define different default rendering attributes. For example, edges in a digraph will be drawn, by default, with an arrowhead pointing to the head node. For ordinary graphs, edges are drawn without any arrowheads by default.

A graph may also be described as *strict*. This forbids the creation of multi-edges, i.e., there can be at most one edge with a given tail node and head node in the directed case. For undirected graphs, there can be at most one edge connected to the same two nodes. Subsequent edge statements using the same two nodes will identify the edge with the previously defined one and apply any attributes given in the edge statement. For example, the graph

```
strict graph {
  a -- b
  a -- b
  b -- a [color=blue]
}
```

will have a single edge connecting nodes a and b, whose color is blue.

If a default attribute is defined using a *node*, *edge*, or *graph* statement, or by an attribute assignment not attached to a node or edge, any object of the appropriate type defined afterwards will inherit this attribute value. This holds until the default attribute is set to a new value, from which point the new value is used. Objects defined before a default attribute is set will have an empty string value attached to the attribute once the default attribute definition is made.

A subgraph receives the attribute settings of its parent graph at the time of its definition. This can be useful; for example, one can assign a font to the root graph and all subgraphs will also use the font. For some attributes, however, this property is undesirable. If one attaches a label to the root graph, it is probably not the desired effect to have the label used by all subgraphs. Rather than listing the graph attribute at the top of the graph, and the resetting the attribute as needed in the subgraphs, one can simply defer the attribute definition in the graph until the appropriate subgraphs have been defined.

If an edge belongs to a cluster, its endpoints belong to that cluster. Thus, where you put an edge can effect a layout, as clusters are sometimes laid out recursively.

There are certain restrictions on subgraphs and clusters. First, at present, the names of a graph and its subgraphs share the same namespace. Thus, each subgraph must have a unique name. Second, although nodes can belong to any number of subgraphs, it is assumed clusters form a strict hierarchy when viewed as subsets of nodes and edges.

3.3 Examples

3.3.1 Basic example

The sample below shows a directed graph with two edges.

```
digraph sample {
    A -> B;
    B -> C;
}
```

3.3.2 Labels

The sample below shows the same example but with both node and edge labels.

```
digraph sample2 {
    A -> B [ label = "Edge A to B" ];
    B -> C [ label = "Edge B to C" ];
    A [label="Node A"];
}
```

3.3.3 Adjacency lists

The sample below shows edges can be put as adjacency lists.

```
digraph sample3 {
    A -> {B ; C ; D}
    C -> {B ; A}
}
```

4 GDF Format

4.1 Introduction

GDF is the file format used by GUESS [3]. It is built like a database table or a coma separated file (CSV). It supports attributes to both nodes and edges. A standard file is divided in two sections, one for nodes and one for edges. Each section has a header line, which basically is the column title. Each element (i.e. node or edge) is on a line and values are separated by coma. The GDF format is therefore very easy to read and can be easily converted from CSV.

4.2 The GUESS .gdf format

The file structure for the *.gdf* files is very simple. We will basically define the nodes with their properties followed by the edges with theirs.

The node definition section starts with the line: “nodedef> name”

The *nodedef* line will tell what the format is of the following lines that actually describe nodes. In the simple case we are just going to have one column on each line, the node name. Nodes are required to have unique names (identifiers).

The simplest file looks something like this:

```
nodedef> name
foobar
```

which tells that we want a node called *foobar*.

All other aspects of the node (color, visibility, style) will be extracted from defaults. After name (the only required column), you may use pre-defined columns and new columns to set and control extra node properties.

