GML: A portable Graph File Format

Michael Himsolt
Universität Passau, 94030 Passau, Germany
himsolt@fmi.uni-passau.de

Abstract

GML, the **G**>raph **M**odelling **L**anguage, is our proposal for a portable file format for graphs. GML's key features are portability, simple syntax, extensibility and flexibility. A GML file consists of a hierarchical key-value lists. Graphs can be annotated with arbitrary data structures. The idea for a common file format was born at the GD'95; this proposal is the outcome of many discussions. GML is the standard file format in the Graphlet graph editor system. It has been overtaken and adapted by several other systems for drawing graphs.

Introduction

There are many different file formats for graphs. The capabilities of these file formats range from simple adjacency lists over adjacency lists with labels or coordinates to complex formats which can store arbitrary data. This has lead to an almost "babylonic" 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. The initiative for that was born at GD'95. As a guideline, there are are several text and graphics interchange formats, like RTF, SGML, HTML, Postscript, GIF, TIFF, to name just a few.

Why do programs still use their own formats ? 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. For example, Encapsulated Postscript defines that a drawing may contain a bitmapped version which can be used by not Postscript capable programs. Software which does not understand Postscript can use this picture for a preview.

Another concern is efficiency. One should not expect a universal format to be more efficient than one which 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 which prevents the use of both an optimized native format, and a second interchange format.

Requirements

Which features are neccessary for a common file format ? First, the 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.

A first example

```
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
    labe
    "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"
   ]
 ]
```

Figure 1: GML description of a circle of three nodes

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 & 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 a `IsPlanar` attribute to the graph.
This may lead to a situations 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`.

Syntax

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

Figure 2: The GML Grammar in BNF Format.

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 neccessary 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 ist 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.

How Graphs and Other Data Structures are Represented

Up to this point, GML is not related to graphs. This is intentional: GML is designed to represent arbitrary data structures. In this section, we will show in detail how to represent graphs, and then briefly discuss data structures in general.

A note on our notion

To simplify our notion, we observe that a GML file defines a tree. Each node in the tree is labeled by a key, and leaves have integer, floating point or string values. We will use the notion

$.k_1.k_2. ... .k_n$

to specify a path in the tree where the nodes are labeled by keys k_1, k_2, ... k_n. We will also use the notion

$x.k_1.k_2. ... .k_n$

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

Graphs

```
graph [
  node [
    id 7
    label "5"
    edgeAnchor "corners"
    labelAnchor "n"
    graphics [
      center [ x 82.0000 y 42.0000 ]
      w 16.0000
      h 16.0000
      type "rectangle"
      fill "#000000"
    ]
  ]
  node [
    id 15
    label "13"
    edgeAnchor "corners"
    labelAnchor "c"
    graphics [
      center [ x 73.0000 y 160.000 ]
      w 16.0000
      h 16.0000
      type "rectangle"
      fill "#FF0000"
    ]
  ]
  edge [
    label "24"
    labelAnchor "first"
    source 7
    target 15
    graphics [
      type "line"
      arrow "last"
      Line [
        point [ x 82.0000 y 42.0000 ]
        point [ x 10.0000 y 10.0000 ]
        point [ x 100.000 y 100.000 ]
        point [ x 80.0000 y 30.0000 ]
        point [ x 120.000 y 230.000 ]
        point [ x 73.0000 y 160.000 ]
      ]
    ]
  ]
]
```

Figure 3: A larger example. This graph is a edited text file which was generated by the Graphlet system.

We have already shown a simple graph in Figure 1. Figure 3 shows a larger example. 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 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, and is undirected if that attribute is omitted. In an undirected graph, `.graph.edge.source` and `.graph.edge.target` may be assigned arbitrarily. There are two reasons why we did not define separate representations for directed and undirected graphs. First, it would have made the parser more complex, especially in applications that read both directed and undirected graphs. Second, if graphics get involved, source and target have a meaning even for undirected graphs: if an edge is represented by a polyline, then the sequence of points implies a direction on the edge.

With these conventions, a simple parser for a Graph in GML works in four steps:

1. Read the file and build the three.
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 can of course be integrated into the other steps. This gains efficiency; we have however observed that the overhead is acceptable, especially if all attributes are saved. Also, it makes it easier to extract data which is attached to nodes, edges and graphs, especially if the program wants to preserve unknown data.

