
EÖTVÖS LORÁND UNIVERSITY
Faculty of Informatics
Department of Algorithms and their Applications

# Binary storage of graphs and related data

BSc thesis

Author:

**Frantisek Csajka**

full-time student

Informatics for Computer Programming BSc

Supervisors:

**Alpár Jüttner, PhD** **Péter Kovács, MSc**

Budapest, 2010

# Acknowledgement

I would like to thank my supervisors Alpár Jüttner, PhD and Péter Kovács, MSc for their support and encouragement during this project.

# Contents

# 1 Introduction

Networks play an important role in wide range of informatics applications. In most cases not only do we need a graph, but also various fields of data related to nodes, arcs and the graph itself. Efficient and flexible storage and handling of these data is an indispensable task. The most popular methods are based on text formats (e.g. XML), which are easily editable without any special program. For example, GraphML and GXL are two of those file formats for storing graphs[6, 7].

In practice, however, we often have to deal with really huge graphs, store them or send them trough the network, so the size of the whole representation can be a major issue. Another important requirement is to make it possible to store the data related to arcs or nodes separated from the graph, which is useful, for example, to minimize the data traffic of parallel algorithms.

Library for Efficient Modeling and Optimization in Networks (LEMON) is a C++ template library providing efficient implementations of common data structures and algorithms (`http://lemon.cs.elte.hu`[5]). It supports a flexible text format for storing graphs and related data.

This thesis proposes a new file format called Binary Lemon Graph Format (henceforth BLGF). BLGF is a file format that satisfies the conditions listed above. As its name has already suggested, it stores the data in a binary way, thus it requires less space as text formats, providing most of their functionality.

Keeping existing functionality of LEMON unchanged was one of the main criteria, so the format in which the data will be stored can be chosen, and when reading data, the reader automatically recognizes the format.

The fundamental types (e.g integers, string, reals) are converted automatically, thus there is no need for implementation of value converters, however, we can also define user-specific data types and implement converters for them.

The proposed file format was submitted to several tests. Results of these tests are discussed in the final part of the thesis.

Assuming that the reader has a basic knowledge of graph theory, terms like graph, node, arc, edge, etc. are not explained in this thesis.

# 2 BLGF (Binary Lemon Graph Format) specification

This section presents a short description of the BLGF with a Backus−Naur Form (henceforth BNF) style grammar included. It is a binary file format for storing graphs and associated data (e.g. edge maps, node maps, etc) supporting all the fundamental types (Subsection 2.7.1), as well as user-defined custom types, but compared to the existing Lemon Graph Format (henceforth LGF), there are also some limitations (Subsection 3.2.2).

## 2.1 File Structure

### 2.1.1 Informal specification

BLGF$< version >$
$< block_1\_type >< block_1 > 0$
...
$< block_k\_type >< block_k > 0$
EOF

### 2.1.2 BNF Grammar

$$
\begin{array}{lll}
< blgf\_file > & ::= & < signature >< blocks > \quad EOF \\
< signature > & ::= & \texttt{BLGF}< version > \\
< version > & ::= & < digit >< digit > \\
< blocks > & ::= & \epsilon | < block >< blocks > \\
< block > & ::= & < nodes\_block > \quad | \quad < arcs\_block > \quad | \\
 & & < attributes\_block > \\
< nodes\_block > & ::= & NODES < block\_header >< nodes\_data > NULL \\
< arcs\_block > & ::= & ARCS < block\_header >< arcs\_data > NULL \\
< attributes\_block > & ::= & ATTRIBUTES < attributes\_data > NULL \\
< block\_header > & ::= & < name >< size > \\
NULL & ::= & 0 \text{ stored as UINT8 (i.e. a zero value byte)} \\
EOF & ::= & \text{End Of File}
\end{array}
$$

$NODES$ ::= UINT16 value defined in source

$ARCS$ ::= UINT16 value defined in source

$ATTRIBUTES$ ::= UINT16 value defined in source

Each BLGF file starts with a signature, which are the "BLGF" letters followed by the two digit version number. After that, the file consists of several blocks. Each block is preceded by its type, which is a 2 byte unsigned integer value (UINT16) and it is closed with a zero byte. Currently, there are three types of blocks:

1. Nodes block
2. Arcs block
3. Attributes block

Note: There may be several Nodes and Arcs blocks but only one Attributes block.

## 2.2 Maps

Maps are widely used data structures in LEMON. They allow assigning values of any type to the nodes or arcs of a graph. In BLGF, the maps are represented as sequences of items with a given size and type.

### 2.2.1 BNF Grammar

$< maps >$ ::= $\epsilon | < map >< maps >$

$< map >$ ::= $< map\_header >< map\_data >$

$< map\_header >$ ::= $< name >< type >$

$< map\_data >$ ::= $\epsilon | < map\_item >< map\_data >$

Each map starts with a header followed by items. The header contains the map name and the type of a sequence of items in it. The type also defines the size of the data, and it is again stored as a 2 byte unsigned integer (UINT16).

## 2.3 Nodes block

### 2.3.1 Informal specification

$< block\_name >< block\_size >$

$0 < type >< label_1 >< label_2 >< label_3 > ... < label_{size} >$

$< map_1\_name >< type_1 >< item_{1,1} >< item_{1,2} >< item_{1,3} > ... < item_{1,size} >$

...