Pre-defined columns are:

- **x** – a double representing the node’s x location (default: random)
- **y** – a double representing the node’s y location (default: random)
- **visible** – a boolean indicating if the node should be displayed (default: true)

- **color** – a string, the default color of the node (default: “blue”). We have a long list of color names that we know about, but if you didn’t want to use one of those you could quote an rgb triplet (e.g. “124,234,222”)
- **fixed** – boolean, can the node be moved? (default: false)
- **style** – an int indicating which style of node to use (default: 1). Currently GUESS maps: rectangle = 1, ellipse = 2, rounded rectangle = 3, text inside a rectangle = 4, text inside an ellipse = 5, text inside a rounded rectangle = 6, and an image = 7
- **width** – double, node width (default: 4)
- **height** – double, node height (default: 4)
- **label** – string, a label for the node in the visualization (default is the name)
- **labelvisible** – boolean, should we show the label? (default: false)
- **image** – string, a filename of the image to use if the node style = 7

These pre-defined attributes can be overridden by simply adding them to the list in the nodedef line. For example:

```
nodedef> name,x,y,color
foo,0,0,blue
bar,100,100,red
```

This means that you want two nodes: a blue one called foo at (0,0) and a red one called bar at (100,100). Notice that you don’t have to quote things explicitly (strings versus numbers). The system should figure that out for you (unless your string has a comma in which case you’ll want to put it in quotes).

Edges are defined in a very similar way, the only required columns for edges are “node1” and “node2” which are the names of the two nodes you are connecting.

A simple example is something like:

```
nodedef> name
a
b
c
d
edgedef> node1,node2
a,b
a,c
a,d
```

which defines a star network centered on node a.

Edges, like nodes, can contain pre-defined and user-defined attributes in the definition lines. Valid pre-defined edge properties are:

- **visible** – a boolean indicating if the edge should be displayed (default: true)
- **color** – a string, the default color of the node (default: “green”).
- **weight** – a double indicating the edge weight (default: 1, but not currently used for calculations)
- **width** – double, node width (default: .3)
- **directed** - boolean, indicating edge directionality (default: false, undirected/bidirected). If true, this will assume node1 is the source and node2 is the destination.
- **label** - string, a label for the node in the visualization (default is the edge weight)
- **labelvisible** - boolean, should we show the label? (default: false)

One critical thing to note is that duplicated edges are not supported. That is, you cannot create more than one edge of the same direction between two nodes. At most you can have 3 edges between two nodes (a->b, b->a, and a-b). Recall that a-b and a<->b are considered to be the same thing.

4.3 Examples

4.3.1 Basic example

The GDF below is a minimal example, where the label column is optional.

```
nodedef>name VARCHAR,label VARCHAR
s1,Site number 1
s2,Site number 2
s3,Site number 3
edgedef>node1 VARCHAR,node2 VARCHAR
s1,s2
s2,s3
s3,s2
s3,s1
```

4.3.2 Example with edge weight

Edge weight is basically edge thickness and is defined as follow.

```
nodedef>name VARCHAR,label VARCHAR
s1,Site number 1
s2,Site number 2
s3,Site number 3
edgedef>node1 VARCHAR,node2 VARCHAR, weight DOUBLE
```

```
s1,s2,1.2341
s2,s3,0.453
s3,s2, 2.34
s3,s1, 0.871
```

4.3.3 Various attributes

We can add as many attributes as we need. Add attributes title in the header line and respect order, as you would do for CSV. On the below example, all attributes are design attributes except "class" that I added.

```
nodedef>name     VARCHAR,label     VARCHAR,class     VARCHAR,     visible
BOOLEAN,labelvisible   BOOLEAN,width   DOUBLE,height   DOUBLE,x   DOUBLE,y
DOUBLE,color VARCHAR
s1,SiteA,blog,true,true,10.0,10.0,-52.11296,-25.921143,'114,116,177'
s2,SiteB,forum,true,true,10.986123,10.986123,-
20.114172,25.740356,'219,116,251'
s3,SiteC,webpage,true,true,10.986123,10.986123,8.598924,-
26.867584,'192,208,223'
edgedef>node1 VARCHAR,node2 VARCHAR,directed BOOLEAN,color VARCHAR
s1,s2,true,'114,116,177'
s2,s3,true,'219,116,251'
s3,s2,true,'192,208,223'
s3,s1,true,'192,208,223'
```

