Université Nice SOPHIA ANTIPOLIS

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

MASCOPT - A NETWORK OPTIMIZATION LIBRARY: GRAPH MANIPULATION

Jean-François Lalande, Michel Syska, Yann Verhoeven

Projet MASCOTTE

Rapport de recherche
ISRN I3S/RR–2004 -12–FR

Avril 2004

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ :

Dans ce rapport nous introduisons une bibliothèque Java dédiée aux problèmes d'optimisation dans les réseaux, notamment pour l'écriture de prototypes logiciels. Nous présentons la première étape de développement qui concerne les problèmes algorithmiques sur les graphes. Cette bibliothèque open source appelée Mascopt (contraction de Mascotte et d'optimisation) offre l'implémentation d'un modèle générique de graphe. La bibliothèque a été conçue dans l'esprit des modèles orientés objet et privilégie l'accessibilité et la facilité d'utilisation plutôt que la vitesse d'exécution des programmes. Nous montrons aussi comment ce modèle de graphes peut être étendu et spécialisé à des cas d'utilisation particuliers en utilisant des mécanismes objets simples. Enfin, une description rapide des fonctionnalités de Mascopt est présentée. Ainsi, les développeurs familiers avec les concepts objets pourront commencer à l'utiliser pour leurs propres expérimentations.

MOTS CLÉS :

Optimisation, Graphes, Réseaux, Java, XML

ABSTRACT:

This report introduces a Java library whose objective is to provide tools for solving some network optimization problems and that may be used to write prototype software. We describe here the first step of the development which concerns algorithmic graph problems. This open source library named Mascopt includes an implementation of a generic model of graph. This library has been designed with an object-oriented model and aims to be user friendly rather than focusing on speed of execution. We show how the model can be extended and dedicated to a user application by using simple object mechanism. We also present a basic description of the Mascopt functionalities so that developers, who are familiar with objects, can use effectively for their own experimentations.

KEY WORDS :

Optimization, Graphs, Networks, Java, XML

Mascopt - A Network Optimization Library: Graph Manipulation[1]

Jean-François Lalande Michel Syska Yann Verhoeven

Avril 2004

[1]Work partially funded by the European project CRESCCO.

Abstract

This report introduces a JAVA™ library whose objective is to provide tools for solving some network optimization problems and that may be used to write prototype software. We describe here the first step of the development which concerns algorithmic graph problems. This open source library named MASCOPT includes an implementation of a generic model of graph. This library has been designed with an object-oriented model and aims to be user friendly rather than focusing on speed of execution. We show how the model can be extended and dedicated to a user application by using simple object mechanism. We also present a basic description of the MASCOPT functionalities so that developers, who are familiar with objects, can use effectively for their own experimentations.

Keywords: Optimization, Graphs, Networks, Java, XML

Dans ce rapport nous introduisons une bibliothque JAVA™ ddie aux problmes d'optimisation dans les rseaux, notamment pour l'criture de prototypes logiciels. Nous prsentons la premire tape de dveloppement qui concerne les problmes algorithmiques sur les graphes. Cette bibliothque open source appele MASCOPT (contraction de Mascotte et d'optimisation) offre l'implmentation d'un modle gnrique de graphe. La bibliothque a t conue dans l'esprit des modles orients objet et privilgie l'accessibilit et la facilit d'utilisation plutt que la vitesse d'excution des programmes. Nous montrons aussi comment ce modle de graphes peut tre tendu et spcialis des cas d'utilisation particuliers en utilisant des mcanismes objets simples. Enfin, une description rapide des fonctionnalits de MASCOPT est prsente. Ainsi, les dveloppeurs familiers avec les concepts objets pourront commencer l'utiliser pour leurs propres exprimentations.

Mots clefs: Optimisation, Graphes, Rseaux, Java, XML

Contents

1 Introduction 4
2 Existing Libraries 4
3 The MASCOPT Graphs Object Oriented Model 5
4 Architecture 7
4.1 Packages 7
4.2 Main Classes Description 8
5 Manipulation of Objects 9
5.1 Valuation System 9
5.2 Internal Built Information 9
5.3 Sets 10
5.4 Genericity and Factories 10
5.5 Shared Mode 11
6 A Short Case Study 12
7 File Saving 12
7.1 MGL Format 12
7.2 How to Extend a Format: the MGX Example 13
8 Graphical User Interface 14
8.1 The Editor 14
8.2 The Viewer 14
9 Future Work 15
10 Acknowledgment 15

A Multicommodity Flow with CPLEX **18**

B DTD **22**

1 Introduction

Several graph libraries have been written but dedicated to precise types of problems. These libraries provide efficient functionalities mainly based on the fact that their implementations match the specific problem they intend to solve. MASCOPT[1] is an Open Source[2] library for general network optimization problems in which networks are modeled as graphs. This library is based on our experience on the RNRT project PORTO [1] which was dedicated to solve dimensioning problems on WDM networks. This document presents the phases of the MASCOPT development project which aims to provide graphs manipulation tools so that one could easily implement any algorithm on graphs (see the workplan in section 9). As this presentation does not include the network modeling, "MASCOPT library" will refer to the MASCOPT's graph workpackage in the remainder of this paper.

Compared to the LEDA platform [10][6], which is the main reference for graph library implementation, we propose an open source platform whose main goal is to be easily understandable and to take advantage of the broad range of classes provided by the JAVA™ language [7]. Whereas the LEDA system guarantees an efficient manipulation of the components of graphs, MASCOPT focuses on the object-oriented way of implementing our model of graphs. We want to insure that the model stays as generic as possible but still provides facilities for special usage and each of its components is reusable. In this way, our development is mainly driven by the consideration of concepts described in [8].

[1] `http://www-sop.inria.fr/mascotte/mascopt`

[2] Inter Deposit Digital Number at Program Protection Agency: IDDN . FR . 001 . 100002 . 000 . S . P . 2004 . 000 . 31235.

2 Existing Libraries

As previously said, MASCOPT is not a rewrite of LEDA in the JAVA™ language. It differs also from the GTL (Graph Template Library [5]) which is an extension of the C++ Standard Template Library to graphs and fundamental graph algorithms. Indeed, GTL does not allow easy deriving of graph classes (objects are attached to nodes with the use of STL maps), and neither LEDA nor GTL are free (MASCOPT is under the LGPL license) and written in JAVA™ (which allows a different type of efficiency : maybe not at runtime but while writting the application).