$< map_j\_name >< type_j >< item_{j,1} >< item_{j,2} >< item_{j,3} > ... < item_{j,size} >$

0

### 2.3.2 BNF Grammar

$< nodes\_block >$ ::= $NODES < block\_header >< nodes\_data > NULL$

$< block\_header >$ ::= $< name >< size >$

$< nodes\_data >$ ::= $< labels >< maps >$

$< labels >$ ::= $< map >$

This type of block describes a set of nodes and associated maps. It consists of two parts, the header and the data terminated by a null byte. The header contains the name of the block and the number of nodes described (i.e. the item count in each map) stored as a 4 byte unsigned integer (UINT32). The data part contains the maps (Subsection 2.2) stored one after the other. The first map in the Nodes block is a special one, called `Label`, which contains unique id (label) for each node. When describing end points of an arc, these ids are used to refer to the target and the source node. `Label` map has an empty name (i.e. only NULL byte) and it must not be absent! The order of nodes in each map must be the same (e.g. the $i^{th}$ item in each map describes the node labeled by the $i^{th}$ item in the `Label` map)!

## 2.4 Arcs block

### 2.4.1 Informal specification

$< block\_name >< block\_size >$

$0 < type >< from_1 >< to_1 > ... < from_{size} >< to_{size} >$

$< map_1\_name >< type_1 >< item_{1,1} >< item_{1,2} >< item_{1,3} > ... < item_{1,size} >$

...

$< map_i\_name >< type_i >< item_{i,1} >< item_{i,2} >< item_{i,3} > ... < item_{i,size} >$

0

### 2.4.2 BNF Grammar

$< arcs\_block >$ ::= $ARCS < block\_header >< arcs\_data > NULL$

$< block\_header >$ ::= $< name >< size >$

$< arcs\_data >$ ::= $< from\_to >< maps > \mid < from\_to > < labels > < maps >$

$< labels >$ ::= $< map >$

$< from\_to >$ ::= $< map >$

This type of block is very similar to the Nodes block. It again starts with a header containing the name of the block and the number of items described (i.e. the number of arcs or edges). The maps have the same structure as it was described in subsection 2.2. The difference between these two types of blocks is the following: the `Label` map is optional here and the first map in this block must be the `From-To` map, which contains node pairs describing the source and the destination for each arc. The `From-To` map has also an empty name. If there is a `Label` map in Arcs block, it must be on the second place right after the `From-To` map (for more information see subsection 4).

As in the Nodes block, the order of the arcs in the maps must be the same!

## 2.5 Attributes block

### 2.5.1 Informal specification

| |
|---|
| $< key_1 >< value_1 >$ |
| $< key_2 >< value_2 >$ |
| $< key_3 >< value_3 >$ |
| ... |
| $< key_l >< value_l >$ |
| 0 |

### 2.5.2 BNF Grammar

$$
\begin{aligned}
&< attributes\_block > && ::= && ATTRIBUTES < attributes\_data > NULL \\
&< attributes\_data > && ::= && \epsilon | < attr\_key > < attr\_value > < attributes\_data > \\
&< attr\_value > && ::= && < string > \\
&< attr\_key > && ::= && < string >
\end{aligned}
$$

As it was mentioned before, only one Attributes block can take place in the BLGF file. This block contains a null-terminated sequence of key-value pairs. The Attributes block contains no header, since the name is not necessary, and the type of the values is fixed (STRING).

## 2.6 Summary of BNF grammar rules

This subsection contains listing of all the grammar rules used for description of BLGF. These rules does not describe all the requirements for the format. For example, the length limitations of maps could not be described by BNF. Other additional conditions, such as the type consistency of the map items could be expressed using BNF, but it will be too long and complicated.

$$
\begin{aligned}
&< blgf\_file > && ::= && < signature >< blocks > \quad EOF \\
&< signature > && ::= && \texttt{BLGF} < version > \\
&< version > && ::= && < digit >< digit > \\
&< blocks > && ::= && \epsilon | < block >< blocks >
\end{aligned}
$$

| | | |
|---|---|---|
| $< block >$ | ::= | $< nodes\_block >$ \| $< arcs\_block >$ \| $< attributes\_block >$ |
| $< nodes\_block >$ | ::= | $NODES < block\_header >< nodes\_data > NULL$ |
| $< arcs\_block >$ | ::= | $ARCS < block\_header >< arcs\_data > NULL$ |
| $< attributes\_block >$ | ::= | $ATTRIBUTES < attributes\_data > NULL$ |
| $< block\_header >$ | ::= | $< name >< size >$ |
| $< nodes\_data >$ | ::= | $< labels >< maps >$ |
| $< labels >$ | ::= | $< map >$ |
| $< arcs\_data >$ | ::= | $< from\_to >< maps >$ \| $< from\_to >< labels >< maps >$ |
| $< from\_to >$ | ::= | $< map >$ |
| $< attributes\_data >$ | ::= | $\epsilon$\| $< attr\_key >$ NULL $< attr\_value >$ NULL $< attributes\_data >$ |
| $< name >$ | ::= | $< string >$ |
| $< maps >$ | ::= | $\epsilon$\| $< map >< maps >$ |
| $< map >$ | ::= | $< map\_header >< map\_data >$ |
| $< map\_header >$ | ::= | $< name >< type >$ |
| $< map\_data >$ | ::= | $\epsilon$\| $< map\_item >< map\_data >$ |
| $< map\_item >$ | ::= | Sequence of bytes. Length depends on type. |
| $< string >$ | ::= | $< chars >< NULL >$ |
| $< chars >$ | ::= | $\epsilon$\| $< char >< chars >$ |
| $< char >$ | ::= | $[A - Za - z0 - 9\_]$ |
| $< digit >$ | ::= | $[0 - 9]$ |
| $< type >$ | ::= | INT8 \| INT16 \| INT32 \| INT64 \| UINT8 \| UINT16 \| UINT32 \| UINT64 \| FLOAT \| DOUBLE \| STRING \| DATA |
| $NULL$ | ::= | 0 stored as UINT8 (i.e. a zero value byte) |
| $EOF$ | ::= | End Of File |
| $NODES$ | ::= | UINT16 value defined in source |
| $ARCS$ | ::= | UINT16 value defined in source |
| $ATTRIBUTES$ | ::= | UINT16 value defined in source |