Writing Graph Files

Writing graphs in GML format is straightforward. All it needs are loops that run through the graph and print the nodes and edges. Figure 4 shows a sample skeleton for a program.

```
procedure print (g : Graph)
begin
  print "graph ["
  foreach node n in g do
    print "node ["
    print "id", n.id
    (* Insert other node attributes here *)
    print "]"
  done
  foreach edge e in g do
    print "edge ["
    print "source", e.source.id
    print "target", e.target.id
    (* Insert other edge attributes here *)
    print "]"
  done
  (* Insert other graph attributes here *)
end
```

Figure 4: Program skeleton to write a graph in GML format

Restrictions

There are only two restrictions for graphs:

1. The values of the `.graph.node.id` elements must be unique *within the graph*.
2. Each edge must have `.graph.edge.source` and `.graph.edge.target` attributes.

We do not require that all nodes have a `.id` field since this field is not neccessary for isolated nodes. Of course, all other nodes need such a field.

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 an `.x.id` field if neccessary.

Order of Attributes

GML does usually not require that attributes come in a specific order in the file. More specific, 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 in a list, then the parser integrated intomust preserve their order.

Unknown Attributes

GML is designed so that any application can add its own features to graphs, nodes and edges. Of course we cannot expect that all applications can understand all attributes. There are two ways to deal with foreign data. The simple one is to ignore them. Unfortunately, this means they get lost when a new file is written. For example, a program which does a simple graph theoretic transformation would throw away any graphics data.

The more complex solution is to save all attributes in a generic structure, and write them back when a new file is written. This guarantees that not data is lost, and will be appreciated by users. However, it can cause consistency problems if the application changes the graph, as illustrated in the next section.

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 my 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.

Graphics And Other Foreign Data

GML intentionally does not define any standards on how to represent graphics or other system dependent information. This is done because there are already many data formats for graphics, so we can use one of these formats. It also seems unlikely that a restriction to a certain format will really help.

Here are however some recommendations which will help for a better interchangeability of data:

- Use common formats if possible. For example, bitmapped graphics are often stored as *GIF* and *JPEG* files, which can be read on many platforms.
- Do not translate external data into GML, use an external file instead.

List of Keys

Global Defined Keys

`.id` *int*
: Defines an identification number for an object. This is usually used to represent pointers.

`.label` *string*
: Defines a label attached to an object.

`.comment` *string*
: Defines a comment embedded in a GML file. Comments are ignored by the application.

`.Creator` *string*

Shows which application created this file and should therefore only be used once per file at the top level. `.Creator` is obviously unsafe.

`.graphics` *list*
: Describes graphics which are used to draw a particular object.Within graphics, the following keys are defined:

  `.graphics.x` *float*
  : Defines the x coordinate of the center of the object.

  `.graphics.y` *float*
  : Defines the y coordinate of the center of the object.

  `.graphics.z` *float*
  : Defines the z coordinate of the center of the object.

  `.graphics.w` *float*
  : Defines the width of the object.

  `.graphics.h` *float*
  : Defines the height of the object.

  `.graphics.d` *float*
  : Defines the depth of the object.

  Coordinates are pixel coordinates on a standard 72 dpi drawing area. Applications may use them as screen coordinates.

Keys for Graphs

`.graph` *list*
: Describes a graph.

`.graph` *int*
: Specifies whether a graph is directed (1) or undirected (0). Default is undirected.

`.graph.node` *list*
: Describes a node. Each non isolated node must have an attribute `.graph.node.id`, who's value must be unique within the graph.

`.graph.edge` *list*
: Describes an edge. Each edge must have `.graph.edge.source` and `.graph.edge.source` attributes.

`.graph.edge.source` *int*
`.graph.edge.target` *int*
: Specify the end nodes of an edge by their `id` keys.

Other File Formats

In this section, we briefly discuss the designs of some other file formats and their relationship to GML.

Simple Adjacency lists

```
1 0.400927 0.939745 2
2 0.314021 0.911935 3
3 0.407879 1.       4
4 0.205098 0.799537 5
5 0.174971 0.689455 1
```

Simple adjacency list file format (from VEGA)

Many systems store graphs as simple adjacency lists, sometimes enriched with labels or coordinates. For example, each line describes a node and its adjacent edges. Often, an adjacency list is terminated by the end of the line.