Although others JAVA™ libraries have been developped and delt well with the implementation of graphs, these libraries are not always dedicated to graphs such as JDSL [4] which is more a generic data structure implementation. Actually, we have found three main libraries which are really close to MASCOPT in terms of functionalities. The figure 1 presents the features of JDSL, GFC [2], OPENJGRAPH [9], and MASCOPT.

As MASCOPT is Open Source, and written in JAVA™, it is quite easy to call external libraries or JAVA™ bytecode classes. We give an example in appendix A of how to interface MASCOPT with the CPLEX solver, which is a commercial tool from ILOG widely used to solve linear programs (LP) and integer linear programs (ILP) as well. To illustrate that feature, we present in section 6 the concrete example of solving the multicommodity flow problem in a network modeled by a graph valuated with capacities on edges. If the data cannot be accessed directly in memory by an external program, we also provide a possibility to dump the data in a XML file, a quite readable and understandable standard format (see section 7).

	JDSL	GFC	OJG	MASCOPT
Open Source			√	√
Graphs	√		√	√
Directed Graphs	√	√	√	√
Sets	√	√		√
Chains			√	√
Traversals	√		√	√
Trees	√		√	
Priority Queues	√			
Heaps	√			√
Shared Mode		√	√	√
File Saving	√		√	√
Valuation System				√
Visualization		√	√	√
Editor			√	√
Drawing Alg.		√	√	

Figure 1: Comparison between different graph libraries

3 The mascopt Graphs Object Oriented Model

The graph description of MASCOPT is based on the standard mathematical definition of graphs. A graph $G = (V, E)$ is composed of a vertex set $V = \{v_1, \ldots, v_n\}$ and an edge set $E = \{e_1, \ldots, e_m\}$. Each element of V or E must be instantiated by the user giving the possibility to freely manipulate vertices and edges. G is a valid formed graph if V and E are coherent as defined in definition 1.

Definition 1 *E and V are coherent iff* $\forall e = (v_1, v_2) \in E,\ v_1 \in V,\ v_2 \in V$

This mathematical model of the MASCOPT graph library is using extensively the possibilities of object management. No general object is representing a graph environment where graphs or elements of graphs are created. For example, we want to avoid the construction of node (vertex [3]) by the graph as it is done in LEDA. Given a graph G:

Listing 1: Creation of a node in LEDA

```
node u = G.new_node();
```

The consequence of giving the direct access to nodes (vertices) and edges to the user is that, first, the user has to build the sets in order to build the final graph. It can appear less practicable to have to build V and E but in that way, the user gains the ability to reuse those sets in another graph. For example, we are then able to build two graphs $G_1 = \{V_1, E_1\}$ and $G_2 = \{V_1, E_2\}$ that share a vertex set. The use of E_1 and E_2 does not forbid to add the same e_i in these sets which means that the two graphs are not different by their vertex or edge objects but only considering their vertex and edge sets: in the previous case, the edge sets are different but

[3] In MASCOPT *nodes* are elements belonging to a network while *vertices* are elements belonging to a graph that may represent that network topology.

the vertex sets are the same (see section 5.5 for more details).

For each type of element (graph, vertex, edge...) we have grouped the common attributes and methods in *abstract* classes. This way, we try to take full advantage of the inheritance principle. Another advantage of this method of building graphs is that the user has the possibility to write his own vertex and edge classes. Then the graph and sets objects do not have to be rewritten but simply derived from the MASCOPT classes. The user only have to write a *factory* for its new objects: he is able to use all the already implemented MASCOPT algorithms that use factories (see section 5.4 for a more detailed presentation of factories).

We show a diagram of the implementation of our model in figure 2. This figure presents the abstract classes which model graphs and paths construction with the notion of sets, vertices, and edges. We briefly detail how each abstract class works and interact with others.

An **AbstractGraph** represents a graph. It needs coherent **AbstractVertexSet** and **AbstractEdgeSet** (see definition 1). The **AbstractVertexSet** (resp. **AbstractEdgeSet**) contains the vertices (resp. edges) of the graph, implemented by **AbstractVertex** (resp. **AbstractEdge**). **AbstractPath** is a special **AbstractGraph** which builds a path on a graph.

Along with these base abstract classes, we provide two different implementations of graphs: the class **Graph** which represents a valued undirected graph and **DiGraph** which represents a valued directed graph. **Graph**s and **Digraph**s classes have no real functionalities. It just provides a user friendly and comprehensive access to the graph, wrapping the abstract classes. Note

Listing 2: Creation of a DiGraph

```
import mascoptLib.graphs.*;

public class Sample {
        public static void main (String[] args){

                Vertex n0 = new Vertex();
                Vertex n1 = new Vertex();
                Vertex n2 = new Vertex();
                Arc a0 = new Arc(n0,n2);
                Arc a1 = new Arc(n1,n2);

                VertexSet V = new VertexSet();
                V.add(n0); V.add(n1); V.add(n2);

                ArcSet E = new ArcSet(V);
                E.add(a0); E.add(a1);

                DiGraph G0 = new DiGraph(V,E);
        }
}
```

also that it gives the ability to the user to implement his own type of graphs overloading the five main classes. The last classes concern the factories, described in section 5.4.

We just present a basic example of how to build a digraph in listing 2. This example may seem to be a little bit longer than the code required by some other graph library, but we think that it is quite natural to build elementary objects first (vertices and edges) and to finish by building the graph G_0, which is drawn in figure 3 (the values drawn are explained in section 5.1).