## 2.7 Data types

BLGF supports all the common primitive types, the null–terminated STRING and a special DATA type which allows the definition of custom data types.

### 2.7.1 List of types

| Type | Size (bytes) | Short description | Range |
|---|---|---|---|
| INT8 | 1 | signed char | $-2^7$ to $2^7 - 1$ |
| UINT8 | 1 | unsigned char | 0 to $2^8 - 1$ |
| INT16 | 2 | signed short integer | $-2^{15}$ to $2^{15} - 1$ |
| UINT16 | 2 | unsigned short integer | 0 to $2^{16} - 1$ |
| INT32 | 4 | signed integer | $-2^{31}$ to $2^{31} - 1$ |
| UINT32 | 4 | unsigned integer | 0 to $2^{32} - 1$ |
| INT64 | 8 | signed 64bit integer | $-2^{63}$ to $2^{63} - 1$ |
| UINT64 | 8 | unsigned 64bit integer | 0 to $2^{64} - 1$ |
| FLOAT | 4 | floating point data type | |
| DOUBLE | 8 | double-precision float | |
| STRING | Variable | Null terminated string | |
| DATA | Variable | Any kind of data with a size given in first the 4 bytes | |

# 3 User documentation

This section contains all the information about reading and writing data using BLGF. It discusses the most common case, when default value converters are used, as well as the specialized user defined value converters. For better comprehension, this section also contains sample codes. As the LGF (Lemon Graph Format) and the other parts of LEMON are not a part of the present thesis, mostly the BLGF related functionalities of the classes listed below are discussed.

## 3.1 Arrangements

### 3.1.1 LEMON

LEMON stands for the Library for Efficient Modeling and Optimization in Networks. It is a C++ template library providing efficient implementations of common data structures and algorithms with focus on combinatorial optimization tasks connected mainly with graphs and networks. For more information about LEMON and system requirement and installation, see
`http://lemon.cs.elte.hu/trac/lemon/wiki/Documentation`[5].

## 3.2 Usage of BLGF

LEMON provides several implementations of graphs. There are two kinds of graphs in LEMON, directed and undirected graphs. All of them can be easily handled through Graph and Digraph interfaces, where Graph is used for undirected graphs and Digraph for the directed ones.

Both interfaces contain useful methods for treating nodes, directed arcs and undirected edges. Since every edge can be regarded as two oppositely directed arcs, Graph also fulfills the concept of Digraph. Using maps (key-value pairs), different kinds of data can be assigned to nodes or arcs/edges. All these data can be written to an output stream and also read from input stream in LGF or BLGF, using the following classes:

- DigraphWriter: Writes a directed graph to an output stream.
- DigraphReader: Reads a directed graph from an input stream.

- GraphWriter: Writes an undirected graph to an output stream.
- GrapReader: Reads an undirected graph from an input stream.

Instances of these classes can be created by specifying the graph, and the input or output stream. If the data is written to a file or read from it, then instead of the stream, the name of the file can also be given.

To provide easier creation, there are factory functions implemented. They take the same parameters as the constructors, but thanks to the argument deduction, template parameters can be omitted.

**Listing 1:** Declaration of the factory functions

```
template<typename TDGR>
DigraphWriter<TDGR> digraphWriter (const TDGR &digraph, std::ostream &
    os)
template<typename TDGR>
DigraphWriter<TDGR> digraphWriter (const TDGR &digraph, const std::
    string &fn)
template<typename TDGR>
DigraphWriter<TDGR> digraphWriter (const TDGR &digraph, const char *fn)
template<typename TDGR>
DigraphReader<TDGR> digraphReader (TDGR &digraph, std::istream &is)
template<typename TDGR>
DigraphReader<TDGR> digraphReader (TDGR &digraph, const std::string &fn
    )
template<typename TDGR>
DigraphReader<TDGR> digraphReader (TDGR &digraph, const char *fn)

template<typename TGR>
GraphWriter<TGR> graphWriter (const TGR &graph, std::ostream &os)
template<typename TGR>
GraphWriter<TGR> graphWriter (const TGR &graph, const std::string &fn)
template<typename TGR>
GraphWriter<TGR> graphWriter (const TGR &graph, const char *fn)
template<typename TGR>
GraphReader<TGR> graphReader (TGR &graph, std::istream &is)
template<typename TGR>
GraphReader<TGR> graphReader (TGR &graph, const std::string &fn)
template<typename TGR>
```