4.3.4 Working with texts

Problems often comes when coma, apostrophe (i.e. single-quote) or double-quote are used in texts. The example below shows how to manage these strings, wrap single-quotes around it.

```
nodedef>name VARCHAR,label VARCHAR,class VARCHAR, visible
BOOLEAN,labelvisible BOOLEAN,width DOUBLE,height DOUBLE,x DOUBLE,y
DOUBLE,color VARCHAR
s1,'Hello "world" !',type1,true,true,10.0,10.0,-52.11296,-
25.921143,'114,116,177'
s2,'Well, this is',type1,true,true,10.986123,10.986123,-
20.114172,25.740356,'219,116,251'
s3,'A correct 'GDF' file',type1,true,true,10.986123,10.986123,8.598924,-
26.867584,'192,208,223'
edgedef>node1 VARCHAR,node2 VARCHAR,directed BOOLEAN,color VARCHAR
s1,s2,true,'114,116,177'
s2,s3,true,'219,116,251'
s3,s2,true,'192,208,223'
s3,s1,true,'192,208,223'
```

5 GML Format

5.1 Introduction

The Graph Modeling Language (GML) [4] is a file format for graphs designed to represent arbitrary data structures and characterized by portability, simple syntax, extensibility and flexibility. A GML file consists of hierarchical key-value lists. Even if GML was bound to a specific system, namely Graphlet., then it has been overtaken and adopted by several other drawing graphs systems, such as Pajek [5], yEd [6], LEDA [7] and NetworkX [8].

5.2 Key issues of GML

A common file format must be platform independent and easy to implement. Furthermore, it must have the capability to represent arbitrary data structures, since advanced programs have the need to attach their specific data to nodes and edges. It should be flexible enough that a specific order of declarations is not needed, and that any non-essential data may be omitted. GML attempts to satisfy all these requirements.

```
graph [
  comment "This is a sample graph"
  directed 1
  IsPlanar 1
  node [
    id 1
    label "Node 1"
  ]
  node [
    id 2
    label "Node 2"
  ]
  node [
    id 3
    label "Node 3"
  ]
  edge [
    source 1
    target 2
    label "Edge from node 1 to node 2"
  ]
  edge [
    source 2
    target 3
    label "Edge from node 2 to node 3"
  ]
  edge [
    source 3
    target 1
    label "Edge from node 3 to node 1"
  ]
]
```

The above example describes a circle of three nodes. This example shows several key issues of GML:

- **ASCII Representation for Simplicity and Portability**. A GML file is a 7-bit ASCII file. This makes it simple to write files through standard routines. Parsers are easy to implement, either by hand or with standard tools like *lex* and *yacc*. Also, files are text files, they can be exchanged between platforms without special converters.
- **Simple Structure**. A GML file consists of hierarchically organized key-value pairs. A key is a sequence of alphanumeric characters, such as graph or id. A value is either an integer, a floating-point number, a string or a list of key-value pairs enclosed in square brackets.
- **Extensibility and Flexibility**. GML can represent arbitrary data, and it is possible to attach additional information to every object. For example, the graph in Figure 1 adds an *IsPlanar* attribute to the graph. This may lead to a situation in where an application adds data which cannot be understood by another application. Therefore, applications are free to ignore any data which they do not understand. They should, however, save these data and re-write them.
- **Representation of Graphs**. Graphs are represented by the keys graph, node and edge. The topological structure is modeled with the node's id and the edge's source and target attributes: the id attributes assign numbers to nodes, which are referenced by source and target.

5.3 GML Syntax

Table 1 - The GML Grammar in BNF Format

GML	::=	List
List	::=	*+* (whitespace Key whitespace Value)
Value	::=	Integer | Real | String | [List]
Key	::=	[**a-z A-Z**] [**a-z A-Z 0-9**]*
Integer	::=	sign digit$^+$
Real	::=	sign digit* **.** digit* mantissa
String	::=	" instring "
sign	::=	empty | **+** | **-**
digit	::=	[**0-9**]
mantissa	::=	empty | **E** sign digit
instring	::=	ASCII - {&,"} | & character$^+$;
whitespace	::=	space | tabulator | newline

In addition to the above grammar, all lines starting with a "#" character are ignored by the parser. This is a standard behavior for most UNIX software and allows the embedding of foreign data in a file. Of course, this information can also be added within the GML structure. However, it is convenient to add large external data through this mechanism, as any lines starting with # will not be read by another application.