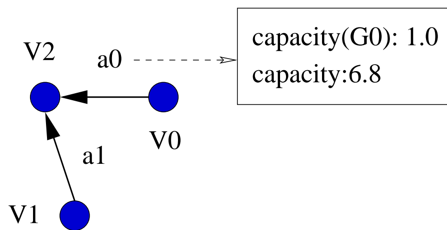

Figure 3: Valued Graph G_0

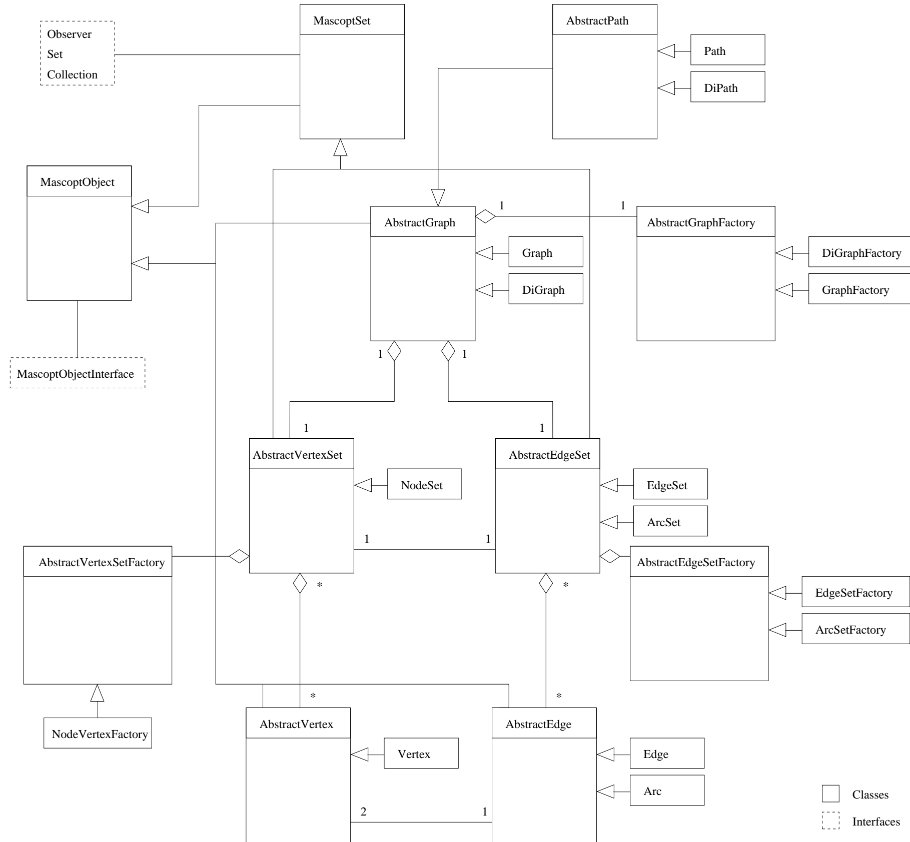

Figure 2: Class diagram of MASCOPT graph library

4 Architecture

4.1 Packages

MASCOPT consists of eight Java packages. Each package provides several interfaces leading the user to build classes similar to the one that are provided.

- **mascopt.abstractGraph**: package of abstract graph library. It implements most of the functionalities of the classes shown in figure 2.
- **mascopt.algos.abstractalgos**: package which provides generic algorithms mainly on **AbstractGraph** and other objects of the **mascopt.abstractGraph** package.
- **mascopt.graphs**: package for graph and digraph.
- **mascopt.gui**: Graphical User Interface. This package is a graphic layer which is plugged over the **mascopt.abstractGraph** package.

- **mascopt.io.graph**: package of input/output capabilities. Several formats are provided to store objects of package **mascopt.graphs**.
- **mascopt.io.util**: utilities to manipulate files written by the **mascopt.io.graph** package.
- **mascopt.util**: utilities and algorithms which are not dedicated to graphs.

4.2 Main Classes Description

To simplify the reading, we do not present the abstract classes but the main classes of the **mascopt.graphs** package only (see the corresponding javadoc and the web site for an exhaustive documentation).

Simple Elements

MascoptObject implements the valuation system to store **String**, **Double** and **Integer**. The valuation system is detailed in section 5.1.

Vertex object is the most basic element which can be built. A **Vertex** provides information about its neighbors, its degree, the edges or arcs exiting or entering it. It derives from **AbstractVertex** and **MascoptObject**.

Edge and Arc. An **Edge** object is built using two **Vertex** objects. Given one vertex, the **Edge** object provides facilities to pass through this edge when covering a **Graph**. **Edge** is derivated from **AbstractEdge** and **MascoptObject**. The **Arc** class is derived from **AbstractEdge** and implements directed edges.

Sets

MascoptSet is derived from **MascoptObject**. It implements the **Set** and **Collection** interfaces. It provides an efficient implementation of **HashSet** characteristics. It adds extra functionalities to perform operations on valuation of objects in sets.

VertexSet. The **VertexSet** class is derived from **AbstractVertexSet** and **MascoptSet**. It allows to group **Vertex** objects into a set.

EdgeSet and ArcSet. The **EdgeSet** and **ArcSet** class are derived from **AbstractEdgeSet** and **MascoptSet**. It allows to group **Edge** objects, respectively **Arc** objects, into a set.

Graph and DiGraph

The **Graph** class constructs a non directed graph using a vertex set and an edge set. It guaranties that, at any time, the **EdgeSet** and **VertexSet** objects stay coherent as defined in definition 1. It provides facilities to copy graphs and construct subgraphs. The **DiGraph** class constructs directed graph with **VertexSet** and **ArcSet** objects. **Graph** and **DiGraph** derive from **AbstractGraph**.

Path and DiPath

A **Path** object gives the ability to build a path over a **Graph**. The path can be merged with another path with same start and end vertex, giving a multi-path. The path can be covered through the graph. The path is also considered as a graph and derives from **Graph**. The **DiPath** object is a directed **Path**. It derives from **DiGraph**.

5 Manipulation of Objects

In order to make the use of the library easy, not only MASCOPT provides a set of methods enabling to manipulate the objects provided, but also internal mechanisms ensuring that any object behaves as expected.

5.1 Valuation System

As previously said, MASCOPT deals mainly with two types of objects, firstly simple objects which derive from the class **MascoptObject** and secondly sets which contain the former objects, deriving from the class **MascoptSet**. Of course all the methods needed in the process of writing algorithms are provided.

The main characteristic of the simple objects deriving from **MascoptObject**s is that they can have multiple values associated with them. At the time of writing, these values can only be of type **String**, **Integer** or **Double**. All the accessors needed to access and modify the values are provided, their names trying to be intuitive for a JAVA^{TM} user. Thus, the accessor for a value of type **Integer** is called **getIntegerValue**. For an easy use of the different types of values we implemented an accessor for each data type and the user must know the type of the stored value or use the method giving that information, namely **getValueDataType**. The default accessor is **getValue** which outputs a **String** such that any data type can be represented.