**Reading and writing rules:** Various reading rules can be added to readers, and analogously, writing rules can be added to writers. These classes do a batch processing. The instance is created, then several reading rules can be added, and eventually the reading or writing process is executed with the `run()` method. Rules can be divided to three categories:

- Rules for reading or writing node maps or arc/edge maps
- Rules for reading or writing certain nodes or arcs/edges
- Rules for reading or writing attributes

**Rules for maps:** The rule for reading or writing a node, arc or edge map, can be added using the `nodeMap()`, `arcMap()` or `edgeMap()` methods. The caption (the name) of the map in the file must be specified as well as the map itself. If there is a need for specialized value converters, they can also be passed by as parameters. If no converters are defined, the default ones are used.

**Listing 2:** Declaration of the nodeMap(), arcMap() and the edgeMap() methods

```
template<typename Map, typename Converter, typename BinaryConverter >
DigraphWriter& nodeMap (const std::string &caption, const Map &map,
    const Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
DigraphWriter& arcMap (const std::string &caption, const Map &map,
    const Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
DigraphReader& nodeMap (const std::string &caption, const Map &map,
    const Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
DigraphReader& arcMap (const std::string &caption, const Map &map,
    const Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
```

```
template<typename Map, typename Converter, typename BinaryConverter >
GraphWriter& nodeMap (const std::string &caption, const Map &map, const
    Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
GraphWriter& edgeMap (const std::string &caption, const Map &map, const
    Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
GraphWriter& arcMap (const std::string &caption, const Map &map, const
    Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
GraphReader& nodeMap (const std::string &caption, const Map &map, const
    Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
GraphReader& edgeMap (const std::string &caption, const Map &map, const
    Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
template<typename Map, typename Converter, typename BinaryConverter >
GraphReader& arcMap (const std::string &caption, const Map &map, const
    Converter &converter=Converter(), const BinaryConverter &
    binaryConverter=BinaryConverter())
```

**Rules for nodes and arcs/edges:** Sometimes we have to store a special node or arc/edge (e.g. destination node of a shortest path problem). The `node()`, `arc()` and `edge()` methods should be used for adding rules for these data.

**Listing 3:** Declaration of the node(), arc() and the edge() methods

```
DigraphWriter& node (const std::string &caption, Node &node)
DigraphWriter& arc (const std::string &caption, Arc &arc)
DigraphReader& node (const std::string &caption, Node &node)
DigraphReader& arc (const std::string &caption, Arc &arc)

GraphWriter& node (const std::string &caption, Node &node)
GraphWriter& arc (const std::string &caption, Arc &arc)
GraphWriter& edge (const std::string &caption, Edge &edge)
```

```
GraphReader& node (const std::string &caption, Node &node)
GraphReader& arc (const std::string &caption, Arc &arc)
GraphReader& edge (const std::string &caption, Edge &edge)
```

**Rules for reading attributes:** Extra data can be read or written too. For example the name of graph, notes on it, other values associated with the algorithm or solution, etc. These rules are added by the following methods:

**Listing 4:** Declaration of attribute() methods

```
template<typename Value, typename Converter >
GraphWriter& attribute (const std::string &caption, Value &value, const
    Converter &converter=Converter())
template<typename Value, typename Converter >
GraphReader& attribute (const std::string &caption, Value &value, const
    Converter &converter=Converter())

template<typename Value, typename Converter >
DigraphWriter& attribute (const std::string &caption, Value &value,
   const Converter &converter=Converter())
template<typename Value, typename Converter >
DigraphReader& attribute (const std::string &caption, Value &value,
   const Converter &converter=Converter())
```

**Captions:** By default the sections/blocks does not have captions, but they can be given by `nodes()`, `arcs()`/`edges()` and `attributes()` methods.

**Listing 5:** Declaration of the nodes(), arcs() and the edges() methods

```
DigraphWriter& nodes (const std::string &caption)
DigraphWriter& arcs (const std::string &caption)
DigraphWriter& attributes (const std::string &caption)

GraphWriter& nodes (const std::string &caption)
GraphWriter& edges (const std::string &caption)
GraphWriter& attributes (const std::string &caption)
```

**Skipping sections/block:** Reading or writing sections/block can be omitted by calling `skipNodes()`, `skipArcs()` or `skipEdges()`.

**Listing 6:** Declaration of the skipNodes(), skipArcs() and the skipEdges() methods

```
DigraphWriter& skipNodes ()
DigraphWriter& skipArcs ()
DigraphReader& skipNodes ()
DigraphReader& skipArcs ()

GraphWriter& skipNodes ()
GraphWriter& skipEdges ()
GraphReader& skipNodes ()
GraphReader& skipEdges ()
```

**Using previously constructed Node or Arc Set:** Since a block can be skipped, there is an option to use previously constructed Node or Arc sets. Sometimes it is unavoidably, for example, when node block is skipped, and arcs are read. In this case, a previously constructed label node map should be passed to `useNodes` method.