Further reglementations are a maximum line length and a maximum key size of 254 characters (this is necessary since some operating systems and editors do not handle longer lines), and the use of 7-bit ASCII characters only. Any other characters are coded in the ISO 8859-1 character set, and have the form &name;. Especially, the characters " and & within strings must be coded this way to avoid ambiguity. The ISO 8859-1 is also used by HTML, which is the most common format for distributing data on the world wide web.

The above grammar is kept as simple as possible, and avoids unnecessary items like an "=" to stress assignments or specific data types for boolean or enumeration values. Keys and values are separated by white space. With that, it is straightforward to generate a GML file from a given structure, and a parser can easily be implemented on various platforms.

5.3.1 How Graphs and Other Data Structures are Represented

A GML file defines a tree. Each node in the tree is labelled by a key. Leaves have integer, floating point or string values. The notion

k1.k2. ... .kn

is used to specify a path in the tree where the nodes are labelled by keys k1, k2, ... kn.

x.k1.k2. ... .kn

is used to specify a path which starts at a specific node x in the tree.

In the above grammar, all lines starting with a "#" character are ignored by the parser. This is a standard behavior for most UNIX software and allows the embedding of foreign data in a file as well as within the GML structure. However, it is convenient to add large external data through this mechanism, as any lines starting with # will not be read by another application. The above grammar is kept as simple as possible. Keys and values are separated by white space. With that, it is straightforward to generate a GML file from a given structure, and a parser can easily be implemented on various platforms.

With GML, a graph is defined by the keys *graph*, *node* and *edge*, where *node* and *edge* are sons of *graph* in no particular order. Each non-isolated node must have a unique *.graph.node.id* attribute. Furthermore, the end nodes of the edges are given by the *.graph.edge.source* and *.graph.edge.target* attributes. Their values are the *graph.node.id* values of end nodes.

Directed and undirected graphs are stored in the same format. The distinction is done with the *.graph.directed* attribute of a graph. If the graph is undirected that attribute is omitted. In an undirected graph, *.graph.edge.source* and *.graph.edge.target* may be assigned arbitrarily. GML does not define separate representations for directed and undirected graphs since it would have made the parser more complex, especially in applications that read both directed and undirected graphs and additionally if graphics are being used source and target have a meaning even for undirected graphs for example, if an edge is represented by a polyline, then the sequence of points implies a direction on the edge.

5.3.2 Restrictions

- The values of the *.graph.node.id* elements must be unique within the graph.
- Each edge must have *.graph.edge.source* and *.graph.edge.target* attributes.
- Not all nodes have a *.id* field since this field is considered not necessary for isolated
- nodes. Referencing the node can be problematic.
- With these conventions, a simple parser for a Graph in GML works in four steps:
  1. Read the file and build the tree.
  2. Scan the tree for a node *g* labeled graph.
  3. Find and create all nodes in *g.node*. Remember their *g.node.id* values.
  4. Find all edges in *g.edge*, and their *g.edge.source* and *g.edge.target* attributes. Find the end nodes and insert the edges.

  Step 1 should be integrated into the other steps to gain efficiency. It requires all attributes to be saved leading to overhead. However, extraction of data attached to nodes, edges and graphs, becomes easier more so to preserve unknown data.
- Validation of the file is not possible using tools.

GML is a capable description language for graph drawing purposes and while it includes provision for extensions; the mechanisms for associating external data with a graph element is provision for extensions; the mechanisms for associating external data with a graph element is not well defined.

5.3.3 How to Represent Common Data Structures

Integers. GML uses signed 32-bit integers, which are commonly available on all architectures and languages. Larger numbers should be represented as strings. Especially, bitsets with more than 31 entries should be represented as strings.

- **Floating point**. Floating point values should stay inside the range of double precision floating point values.