We choose to restrict the types of the values in order to guarantee that the information about the type is not lost during the process of dumping the graph objects in the file. An illustration of this feature is presented in section 7.1. These methods are common to all the inherited members of the class **MascoptObject** but for every class some methods are implemented to access the specific attributes. For example it is possible to ask a **Vertex** for its coordinates with the methods **getX** and **getY** and you can ask an **Edge** about its extremities with methods such as **getVertices** that outputs a set containing the vertices linked by that edge.

Listing 3: Valuation of vertex v0 and arc a0

```
v0.setValue("function","start");
a0.setDoubleValue("capacity", new Double(6.8));
a0.setIntegerValue("length", new Integer(110));
```

As an example, if we want to associate values to some of the previously constructed vertices and arcs, we just have to write the following (listing 3) to obtain the valued graph of figure 3. Nevertheless, thanks to the object oriented design of MASCOPT any user can redefine its own objects and store their own values, of any chosen type.

5.2 Internal Built Information

As the MASCOPT library is designed to be useful when writing graph algorithms programs, we provide standard methods to ask structure information to the objects of the library. For example, when asking for the neighbors of a vertex u, the library returns the vertex set $vs(u)$ to the user. We choose here to give a quick access to the data $vs(u)$ by pre-building the set. This is really different from building "on-the-fly" information which will need some time before returning the data, which could be a problem when the information is often accessed in the program. Nevertheless, some time is lost when building objects and when updating the internal data (as $ns(u)$) *e.g.* when modifying a graph (as

inserting some edges). As a result, the objects take more space in memory, which can be seen as a strong limitation. However, all the computable information is directly available in objects and we think it is worth doing it. Moreover, as usual, a tradeoff had to be found between space and time.

5.3 Sets

As MASCOPT is meant to deal with combinatorial structures such as graphs and in order to do so, it must be able to store objects representing the various elements in sets. Thus a complete hierarchy deriving from a base class named **MascoptSet** has been developped. The methods accessible from this class reflect the fact that sets are an implementation of **HashSet** class. Indeed the user can add, remove, test the belonging of objects with standard functions of JAVA™. Moreover some more specific methods have been implemented. As for any object deriving from **MascoptObject**, values can be assigned to sets but some specific methods enable the user to manage all the elements of a set at once, *e.g.* a same value can be assigned at once to all the elements of a set with **setValueForAllElements**. As previously, each deriving class implements particular methods that enable the user to access to specific attributes. As an example, a set containing some vertices requires some different methods than a set containing some edges. Even if in MASCOPT the names are the same, the methods **add** and **remove** present in each set are different because some specific checkings have to be done in each case.

As sets are an important part of MASCOPT, the notion of subset is also implemented. But then some new problems arise such as the one of coherence between sets (as defined in definition 1). Indeed we must guarantee that a set (of any sort) that is defined as the subset of another one will stay so, whatever operation is applied to it or to its superset. Thus, if the user has defined the **VertexSet**s *VS1* and *VS2*, *VS2* being a subset of *VS1*, a new vertex can be added in *VS2* if and only if it already belongs to *VS1*. Moreover, deleting a vertex in *VS1* will automatically delete this vertex from *VS2*. Of course there are equivalent behaviour for all kind of sets, thanks to the inheritance of methods in the model. Since the class **Graph** relies on the classes **VertexSet** and **EdgeSet**, the coherence is naturally insured between a graph and its sub-graph.

This feature is implemented via basic message passing between different objects and works efficiently.

5.4 Genericity and Factories

Some algorithms are based only on generic properties of graphs and don't take into account particularities of the different structures such as the fact that a graph is directed or not. Without a special mechanism, the implementation of the algorithm has to be rewritten for each kind of structure since some structures are independant. This can be avoided in MASCOPT for two main reasons, first because the object oriented similar structures share a common ancestor and secondly because we use the concept of factory [11]. This latter concept is directly taken from the design pattern theory where it is often used.

The code of the algorithm is written into the common ancestor and can be used in any derived class where a factory class has been implemented. Of course, this is possible only if the algorithm uses only generic attributes, but this is often the case with graph algorithms.

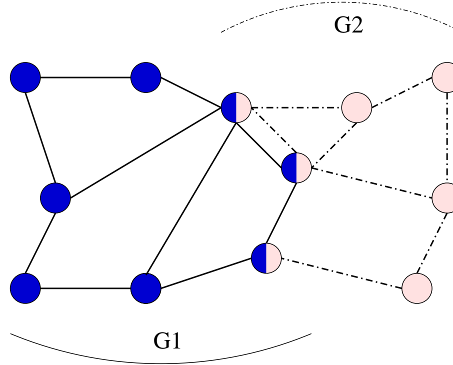

Figure 4: Share of vertices and edges between graphs $G1$ and $G2$

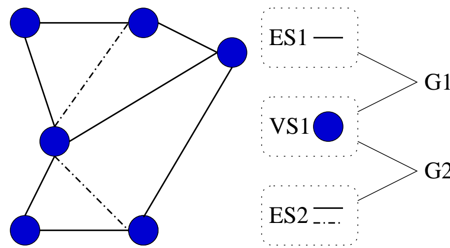

Figure 5: Share of the set of vertices $VS1$ with two different edge sets, $ES1$ and $ES2$

5.5 Shared Mode

The way we choose to build graphs gives the user the ability to share objects between graphs or sets. This possibility avoids the duplication of shared objects, for example, when constructing a lot of paths on a graph: the vertices and edges are the same objects in all paths. Nevertheless, some disadvantages appear: when asking to a vertex its exiting edges, the user has to precise the graph it considers, because the vertex may appear in several graphs.

The figure 4 presents two graphs sharing three vertices and one edge (this edge is displayed two times in the figure). The figure 5 shows a more complex example: the **VertexSet** object $VS1$ is shared by the two graphs $G1$ and $G2$. It allows to build two different graphs not duplicating all vertices. Note that the edge sets are different. No more than nine edges are instanciated and seven are shared by $ES1$ and $ES2$.

As a concrete example, we can now create in listing 4 a graph $G3$ represented in figure 6, using objects created in section 3. Note that, as a consequence of listing 3, $v0$ and $a0$ are also valued for $G3$.