**Listing 7:** Declaration of the useNodes(), useArcs() and the useEdges() methods

```
template<typename Map, typename Converter >
DigraphReader& useNodes (const Map &map, const Converter &converter=
    Converter())
template<typename Map, typename Converter >
DigraphReader& useArcs (const Map &map, const Converter &converter=
    Converter())

template<typename Map, typename Converter >
GraphReader& useNodes (const Map &map, const Converter &converter=
    Converter())
template<typename Map, typename Converter >
GraphReader& useEdges (const Map &map, const Converter &converter=
    Converter())
```

**The execution of the process:** After all the rules and other necessary parameters have been added, the process can be executed with the `run()` method.

**The reading:** Since the reader automatically detects the format, no additional parameters are required by this method.

```
void run()
```

**The writing:** When executing a writing process, two parameters can be passed to the `run()` method. The first determines the format whether it will be written in LGF or BLGF, and the second tells the writer to append a file or not. The last parameter is used only if the BLGF was chosen, and its value has to be set to true when more then one arc/edge block should be written to file. By default, the process writes the stream in LGF to keep backward compatibility of LEMON..

**Listing 8:** Declaration of the run() method

```
void run(bool binary = false, bool append = false)
```

**Example:** The following code writes several maps and attributes to the "test.blgf" file in BLGF.

**Listing 9:** Sample code: adding rules to the writer

```
digraphWriter(digraph, "test.blgf").
  nodeMap("coordinates", coord_map).
  nodeMap("size", size).
  nodeMap("title", title).
  arcMap("capacity", cap_map).
  node("source", src).
  node("target", trg).
  attribute("caption", caption).
  run(true, false);
```

### 3.2.1 User defined value converters:

Default value converters may not work for user defined types. To solve this problem, specialized value converters can be defined and passed to the writer or the reader.

**Text converters:** Basically, they are standard functors used for converting values to std::string or for converting in the opposite way. Implementation of a custom value converter is quite easy. The class used as a converter for reading must have an

```
Value operator()(const std::string& str)
```

operator defined, where `Value` is a type of the read data. In case of writing,

```
std::string operator()(const Value& value)
```

has to be defined. Of course, these operators can be put into a single class.

**Binary converters:** They are used for converting values to or from their binary representations.

Converters used for converting the data from binary are functors that must have an

```
Value operator()(const _blgf_bits::BlgfType blgfType, const char* data)
```

operator defined, where the Value is a C++ type to which the data will be converted. The operator takes two parameters, the type of the binary data, and the data itself.

For conversion to binary, the converters has to be derived from

```
template <typename Value>
class DefaultBinaryConverterBase : public BinaryTokenBase
```

template class, because it provides an interface for easy manipulation with the binary data. As the new class will inherit from `DefaultBinaryConverterBase` and `BinaryTokenBase`, the following methods should to be defined:

- `_blgf_bits::BlgfType getType() const`

  By default, this function returns `_blgf_bits::DATA` value indicating that this type is user defined. It can be overridden, but in most cases it is not recommended!

- `int getSize() const`

  This method should return the size (in bytes) of value converted to binary.

- `int write(char* buffer, int buffer_size) const`

  It has to write the binary data to a specified buffer. The second parameter only defines the size of the buffer, guaranteeing the proper memory handling.

- `void setValue(const Value& value)`

  This method sets a new value for the converter.

- **void** setValue(**const** Value& value, **const** _blgf_bits::BlgfType type) = 0;

  It sets a new value for the converter and specifies the BLGF type in which the data will be written. By default, this function calls `setValue(const Value& value)` and throws exception if the type differs from `_blgf_bits::DATA`. It is not recommended to override this function!

**Advice:** The best way to implement custom a converter is to write a single class that satisfies the requirements for the text as well as for the binary converters.

**Example:** The following code demonstrates how to use specialized value converters.

**Listing 10:** Sample code: Using specialized value converters

```
//User-defined type
class TestClass{
private:
  int a;
  double d;
public:
  TestClass(){
    a = 0;
    d = 0;
  }

  void setInt(int value){
    a = value;
  }
  void setDouble(double value){
    d = value;
  }
  int getInt() const{
    return a;
  }
  double getDouble() const {
    return d;
  }
```

```
};

class BiConverter : public _writer_bits::DefaultBinaryConverterBase<
    TestClass>{
private:
  char* _data;
  int _size;
public:

  BiConverter():_data(0), _size(0){}

  ~BiConverter(){
    delete _data;
  }


  void setValue(const TestClass& value){
    delete _data;
    ostringstream os;

    os << "b(" << value.getInt() << ","<< value.getDouble() <<")";
    _size = os.str().length()+1;
    _data = new char[getSize()];
    _data[getSize()] = 0;

    memcpy(_data, os.str().c_str(), getSize());
  }

  int getSize() const{
    return _size;
  }

  int write(char * buffer, int size) const{
    memcpy(buffer, _data, getSize());
  }


  //Operator for text conversion(write)
```

```
std::string operator()(const TestClass& value) {
      std::ostringstream os;
      os << "(" << value.getInt() << ","<< value.getDouble() <<")";
      return os.str();
}

 //Operator for binary conversion(read)
TestClass operator()(const _blgf_bits::BlgfType blgfType, const char*
    data) {
  //skips the 'b' letter
  string s(data+1);
  TestClass tmp;
  char c;
  stringstream is;
  double d;
  int a;

  is <<s;
  is >>c;
  is >>a;
  is >>c;
  is >>d;

  tmp.setInt(a);
  tmp.setDouble(d);
  return tmp;
}

//Operator for text conversion(read)
TestClass operator()(string s) {
  TestClass tmp;
  char c;
  stringstream is;
  double d;
  int a;

  is <<s;
  is >>c;
  is >>a;
```

```
    is >>c;
    is >>d;

    tmp.setInt(a);
    tmp.setDouble(d);
    return tmp;
  }

};
```

### 3.2.2 Limitations

Compared to LGF, the implementation of BLGF has some limitations:

- Using real types with their default value converters, the BLGF becomes portable only on machines with the same memory endianness (i.e. big-endian or little-endian).
- Long double type is not supported yet.
- The values of the label maps must be fundamental types (integer, real or string).