- **Boolean**. Boolean values are represented by 0 (false) and 1 (true).

- **Pointers**. Pointers are modeled by id attributes. id values are not necessarily unique through the file; details are specified by the application. Alternatively, one could use a name attribute which assigns a string name to an object.

- **Record**. A record data structure can easily be translated into a GML subtree, like in the following example:

  ```
  name: record
        a: typea;
        b: typeb;
        c: typec;
  end;
  ```

  translates into

  ```
  name: [
        a Insert the value(s) of a here
        b Insert the value(s) of b here
        c Insert the value(s) of c here
  ]
  ```

- **List, Set, Array**. These data types are represented in the same ways as records are. e.g.

  ```
  name: List of x;
  ```

  translates into

  ```
  name: [
        x Insert the value(s) of the first element here
        x Insert the value(s) of the second element here
        x Insert the value(s) of the third element here
  ]
  ```

  Note that the key x occurs more than once within name. integrated intoParsers must preserve the order of objects to guarantee that the list is read correctly (see also the next section). integrated intoArrays should make x a list and specify the index in an .x.id field if necessary.

5.3.4 Order of Attributes

GML does usually not require that attributes appear in a specific order in the file. The order of objects is not considered significant as long as their keys are different. That is, if there are several attributes with the same key (id, label) in a list, then the parser integrated into must preserve their order.

5.3.5 Unknown Attributes

GML is designed so that any application can add its own features to graphs, nodes and edges. However, not all applications understand all attributes. GML deals with foreign data in two ways:

1. Simply ignore it. However, this means the data gets lost when the file is written, for example, a program that does graph transformations would throw away any graphics data.
2. An even greater complication is to save everything to a generic structure and write it back when a new file is written. This may guarantee no data is lost but can result in inconsistencies if the application alters the graph since both changes in the structure and in the values of attributes can make other attributes invalid.

5.3.6 Consistency

Consider the following situation: a file includes information of some graph theoretical property, say the existence of a Hamiltonian circle. It is easy to see that this information may become invalid if an edge is removed, but not if an edge is added. However, a program that does not know about Hamilton cycles will not be able to check and guarantee this property.

Another example is if a node is moved, then the coordinates of its adjacent edges must be updated. However, some programs always treat edges as straight lines from center to center and do not take care about this. Other programs might draw the edges in a more complex way, for example adjust the arrows at the end of the edge to the node's shape. Even more, an attribute *IsDrawnPlanar* might become invalid when node or edge coordinates have changed.

As these examples show, both changes in the structure and in the values of attributes can make other attributes invalid. We therefore need a way to specify which attributes are safe with changes and which not. This is done by the following rule:

Any keyword which starts with a capital letter should be considered invalid as soon as any changes have occurred. We call such a key unsafe.

This means that it is still possible to add the above information with keys like "*HasHamiltonianCircle*" or "*IsDrawnPlanar*", but in practice, this information will not be written to a file unless the application knows how to deal with that particular attribute.

5.4 Examples

5.4.1 Basic example

The sample below shows a graph of three nodes and two edges.

```
graph
[
  node
  [
   id A
  ]
  node
  [
   id B
  ]
  node
  [
   id C
  ]
   edge
  [
   source B
   target A
  ]
  edge
  [
   source C
   target A
  ]
]
```

5.4.2 Labels

The sample below shows the same example but with both node and edge labels.

```
graph
[
  node
  [
   id A
   label "Node A"
  ]
```

```
   node
   [
    id B
    label "Node B"
   ]
   node
   [
    id C
    label "Node C"
   ]
    edge
   [
    source B
    target A
    label "Edge B to A"
   ]
   edge
   [
    source C
    target A
    label "Edge C to A"
   ]
 ]
```

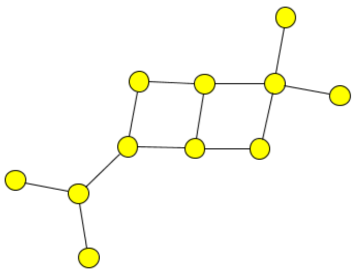
6 GraphML Format

6.1 Introduction

The GraphML file format [9] [10] uses *.graphml* extension and is XML structured. It supports attributes for nodes and edges, hierarchical graphs and benefits from a flexible architecture. Gephi supports a limited set of this format (no sub-graphs and hyperedges). This format is supported by NodeXL [9], Sonivis [10], GUESS [3] and NetworkX [8].