Listing 4: Creation of DiGraph G3

```
ArcSet E3 = new ArcSet(V);
Arc a2 = new Arc(v0,v1);
E3.add(a0); E3.add(a2);

DiGraph G3 = new DiGraph(V,E3);
```

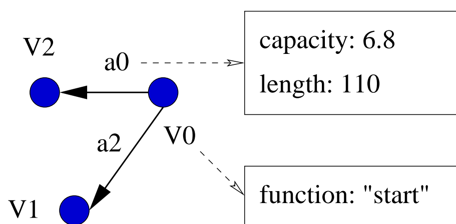

Figure 6: Valued Graph $G3$

6 A Short Case Study

We give a short example of a use of MASCOPT in a network optimization problem: we show how to implement a routing algorithm with multicommodity flow. We use the network model described in [3]. The input data of the problem is a valued digraph $D = (V, E_D)$ and some requests between pairs of verstices of V. These requests are stored in a graph R whose vertex set is shared with D, *i.e.* $R = (V, E_R)$.

The output of the problem is a set of paths satisfying the request *i.e.* giving for a certain amount of flow the route to use to go from the start vertex of the request to the end vertex. All these paths are **DiPath** objects and are based on the graph D which means precisely that they share the same vertex set and that the edge set of each path is a sub-set of the edge set of D. Note that if an arc of D is removed, all the paths using this arc die because of the coherence guarantee of the model: a path must stay connected and all its edges must belong to the underlying graph.

In order to implement this problem with MASCOPT, one has to implement the algorithm given in [3] which computes flow on D. That is, for each request $r = (v_s, v_t) \in E_R$, we extract a set of paths which leads, using different routes, all the flow from v_s to v_t. Next we aggregate all those paths in one multi-path by merging them in a special **DiPath** object, and make sure that the merge is correct. And finally we just need to plug our algorithm to the I/O classes so we can choose a file containing a graph and give it as an input to the algorithm and save the result in a file. We can also visualize directly the result.

The main work for the user is then to write the algorithm of his choice using the different objects provided by MASCOPT; the rest of the program consists in calling the right methods.

7 File Saving

7.1 MGL Format

MASCOPT provides several input and output formats. We only present MGL and MGX which means **M**ASCOPT **G**raph **L**ibrary and **M**ASCOPT **G**raph **E**xtended. MGL format is the MASCOPT's native format and is based on the XML standard, controled by a DTD (see appendix B for more details). It provides a readable description of MASCOPT's objects which can be easily extended for the user's own objects or specializations: as the MGL reader parse an XML file, it can also read a derived file, containing new tags.

One can create his own format implementing the interface **WriterInterface** which consists only in two methods: an **add** method to add objects to write and a **write** method to physically write the file. The file

is quite understandable. The information contained in a MGL file reflects the oriented-object structure of MASCOPT. This is a simple manner to enable the sharing of objects. The listing 5 presents the code of the digraph created in listing 2, 4, and valued in listing 3.

Listing 5: MGL File for G0 and G3

```
<?xml version="1.0" ?>
<!DOCTYPE OBJECTS SYSTEM "ftp://ftp-sop.
    inria.fr/mascotte/mascopt/dtd/mgl_v1.2.dtd">

<OBJECTS>
 <VERTICES>
  <VERTEX id="V0">
   <POSITION>
    <X>50.0</X> <Y>0.0</Y>
   </POSITION>
   <VALUE type="function" dataType="String"> node0
      </VALUE>
  </VERTEX>
  <VERTEX id="V1">
   <POSITION>
    <X>10.0</X> <Y>50.0</Y>
   </POSITION>
  </VERTEX>
  <VERTEX id="V2">
   <POSITION>
    <X>0.0</X> <Y>0.0</Y>
   </POSITION>
  </VERTEX>
 </VERTICES>

 <LINKS>
  <ARC id="AE1">
   <VERTEX_REF idref="V1"/>
   <VERTEX_REF idref="V2"/>
  </ARC>
  <ARC id="AE0">
   <VERTEX_REF idref="V0"/>
   <VERTEX_REF idref="V2"/>
   <VALUE type="Capacity" dataType="Double"> 6.8
      </VALUE>
   <VALUE type="length" dataType="Integer"> 110 </
      VALUE>
  </ARC>
  <ARC id="AE2">
   <VERTEX_REF idref="V0"/>
   <VERTEX_REF idref="V1"/>
  </ARC>
 </LINKS>

 <SETS>
  <VERTEX_SET id="VS0">
   <VERTEX_REF idref="V0"/>
   <VERTEX_REF idref="V1"/>
   <VERTEX_REF idref="V2"/>
  </VERTEX_SET>
  <ARC_SET id="AES0">
   <ARC_REF idref="AE0"/>
   <ARC_REF idref="AE1"/>
  </ARC_SET>
  <ARC_SET id="AES3">
   <ARC_REF idref="AE0"/>
   <ARC_REF idref="AE2"/>
  </ARC_SET>
 </SETS>

 <GRAPHS>
  <DIGRAPH id="G0">
   <VERTEX_SET_REF idref="VS0"/>
   <ARC_SET_REF idref="AES0"/>
  </DIGRAPH>
  <DIGRAPH id="G3">
   <VERTEX_SET_REF idref="VS0"/>
   <ARC_SET_REF idref="AES3"/>
  </DIGRAPH>
 </GRAPHS>
</OBJECTS>
```

7.2 How to Extend a Format: the MGX Example

We give an example of how to extend the graph model when it is required by the user application. MGX is an extended version of MGL. It solves the following problem: as all objects are shared between graphs, the valuation of an object may differ between one graph and another. In section 5.2, the arc $a0$ is valued with $Capacity = 6.8$ and this value is the same in all graphs containing this edge. As a result, we introduce the notion of context, which specify the context where the value is valid. The simplest way of using context is to put the graph itself as context. It expresses directly that a value is valid only in this graph.

With the two graphs $G0$ and $G3$ we can now add contexted values named "Capacity" on arc $a0$ as shown in listing 6. Then, the differences between MGL and MGX files are shown in listing 7. Note that the default value 6.8 is kept on $a0$: without context, the valuation system gives this value. The figure 7 shows the resulting two graphs.