See also Future development (Subsection 4.6).

# 4 Developer documentation

The developer documentation contains all the details about the implementation of BLGF. There were several design problems and theoretical problems during the implementation. All these problems and their solutions and compromises are included, too. To keep all the functionalities of the existing applications using LGF, the main interface of the classes has not been changed. These classes and their methods have been extended and overloaded, so their default behavior remained the same.

## 4.1 Common types, constants and functions

### 4.1.1 Types, enums and constants

- **const unsigned char** BLGF_VERSION[]

  C style string containing the signature of the BLGF file followed by its version.

- **const unsigned int** TYPE_GROUP_MASK

  Binary mask for getting the group in which the type belong using the & bitwise operator. For example:

  ```
  return (((type) & TYPE_GROUP_MASK) == TYPE_SIGNED_BASE);
  ```

  returns true if the type is signed integer.

- **const unsigned int** TYPE_UNSIGNED_BASE

  Base number for enumerating unsigned integer types.

- **const unsigned int** TYPE_SIGNED_BASE

  Base number for enumerating signed integer types.

- **const unsigned int** TYPE_REAL_BASE

  Base number for enumerating real types.

- **const unsigned int** TYPE_OTHER_BASE

  Base number for enumerating the rest of the types.

- **enum** BlgfType

Enumeration of BLGF data types. It is divided to 4 groups (unsigned, signed, real and other). Each group's first item gets a value defined by one of the base integer constants described above, increased by one.

- **enum** BlgfBlockType

  Enumeration of BLGF block types.

- **const unsigned int** UINT8_SIZE, INT8_SIZE, UINT16_SIZE, INT16_SIZE, UINT32_SIZE, INT32_SIZE, UINT64_SIZE, INT64_SIZE, FLOAT_SIZE, DOUBLE_SIZE

  Integer constants defining the size (in bytes) of the data types.

### 4.1.2 Functions

All the functions listed below are used for getting properties of BLGF types.

- **inline bool** is_signed(BlgfType type)

  Returns true if the type is signed integer.

- **inline bool** is_unsigned(BlgfType type)

  Returns true if the type is unsigned integer.

- **inline bool** is_integer(BlgfType type)

  Returns true if the type is integer.

- **inline bool** is_real(BlgfType type)

  Returns true if the type is real.

- **inline int** get_type_size(BlgfType type)

  Returns the size (in bytes) of the type.

## 4.2 Value converters

Basically, the value converters are used for converting C++ types to their binary or string representation, or backwards. The BLGF has an implementation of default converters, which can handle all the fundamental C++ data types. The existing implementation of LGF uses operators << and >> and std::stringstream for conversions between std::strings and C++ types. In the case of the BLGF, the conversion is not so trivial.

**Converting integers:** Integer types are written out in Little-endian (i.e. the least significant byte (LSB) is written out first).

**Listing 11:** Pseudo code of integer conversion

```
for (int i = sizeof(BLGF_INTEGER_TYPE); i >= 0; i--){
  bytes[i] = value & 0xff;
  value = value >> 8;
}
```

This type of conversion is independent from the memory's endianness, thus it is a portable solution.

**Converting reals:** Because there is no standard for transferring floating point values, reals are written out exactly as they are in the memory. This solution is not portable, so when using real types, the BLGF becomes portable only on machines with the same architecture.

For more information see subsections 3.2.2 and 4.6

**Converting string:** As the BLGF STRING is a null-terminated sequence of characters, they can be handled as C-style strings. The std::string has a constructor that takes a null-terminated string and it also has a method c_str() for getting the value in C-style, so the conversion is already implemented in the Standard Template Library.

### 4.2.1 Reader converters and related classes

**SuperType:** This class provides converting binary data to C++ types. Since there is not an exact correspondence between BLGF types and C++ types, the

SuperType provides operators for converting binary representation of the data to several types (e.g. INT16 can be converted to int, float, double, string, etc.). It also throws appropriate exceptions when trying to convert to incompatible type.

### Public methods

- `read(const _blgf_bits::BlgfType type, const char* buffer)`

  Reads a given BLGF type from a buffer.

- `template<typename T> operator T()`

  This template operator is used for converting integer types. It converts a BLGF type to an integer C++ type given in the template parameter.

- `operator float()`

  Specialized operator for converting to float.

- `operator double()`

  Specialized operator for converting to double.

- `operator std::string()`

  Specialized operator for converting to std::string.

### Private methods

- `template <typename CppType>`
  `void fromBinary(const char* array, int n, bool is_signed, CppType & value)`

  It converts a binary representation of an integer value to a C++ type given in the template parameter.