6.2 Basic Concepts

The purpose of a GraphML document is to define a graph. Let us start by considering the graph shown in the figure below. It contains 11 nodes and 12 edges.



That graph is defined by the following *.graphml* document:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <node id="n2"/>
    <node id="n3"/>
    <node id="n4"/>
    <node id="n5"/>
    <node id="n6"/>
    <node id="n7"/>
    <node id="n8"/>
    <node id="n9"/>
    <node id="n10"/>
    <edge source="n0" target="n2"/>
    <edge source="n1" target="n2"/>
    <edge source="n2" target="n3"/>
    <edge source="n3" target="n5"/>
    <edge source="n3" target="n4"/>
    <edge source="n4" target="n6"/>
    <edge source="n6" target="n5"/>
    <edge source="n5" target="n7"/>
    <edge source="n6" target="n8"/>
    <edge source="n8" target="n7"/>
    <edge source="n8" target="n9"/>
    <edge source="n8" target="n10"/>
  </graph>
</graphml>
```

The GraphML document consists of a `graphml` 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.

6.2.1 The Header

In this section we discuss the parts of the document which are common to all GraphML documents, basically the `graphml` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">

  ...

</graphml>
```

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 GraphML documents.

The second line contains the *root-element* element of a GraphML document: the `graphml` element. The `graphml` element, like all other GraphML elements, belongs to the namespace `http://graphml.graphdrawing.org/xmlns`. For this reason, we define this namespace as the default namespace in the document by adding the XML Attribute `xmlns="http://graphml.graphdrawing.org/xmlns"` 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 GraphML documents located on the `graphdrawing.org` 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://graphml.graphdrawing.org/xmlns http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd"` , defines the XML Schema location for all elements in the GraphML namespace.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">

  ...

</graphml>
```

6.2.2 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 a `node` element, and an egde with an `edge` element.

```
<graph id="G" edgedefault="directed">
    <node id="n0"/>
    <node id="n1"/>
    ...
    <node id="n10"/>
    <edge source="n0" target="n2"/>
    <edge source="n1" target="n2"/>
    ...
    <edge source="n8" target="n10"/>
</graph>
```

In GraphML there is no order defined for the appearance of node and edge elements. Therefore, the following example is a perfectly valid GraphML fragment:

```
<graph id="G" edgedefault="directed">
    <node id="n0"/>
    <edge source="n0" target="n2"/>
    <node id="n1"/>
    <node id="n2"/>
    ...
</graph>
```

6.2.3 Declaring a Graph

Graphs in GraphML 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* is applied to the edge. The default direction is declared as the XML Attribute `edgedefault` of the `graph` element. The two possible values for this XML Attribute are `directed` and `undirected`. Note that the default direction must be specified.

Optionally an identifier for the graph can be specified with the XML Attribute `id`. The identifier is used, when it is necessary to reference the graph.

6.2.4 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`.

6.2.5 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.

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

The optional XML-Attribute `directed` declares if the edge is directed or undirected. The value `true` declares a directed edge, the value `false` an undirected edge. If the direction is not explicitly defined, the default direction is applied to this edge as defined in the enclosing graph.

Optionally an identifier for the edge can be specified with the XML Attribute `id`. When it is necessary to reference the edge, the `id` XML-Attribute is used.