Listing 6: Valuation of arc $a0$ with contexts

```
a0.setDoubleValue("Capacity",G0, new Double(1.0));
a0.setDoubleValue("Capacity",G3, new Double(3.2));
```

Listing 7: Modified part of listing 5

```
<ARC id="AE0">
 <VERTEX_REF idref="V0"/>
 <VERTEX_REF idref="V2"/>
 <VALUE type="Capacity" dataType="Double"> 6.8 </
      VALUE>
 <VALUE type="Capacity" dataType="Double" context="
      G0"> 1.0 </VALUE>
 <VALUE type="Capacity" dataType="Double" context="
      G3"> 3.2 </VALUE>
</ARC>
```

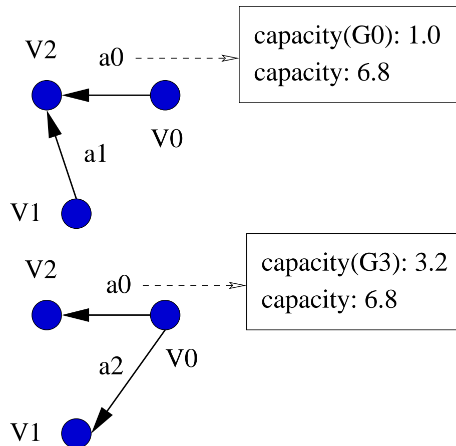

Figure 7: $G0$ and $G3$ with contexted values

8 Graphical User Interface

Two graphic tools have been developed with MASCOPT over the core library. The first tool is a simple editor which enables the graphical creation and edition of graphs, and the second is a full-featured graph viewer, designed to display complex graphs.

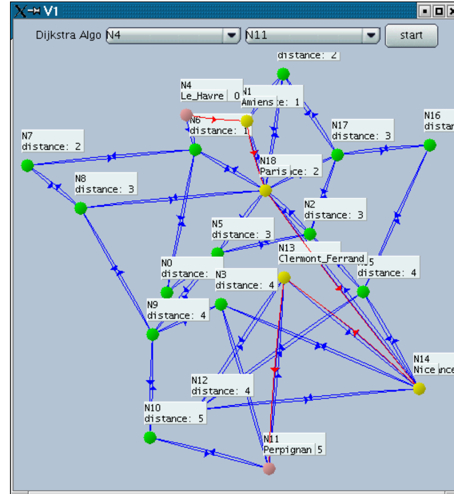

Figure 8: The viewer displaying a graph of some French telecommunication network

8.1 The Editor

The editor allows the addition or removal of vertices and edges for a graph or a digraph. Then, the user can add values on vertices or edges as done with the valuation system by the code. The graph is stored in a file with MGL format or can be exported as an image in the PNG format. Note also that one can of course import a graph from a MGL file.

8.2 The Viewer

The viewer is a multi-layered and multi-view graphical interface. The user can have some views opened on different graphs. The user can also have, in the same view, several graphs displayed at the same time. To hide and display groups of graphs in a view (for example, several paths on a graph), we introduced layers in views which are piled up and can be hidden or displayed independently.

WPX	Workpackage	Description
WP1	Graphs	Modeling and processing of graphs
WP2	Networks	Modeling and processing of networks
WP3	Virtual networks	Interaction between different types of networks
WP4	Linear Programming	Linear optimization in networks
WP6	Experimental data's	Realistic data for network experiments
WP7	Graphical User Interface	Display of graphs and networks
WP8	Input/Output graphs and tools	XML Modeling, Export formats
WP9	Algorithms on graphs or digraphs	Well known algorithms for graphs

Figure 9: MASCOPT's Workplan

The viewer displays **String** labels on vertices and edges, which can contain values stored by the valuation system. Note also that you can import a collection of graphs from a MGL file and put each graph in different layers or views.

The viewer is not only graphically usable. It has been designed to be controlled with JAVA™ code and called to display the results of user's algorithm. The user can instantiate a **MascoptViewer** object which controls all the graphical objects that are useful. This class is a link between basic objects of MASCOPT and their graphical representation. It computes automatically the graphical updates when some objects of the library changes. With this system, the views are not a static representation of graphs. For example, if a value on a vertex is displayed and is changed by an algorithm, the label of the vertex in the graphical interface is also updated.

9 Future Work

Our work in MASCOPT is divided in several workpackages, shown in figure 9. The first public release (1.1) of MASCOPT contains mainly WP1 and a significant part of WP7 and WP8. The next step of our workplan is to implement data structure for networks, based on the packages contained in WP1. Other workpackages will be developed concurrently to explore our research topics, including:

- Data model of the network and the demands.
- Routing end to end connections in a network with capacity limitations.
- Routing under vulnerability constraints (protection and restoration).
- Grooming multiplex (eg. SDH over WDM).

10 Acknowledgment

We would like to thank all the people who actively participated to the MASCOPT project.

- Main authors: Bruno Bongiovanni, Sbastien Choplin, Jean-Franois Lalande, Michel Syska, Yann Verhoeven
- Contributors: Sverine Petat, Smita Rai

Bibliography

[1] Project RNRT PORTO. http://www.telecom.gouv.fr/rnrt/projets/pporto.htm.

[2] IBM alphaWorks. Graph foundation classes for java. http://www.alphaworks.ibm.com/tech/gfc.

[3] M. Bouklit, D. Coudert, J-F. Lalande, C. Paul, and H. Rivano. Approximate multicommodity flow for WDM networks design. In J. Sibeyn, editor, *SIROCCO 10*, number 17 in Proceedings in Informatics, pages 43–56, Umea, Sweden, 2003. Carleton Scientific.

[4] Roberto Tamassia et al. An overview of jdsl 2.0, the data structures libary in java. Technical report, 2003.

[5] M. Forster, A. Pick, M. Raitner, and C. Bachmaier. GTL : Graph template library. University of Passau. http://infosun.fmi.uni-passau.de/GTL.

[6] Algorithmic Solutions Software GmbH. LEDA. http://www.algorithmic-solutions.com/enleda.htm.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[8] David R. C. Hill. *Object-Oriented Analysis and Simulation*. Addison-Wesley Longman, 1996.

[9] Jesus M. Salvo Jr. Openjgraph - java graph and graph drawing project. http://openjgraph.sourceforge.net/.

[10] Kurt Mehlhorn, Stefan Naher, and Christian Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.