**BinaryToken:** As we don't know the size of the data before we read it (e.g. STRING, DATA have a variable size) and using dynamically allocated character arrays (char *) is not a best solution, the `BinaryToken` class has been implemented for encapsulating binary data. It contains useful methods for an easy manipulation as well as a copy constructor, destructor and assignment operator to guarantee the proper allocation and deallocation of dynamic variables.

**Major methods**

- `char const* getData() const`

  Returns the pointer to the binary data.

- `int getSize() const`

  Return the size of the data (in bytes).

- `char* read(std::istream& _is, const _blgf_bits::BlgfType type)`

  Private method that reads specified type of data from an input stream.

**Default converters:** Default converters are used to convert fundamental data types from their binary representation. Conversion is made by `SuperType` (Subsection 4.2.1) These converters can be divided into 4 groups:

- Default integer converters
- Default double converter
- Default float converter
- Default string converter

**Default integer converters:** Integer values can be converted using

```
template <typename Value> struct DefaultBinaryConverter
```

template and its operator

```
Value operator()(const _blgf_bits::BlgfType blgfType, const char* data)
```

where the template parameter `Value` defines the C++ type in which the data will be converted and the `blgfType` specifies the BLGF type of the data stored in `data` parameter.

**Default double, float and string converters:** For converting data to double, float or string, there are specializations of the `DefaultBinaryConverter`. The conversion are made again by the operators:

- `double operator()(const _blgf_bits::BlgfType blgfType, const char* data)`

- `float operator()(const _blgf_bits::BlgfType blgfType, const char* data)`

- `string operator()(const _blgf_bits::BlgfType blgfType, const char* data)`

**readTokenBinary:** This function reads data from input stream to `BinaryToken` type.

```
inline std::istream& readTokenBinary(std::istream& is, BinaryToken&
    token, const _blgf_bits::BlgfType type)
```

**Parameters:**

- `std::istream& is`

  Reference to the input stream.

- `BinaryToken& token`

  Reference to a BinaryToken variable into which the data will be read.

- `_blgf_bits::BlgfType type`

  Type of the data to read.

### 4.2.2 Writer converters and related classes

**BinaryTokenBase:** This class is an abstract one, defining the interface of all derived classes for easy manipulation with the binary data.

**Public methods:**

- `virtual _blgf_bits::BlgfType getType() const`

  Method for getting the BLGF type of the binary data encapsulated by this class.

- `virtual int getSize() const = 0`

  Overridden versions of this method have to return the size (in bytes) of the binary data.

- `virtual int write(char* buffer, int buffer_size) const = 0`

  This method should write the data to a specified buffer. It takes 2 parameters, a pointer to the buffer and its size, to avoid segmentation error. The return value is a number of bytes written in the buffer. It should be the same value as the one returned by `getSize()`.

**DefaultBinaryConverterBase:** To define any type of converter for converting to binary, it has to be derived from this class. DefaultBinaryConverterBase defines an interface for value conversions and the class itself is derived from the `BinaryTokenBase` expanded with 2 virtual methods:

- `virtual void setValue(const Value& value, const _blgf_bits::BlgfType type) = 0`

  Pure virtual method for setting the value of the converter with the given BLGF type. It allows storing the data as a different type (e.g. store int as DOUBLE or short as INT64, etc.).

- `virtual void setValue(const Value& value) = 0`

  Also a pure virtual method for setting the value, but in this case the BLGF type is not given, it is determined automatically. In most cases the type is obvious, but if there are more options, the smallest compatible type is chosen (e.g. Assuming that short is 3 bytes long, INT32 is chosen for a BLGF type, because it is signed and stored in 4 bytes so there will be no numeric overflow).

**Default converters** As in the case of reading data, implementation of BLGF provides default converters for converting fundamental types to their binary representation. They can be divided to four groups again:

- Default integer converters
- Default double converter

- Default float converter
- Default string converter

**Default integer converters:** Integer values are converted to binary using

```
template <typename Value> class DefaultBinaryConverter: public
    DefaultBinaryConverterBase<Value>
```

template class. As it is derived from `DefaultBinaryConverterBase`, it has implemented all the functions required by `BinaryTokenBase` and `DefaultBinaryConverterBase`. Function

```
virtual int write(char* buffer, const int buffer_size) const
```

writes the binary data to a buffer. For integer values it uses

```
template <typename CppType>
void toBinary(CppType value, char* array, int n) {
```

template function, which by default throws an error because it can be used only for integer types. Using the following macro it is specialized for all the integer types. Example:

```
#define MK_INTEGER_TO_BINARY(CppType)\
template <> void toBinary<CppType>(CppType value, char* array, int n)
    {\
  for (int i = n - 1; i >= 0; i--) {\
    array[i] = static_cast<char> (value & static_cast<CppType> (0xff))
        ;\
    value = value >> static_cast<CppType>(8);\
  }\
};

MK_INTEGER_TO_BINARY(char)
MK_INTEGER_TO_BINARY(signed char)
MK_INTEGER_TO_BINARY(unsigned char)
...
```

**Default double, float and string converters:** The non-integer fundamental types have their specialized `DefaultBinaryConverter` template classes. Since they

have the same base classes, the public interface remains the same. The main difference is that for serializing the data they do not use `toBinary()` function because there is no & operator defined for real types nor for the string. Strings are written sequentially character by character and terminated by a zero byte. Real types are written in the same order and size as they are in memory (Subsections 3.2.2 and 4.2).