```
...
<edge id="e1" directed="true" source="n0" target="n2"/>
...
```

6.3 GraphML-Attributes

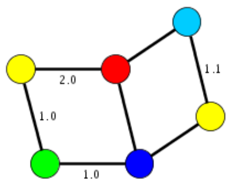
In the previous section we discussed how to describe the topology of a graph in GraphML. While pure topological information may be sufficient for some appications of GraphML, for the most time additional information is needed. With the help of the extension *GraphML-Attributes* one can specify additional information of simple type for the elements of the graph. Simple type means that the information is restricted to scalar values, e.g. numerical values and strings.

If you want to add structured content to graph elements you should use the key/data extension mechanism of GraphML.

GraphML-Attributes must not be confounded with XML-Attributes which are a different concept.

6.3.1 GraphML-Attributes Example

In this section a graph with colored nodes and edge weights will be our running example.



We will use GraphML-Attributes to store the extra data on the nodes and edges. The related .graphml file will be as follow:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
        http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="d0" for="node" attr.name="color" attr.type="string">
    <default>yellow</default>
  </key>
  <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
  <graph id="G" edgedefault="undirected">
    <node id="n0">
      <data key="d0">green</data>
    </node>
    <node id="n1"/>
    <node id="n2">
      <data key="d0">blue</data>
    </node>
    <node id="n3">
      <data key="d0">red</data>
    </node>
    <node id="n4"/>
    <node id="n5">
      <data key="d0">turquoise</data>
    </node>
    <edge id="e0" source="n0" target="n2">
      <data key="d1">1.0</data>
    </edge>
    <edge id="e1" source="n0" target="n1">
      <data key="d1">1.0</data>
    </edge>
    <edge id="e2" source="n1" target="n3">
      <data key="d1">2.0</data>
    </edge>
    <edge id="e3" source="n3" target="n2"/>
    <edge id="e4" source="n2" target="n4"/>
    <edge id="e5" source="n3" target="n5"/>
    <edge id="e6" source="n5" target="n4">
      <data key="d1">1.1</data>
    </edge>
  </graph>
</graphml>
```

6.3.2 Declaring GraphML-Attributes

A GraphML-Attribute is defined by a `key` element which specifies the *identifier*, *name*, *type* and *domain* of the attribute.

The identifier is specified by the XML-Attribute `id` and is used to refer to the GraphML-Attribute inside the document.

The name of the GraphML-Attribute is defined by the XML-Attribute `attr.name` and must be unique among all GraphML-Attributes declared in the document. The purpose of the name is that applications can identify the meaning of the attribute. Note that the name of the GraphML-Attribute is not used inside the document, the identifier is used for this purpose.

The type of the GraphML-Attribute can be either `boolean`, `int`, `long`, `float`, `double`, or `string`. These types are defined like the corresponding types in the Java™ programming language.

The domain of the GraphML-Attribute specifies for which graph elements the GraphML-Attribute is declared. Possible values include `graph`, `node`, `edge`, and `all`.

```
    ...
    <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
    ...
```

It is possible to define a default value for a GraphML-Attribute. The text content of the `default` element defines this default value.

```
    ...
  <key id="d0" for="node" attr.name="color" attr.type="string">
    <default>yellow</default>
  </key>
    ...
```

6.3.3 Defining GraphML-Attribute Values

The value of a GraphML-Attribute for a graph element is defined by a `data` element nested inside the element for the graph element. The `data` element has an XML-Attribute `key`, which refers to the identifier of the GraphML-Attribute. The value of the GraphML-Attribute is the text content of the `data` element. This value *must* be of the type declared in the corresponding `key` definition.

```
...
<key id="d0" for="node" attr.name="color" attr.type="string">
  <default>yellow</default>
</key>
<key id="d1" for="edge" attr.name="weight" attr.type="double"/>
<graph id="G" edgedefault="undirected">
  <node id="n0">
    <data key="d0">green</data>
  </node>
  <node id="n1"/>
  ...
  <edge id="e0" source="n0" target="n2">
    <data key="d1">1.0</data>
  </edge>
  <edge id="e1" source="n0" target="n1">
    <data key="d1">1.0</data>
  </edge>
  <edge id="e2" source="n1" target="n3">
    <data key="d1">2.0</data>
  </edge>
  <edge id="e3" source="n3" target="n2"/>
  ...
</graph>
...
```

There can be graph elements for which a GraphML-Attribute is defined but no value is declared by a corresponding `data` element. If a default value is defined for this GraphML-Attribute, then this default value is applied to the graph element. In the above example no value is defined for the node with identifier `n1` and the GraphML-Attribute with name `color`. Therefore, this GraphML-Attribute has the default value, `yellow` for this node. If no default value is specified, as for the GraphML-Attribute weight in the above example, the value of the GraphML-Attribute is undefined for the graph element. In the above example the value is undefined of the GraphML-Attribute weight for the edge with identifier `e3`.