[11] Theo D. Meijler, Serge Demeyer, and Robert Engel. Making design patterns explicit in FACE - A framework adaptive composition environment. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 94–110. Springer–Verlag, 1997.

[12] A. Ouorou, P. Mahey, and J.-P. Vial. A survey of algorithms for convex multicommodity flow problems. Technical report, Switzerland, 1997.

Appendix A

Multicommodity Flow with CPLEX

We present a short presentation of a solution for the multicommodity flow problem [12]. It is commonly used when computing some paths for some requests in a network, as presented in section 6. The solution we give here uses the ILOG CPLEX Concert API. It shows how our library can be interfaced with other JAVA™ code.

We consider a graph $G = (V, E)$, where E contains edges with capacities e_c. On this graph, we consider the set of request $\mathcal{D}(G)$ which contains requests d adding flow of size $size(d)$ in v_d^+ and removing this flow in v_d^-. Let us note $x_{e,d}$ the amount of flow used on edge e for request d, $adj(v^+)$ and $adj(v^-)$ the set of edges exiting and entering the vertex v.

The problem is the direct expression of the well known integer or real flow commodity problem where the objective is to minimize $\sum_{d \in \mathcal{D}(G), e \in E} x_{e,d}$. This objective function computes the total capacity used in the network to satisfy all the requests. The following constraints model the integrity of capacity and the fact that the flow is well distributed.

$$\forall e \in E \qquad \sum_{d \in \mathcal{D}(G)} x_{e,d} \leq e_c \tag{A.1}$$

$$\forall d \in \mathcal{D}(G), \forall v \in V, v \notin \{v_d^+, v_d^-\} \qquad \sum_{e^- \in adj(v^-), e^+ \in adj(v^+)} x_{e^-,d} - x_{e^+,d} = 0 \tag{A.2}$$

$$\forall d \in \mathcal{D}(G) \qquad \sum_{e^+ \in adj(v_d^+)} x_{e^+,d} = size(d) \tag{A.3}$$

$$\forall d \in \mathcal{D}(G) \qquad \sum_{e^- \in adj(v_d^-)} x_{e^-,d} = size(d) \tag{A.4}$$

$$\forall e \in E, \forall d \in \mathcal{D}(G) \qquad x_{e,d} \geq 0 \tag{A.5}$$

Listing A.1: Flow conservation code

```
/**
 * Adds a flow system on the network.
 */
public void createFlow(AbstractGraph g_, DiGraph requests_, IloCplex cplex)
{
  try
  {
    // For all requests
    Iterator itArcRequest = requests_.iterator();
    while (itArcRequest.hasNext())
    {
      // Getting request r
      Arc r = (Arc)itArcRequest.next();
      AbstractVertex n_start = r.getSource();
      AbstractVertex n_end = r.getTarget();
      double r_size = 0;

      // Getting the request size
      r_size = r.getDouValue(REQUEST_SIZE);

      // For all vertices
      Iterator itVertex = g_.getAbstractVertexSet().iterator();
      while (itVertex.hasNext())
      {
        // Getting vertex n
        AbstractVertex n = (AbstractVertex)itVertex.next();

        // Flow entering the vertex
        Iterator itEdge = n.getIn(g_).iterator();
        IloLinearNumExpr in = cplex_.linearNumExpr();
        while (itEdge.hasNext())
        {
          AbstractEdge edgeHere = (AbstractEdge)itEdge.next();
          in.addTerm(1.0, getVarEdgeReq(edgeHere,r));
        }

        // Flow exiting the vertex
        itEdge = n.getOut(g_).iterator();
        IloLinearNumExpr out = cplex_.linearNumExpr();
        while (itEdge.hasNext())
        {
          AbstractEdge edgeHere = (AbstractEdge)itEdge.next();
          out.addTerm(1.0, getVarEdgeReq(edgeHere,r));
        }

        // Case of the start
        if (n == n_start)
        {
          cplex_.addEq(out, r_size);
          cplex_.addEq(in, 0);
        }
        else if (n == n_end) // Case of the end
        {
          cplex_.addEq(in, r_size);
          cplex_.addEq(out, 0);
        }
        else // Normal case
        {
          cplex_.addEq(in, out);
        }
      }
    }
```

```
    // Solving
    cplex.solve();
  }
  catch (IloException exception)
  {
    System.err.println("Error in Cplex constraints " + exception);
  }
}
```

To implement this linear program with Ilog Cplex, we use the Concert API which provides a Java^{TM} wrapper to the functions of Cplex. Each equation is implemented with overlapped **while** statements, one block covering the requests and the other covering the vertices. Then, for each vertex, an other **while** loop covers the edges entering or exiting this vertex. The listing A.1 shows how to program equations A.2, A.3, and A.4. This code needs a function called **getVarEdgeReq(AbstractEdge e, Arc r)** which gives the cplex variable corresponding to an edge e and a request r. The most basic way of implementing this is to create a first **HashMap** containing, for each edge e, a second **HashMap** that contains, for each request r, the Cplex variable. We do not show how to implement equations A.1, A.5 and the objective function because the code is quite similar.

After the call of Cplex solver, which try to solve the implemented linear program, we have to read the solution. We can build on the graph the paths covering the flow solutions (several paths per flow and commodity). This part is shown in listing A.2.