While these formats are convenient and easy to implement, it has several disadvantages. First, it is not expansible. Second, labels are usually restricted to one character or a single word. Further, the degree of a node is limited on systems which do not support arbitrary line lengths. Also, these formats are usually not extensible.

GraphEd

```
GRAPH "" =
1 {$ NS 32 32 $} ""
 2 ""
 ;
2 {$ NS 32 32 $} ""
 3 ""
 ;
3 {$ NS 32 32 $} ""
 1 ""
;
END
```

GraphEd file format

GraphEd uses a file format which is in spirit very similar to the one which is presented in this paper. Table However, its syntax is more complex than necessary in several aspects:

1. There are several ways to represent lists: `[` ... `]`, `{$` ... `$}` , *number* ... `;` , and `GRAPH` ... `END`.
2. Some syntax elements are superficial, like the "=" after the `GRAPH` keyword.
3. GraphEd separates graph structure and labels from the rest, and uses a different syntax for each part, whereas GML combines them. GraphEd's approach separates the topological structure more clearly from the attributes, but our experience has shown that this is not neccessary.

On the positive side, the format supports generic attributes (inside `{$` ... `$}`) which are similar to those in GML. The main difference is that GraphEd's attributes have a key and a list of values, while GML uses only have one value per key, which simplifies the data structure a lot. GraphEd's data structures need lists of attributes and lists of values, while GML only needs a one list structure for the list of key-value pairs.

`dot` Format

```
digraph G {
  subgraph cluster_0 {
    label = "hello world";
    a -> b;
    a -> c;
    color = hot_pink;
  }
  subgraph cluster_1 {
    label = "MSDOT";
    style = "dashed";
    color = purple;
    x -> y;
    x -> z;
    y -> z;
    y -> q;
  }
  top -> a;
  top -> y;
  y   -> b;
}
```

`dot` file format

The `dot` file format uses annotated adjacency lists similar to GraphEd, and is one of the most powerful formats around. Application defined attributes can be attached to graphs, nodes and edges. Edges are represented by `->` arrows in directed and by `-` in undirected graphs. `dot` is also the only format (as known to the author) which supports subgraphs. However, the syntax uses slightly more elements than neccessary, like the `->` or the `=` for assignments. Directed and undirected graphs use a slightly different syntax.

Tom Sawyer Software Format

```
// Graph Layout Toolkit
Hierarchical Layout
// minimumSlopePercent
20
// Nodes
// Node
Untitled2
Untitled42
// Edges
// Edge
Untitled42
Untitled2
```

Tom Sawyer file format

Their file format of Tom Sawyer Software uses keys and lists of values. Each key is started by "`//`", and followed by a list of values, each on its line. The syntax does not define a hierarchical list structure, although that can be modeled with dummy begin/end keys. The format is extensible; new key/value elements can be added through a C or C++ interface.

Conclusion

The GML file format is a flexible, portable file format for graphs which is easy to implement. GML is currently used by the Graphlet system, and software for converting GML from and to other formats is under construction. GML will also be supported by several other systems.

Acknowledgements

I would like to thank everybody on the file format mailing list, especially Jim Baker, Yves Carbonneaux, Jim Blythe, David Dobson, Arne Frick, Serge Heiden, Uwe Hubert, Thomas Lange, Brendan Madden, Stephen North, Tomaz Pisanski, Georg

Sander, Roberto Tamassia, and Richard Webber.

References

1. M. Himsolt: GraphEd: A Graphical Platform for the Implementation of Graph Algorithms. In R. Tamassia, I.G. Tollis (editors): Graph Drawing, Lecture Notes in Computer Science **894**, pp. 182-193. (1994)
2. Information on GML is availabe on the world wide web at `http://www.uni-passau.de/Graphlet/GML`.
3. North, S.C., Koutsofios, E.: *Applications of Graph Visualization*. In: Graphics Interface'94, pages 235-245, 1994.
4. Tom Sawyer Software: *Graph Layout Toolkit Reference Manual*, Berkeley, CA (1992-1996).
5. Information on VEGA is available on the World Wide Web at `http://www.mat.uni-lj.si/ftp/ftpout/vegadoc/htmldoc/vega03.htm`.
6. Details on HTML and the ISO 8859-1 charset are available on the World Wide Web at `http://www.w3.org/`.

Michael Himsolt