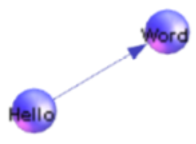
7 GEXF

7.1 Introduction

The Graph Exchange XML Format (GEXF) [13] is a language for describing complex networks structures, their associated data and dynamics. Started in 2007 at Gephi [14] project by different actors, deeply involved in graph exchange issues, the gexf specifications are mature enough to claim being both extensible and open, and suitable for real specific applications.

7.2 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.



A 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft"
      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"
    version="1.2">
  <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>
```

7.2.1 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft"
      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"
      version="1.2">
  <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/1.2draft`. 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.

7.2.2 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 orders doesn't matter.

```
<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>
```

7.2.3 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. Static mode is assumed by default.

The edges element must be declared after the nodes element.

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

7.2.4 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.

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

7.2.5 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 an 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:

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

7.3 Network Data

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.

7.3.1 Data types

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

7.3.2 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft"
      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"
      version="1.2">
  <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>
```

7.3.3 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.

```
<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>
```

7.3.4 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.

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

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

The *liststring* type allows to replace 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.

The attribute *options* 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.

```
<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:

```
<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>
```

8 References

[1] Wikipedia, "Comma-separated values," [Online]. Available: https://en.wikipedia.org/wiki/Comma-separated_values. [Accessed 26 11 2018].

[2] "Graphviz - Graph Visualization Software," [Online]. Available: http://www.graphviz.org. [Accessed 30 11 2018].

[3] E. Adar, "GUESS: a language and interface for graph exploration," in *CHI '06 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Montréal, Québec, Canada, 2006.

[4] M. Himsolt, "GML: A portable Graph File Format," 30 11 2010. [Online]. Available: https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf. [Accessed 2 12 2018].

[5] W. De Nooy, A. Mrvar and V. Batageli, Exploratory Social Network Analysis with Pajek, Cambridge University Press, 2018.

[6] yWorks GmbH, "yEd Graph Editor," [Online]. Available: https://www.yworks.com/products/yed. [Accessed 2 12 2018].

[7] K. Mehlhorn and S. Näher, The LEDA Platform of Combinatorial and Geometric Computing, Cambridge University Press, 1999.

[8] A. Hagberg, D. Schult and P. Swart, "NetworkX," NetworkX developers, 2018. [Online]. Available: http://networkx.github.io. [Accessed 2 12 2018].

[9] GraphML Project Group, "The GraphML File Format," [Online]. Available: http://graphml.graphdrawing.org. [Accessed 3 12 2018].

[10] U. Brandes, M. Eiglsperger and J. Lerner, "GraphML Primer," [Online]. Available: http://graphml.graphdrawing.org/primer/graphml-primer.html. [Accessed 3 12 2018].

[11] Social Media Research Foundation, "NodeXL: Network Overview, Discovery and Exploration for Excel," [Online]. Available: https://archive.codeplex.com/?p=NodeXL. [Accessed 3 12 2018].

[12] C. Müller, "SONIVIS: Social networks in virtual information spaces – a wiki-approach," in *Presentation at the XXVIII Sunbelt Social Network Conference*, St. Pete (USA), 2008.

[13] GEXF Working Group, "GEXF 1.2draft Primer," 28 3 2012. [Online]. Available: https://gephi.org/gexf/1.2draft/gexf-12draft-primer.pdf. [Accessed 3 12 2018].

[14] The Gephi Consortium, "Gephi: The Open Graph Viz Platform," [Online]. Available: https://gephi.org. [Accessed 3 12 2018].