Listing A.2: Build of a path for a request r

```
/**
 * Builds a paths covering a part of the flow.
 */
public AbstractPath buildPath(AbstractGraph g_, Arc r)
{
  AbstractVertex n_start = r.getSource();
  AbstractVertex n_end = r.getTarget();
  AbstractVertex n_current = n_start;
  double maxFlowAllocated = MAX_FLOW_PER_REQUEST;
  AbstractPath result = g_.getFactory().newAbstractPath(g_.getAbstractEdgeSet());

  // While the path has not reached the end vertex
  Iterator itArc = g_.getAbstractEdgeSet().iterator();
  while (n_current != n_end)
  {
    AbstractEdge candidate = null; AbstractEdge loopCandidate = null;
    double tryFlowAllocated = 0; double tryLoopFlowAllocated = 0;

    // We search for the best edge to continue the path
    Iterator itEdgeOut = n_current.getOut(g_).iterator();
    while (itEdgeOut.hasNext())
    {
      AbstractEdge eOut = (AbstractEdge)itEdgeOut.next();
      double cplexValue = getCplexValue(eOut,r); // Getting the result value of Cplex
      double reserved = eOut.getDouValue("pathConstructionReserved");
      AbstractVertex destOut = n_current.getConnected(eOut);

      // Searching the best candidate
      if (cplexValue - reserved > tryFlowAllocated && !result.getAbstractVertexSet().contains(destOut))
```

```
        {
          tryFlowAllocated = cplexValue − reserved;
          candidate = eOut;
        }

        // Searching the best loop candidate
        if (cplexValue − reserved > tryLoopFlowAllocated)
        {
          tryLoopFlowAllocated = cplexValue − reserved;
          loopCandidate = eOut;
        }
      }

      // If the path can be constructed
      if (candidate != null)
      {
        maxFlowAllocated = Math.min(maxFlowAllocated, tryFlowAllocated);
        result.concatAbstractEdge(candidate);
        n_current = n_current.getConnected(candidate);
      }
      else // Removing a found loop
      {
        n_current = this.removeLoop(result, loopCandidate, r);
      }
    }

    // Update of the reserved value on this path
    AbstractVertex vertex_covering = result.getAbstractStart();
    while (vertex_covering != result.getAbstractEnd())
    {
      AbstractEdge edge_parcours = result.nextAbstractEdge(vertex_covering);
      double reserved = edge_parcours.getDouValue("pathConstructionReserved");
      edge_parcours.setDouValue("pathConstructionReserved", reserved + maxFlowAllocated);
      vertex_covering = result.nextAbstractVertex(vertex_covering);
    }

    // Storing the flow value of this computed path
    result.setDouValue("flowAllocated", maxFlowAllocated);

    return result;
  }
```

Appendix B

DTD

We formerly wrote the DTD which validates the MGL files that are read. We give the possibility to disable the validation, but the use of DTD is a strong guaranty that the file is well formed. The listing B.1 shows the DTD which controls MGL format. Then, the listing B.2 shows how to build the MGX DTD which inherits of the MGL one.

Listing B.1: DTD of MGL format: mgl_v1.2.dtd

```
<!ELEMENT OBJECTS (VERTICES, LINKS, SETS, PATHS?, GRAPHS)>

<!-- Groups -->

<!ELEMENT VERTICES (VERTEX*)>
<!ELEMENT LINKS (EDGE*, ARC*)>
<!ELEMENT SETS (VERTEX_SET*, EDGE_SET*, ARC_SET*)>
<!ELEMENT PATHS (CHAIN*, PATH*)>
<!ELEMENT GRAPHS (GRAPH*, DIGRAPH*)>

<!-- Infos -->

<!ELEMENT NAME (#PCDATA)>

<!ELEMENT VALUE (#PCDATA)>
<!ATTLIST VALUE type CDATA #REQUIRED>
<!ATTLIST VALUE dataType (String|Integer|Double) "String">

<!ELEMENT POSITION (X, Y)>

<!ELEMENT X (#PCDATA)>
<!ELEMENT Y (#PCDATA)>

<!-- Objects -->

<!ELEMENT VERTEX (NAME?, POSITION?, VALUE*)>
<!ATTLIST VERTEX id CDATA #REQUIRED>

<!ELEMENT EDGE (NAME?, VERTEX_REF, VERTEX_REF, VALUE*)>
<!ATTLIST EDGE id CDATA #REQUIRED>

<!ELEMENT ARC (NAME?, VERTEX_REF, VERTEX_REF, VALUE*)>
<!ATTLIST ARC id CDATA #REQUIRED>
```

```
<!-- Sets -->

<!ELEMENT VERTEX_SET (NAME?, VERTEX_REF*, VALUE*)>
<!ATTLIST VERTEX_SET id CDATA #REQUIRED>

<!ELEMENT EDGE_SET (NAME?, VERTEX_SET_REF, EDGE_REF*, VALUE*)>
<!ATTLIST EDGE_SET id CDATA #REQUIRED>

<!ELEMENT ARC_SET (NAME?, VERTEX_SET_REF, ARC_REF*, VALUE*)>
<!ATTLIST ARC_SET id CDATA #REQUIRED>

<!-- Paths -->

<!ELEMENT CHAIN (NAME?, EDGE_REF*, VALUE*)>
<!ATTLIST CHAIN id CDATA #REQUIRED>

<!ELEMENT PATH (NAME?, ARC_REF*, VALUE*)>
<!ATTLIST PATH id CDATA #REQUIRED>

<!-- Graphs -->

<!ELEMENT GRAPH (NAME?, VERTEX_SET_REF, EDGE_SET_REF, VALUE*)>
<!ATTLIST GRAPH id CDATA #REQUIRED>

<!ELEMENT DIGRAPH (NAME?, VERTEX_SET_REF, ARC_SET_REF, VALUE*)>
<!ATTLIST DIGRAPH id CDATA #REQUIRED>

<!-- Pointers -->

<!ELEMENT VERTEX_REF EMPTY>
<!ATTLIST VERTEX_REF idref CDATA #REQUIRED>

<!ELEMENT EDGE_REF EMPTY>
<!ATTLIST EDGE_REF idref CDATA #REQUIRED>

<!ELEMENT ARC_REF EMPTY>
<!ATTLIST ARC_REF idref CDATA #REQUIRED>

<!ELEMENT VERTEX_SET_REF EMPTY>
<!ATTLIST VERTEX_SET_REF idref CDATA #REQUIRED>

<!ELEMENT EDGE_SET_REF EMPTY>
<!ATTLIST EDGE_SET_REF idref CDATA #REQUIRED>

<!ELEMENT ARC_SET_REF EMPTY>
<!ATTLIST ARC_SET_REF idref CDATA #REQUIRED>

<!ELEMENT SUPER_SET_REF EMPTY>
<!ATTLIST SUPER_SET_REF idref CDATA #REQUIRED>

<!ELEMENT SUPER_GRAPH_REF EMPTY>
<!ATTLIST SUPER_GRAPH_REF idref CDATA #REQUIRED>
```

Listing B.2: DTD of MGX format: mgx_v0.3.dtd

```
<!ENTITY % mgldtd SYSTEM "mgl_v1.2.dtd">

%mgldtd;

<!-- Extended -->

<!ATTLIST VALUE context CDATA #IMPLIED >
```