## 4.3 Storage classes

The classes described below are already widely used in the implementation of the LGF, so this section provides only a short description of their functionality and changes that have been made during the implementation of the BLGF.

### 4.3.1 Storage classes used for reading

**_reader_bits::MapStorageBase:** Base class defining the common interface for map storage classes. This class and all the classes derived from it has been expanded with a `setBinary()` method in order to handle binary conversions as well as the text conversions.

**Major methods:**

- `virtual void set(const Item& item, const std::string& value)`
- `virtual void setBinary(const Item& item, const char* data, const _blgf_bits::BlgfType blgfType)`

**_reader_bits::MapStorage:** Stores a map, and allows an easy setting of the values, including conversion from text or binary. If there is a need for specialized converters, they can be passed by as template parameters.

**Major methods:**

- `virtual void set(const Item& item, const std::string& value)`

  This method converts a string value taken as a parameter using the text converter, and sets the the new value for the item in the map.

- `virtual void setBinary(const Item& item, const char* data, const _blgf_bits::BlgfType blgfType)`

  `setBinary()` converts a binary data with a given type taken as a parameter using the binary converter, and sets the the new value for the item in the map.

**_reader_bits::GraphArcMapStorage:** It is very similar to the `MapStorage` class implemented for storing arc maps. The main benefit of this class is, that it can handle storage of arc maps when reading undirected graph. Each edge added is directed and stored as arc. The direction depends on template parameters of this class.

**_reader_bits::ValueStorageBase:** This class is just a base class for all the value storage classes. The implementation of BLGF uses classes derived from this one during the reading of the attributes block, and as its items are stored as strings and std::string and C-style string conversion is defined (Subsection 4.2), there is no need for a setBinary() method or a binary converter template parameter.

**Major methods:**

- `virtual void set(const std::string&)`

**_reader_bits::ValueStorage:** Stores an item and allows setting a new value for it.

**Major methods:**

- `virtual void set(const std::string&)`

  Converts the string using a converter and stores the new value.

### 4.3.2 Storage classes used for writing

**_writer_bits::MapStorageBase:** Base class defining the common interface for map storage classes. A new `getBinary(...)` method has been added to this class for an easy get of the binary representations of the values in the map.

**Methods:**

- `virtual std::string get(const Item& item)`

- `virtual BinaryTokenBase const* getBinary(const Item& item) = 0;`

**_writer_bits::MapStorage:** Stores a map and has useful methods for getting string or binary representations of the values in it. Specialized converters can be passed by as template parameters.

**Methods:**

- `virtual std::string get(const Item& item)`

  This method gets the value associated with the item from the map, converted to std::string.

- `virtual BinaryTokenBase const* getBinary(const Item& item) = 0;`

  This method gets a binary representation of the value associated with the item.

**_writer_bits::GraphArcMapStorage:** Similar to the `MapStorage` class. It is used for writing out arc maps associated to an undirected graph. The class stores an arc map and a direction (forward or backward) and allows getting a binary representation of a value associated to one of the edge's directions (i.e. arc).

**_writer_bits::ValueStorageBase:** Base class for all value storage classes.

**Major methods:**

- `virtual std::string get(const Item& item)`

**_writer_bits::ValueStorage:** Stores an item and allows getting its value converted to std::string.

**Major methods:**

- `virtual std::string get(const Item& item)`

  This method gets the string representation of the item.

## 4.4 Reading and writing directed graphs

LEMON uses the `DigraphReader` and `DigraphWriter` classes for reading and writing directed graphs. Both classes can handle LGF as well as BLGF. Most functionality of them had already been implemented, but there has been several changes made, in order to handle the BLGF.

### 4.4.1 DigraphReader

Class providing reading directed graphs from an input stream. For information about using this class see section 3 or `http://lemon.cs.elte.hu` [5].

**Changes made to DigraphReader**

- Changed `run()` method
- New overloaded `arcMap(...)` method
- New overloaded `nodeMap(...)` method
- New private `skipMap(...)` method
- New private `skipBlock(...)` method
- New private `readNodesBinary()` method
- New private `readArcsBinary()` method
- New private `readAttributesBinary()` method

**run():** This method starts the batch processing, but before the processing of the data starts, the type of the format is detected. Since every BLGF file starts with its signature, the type detection is unambiguous.
The original method has been divided to two methods according to format:

- **void** runBinary()

  This method reads binary data. Reads all the blocks from the stream. If there is no rule to read the block, it skips to the next one.

- **void** runText()

  A method that reads data in LGF. The implementation of reading remained unchanged.

**arcMap() and nodeMap():** New overloaded versions of `arcMap` and `nodeMap` methods have been added to DigraphReader with a new optional template parameter, a specialized binary value converter.

```
template <typename Map, typename Converter, typename BinaryConverter>
DigraphReader& arcMap(const std::string& caption, Map& map,
  const Converter& converter = Converter(),
  const BinaryConverter& binaryConverter = BinaryConverter())

template <typename Map, typename Converter, typename BinaryConverter>
DigraphReader& nodeMap(const std::string& caption, Map& map,
  const Converter& converter = Converter(),
  const BinaryConverter& binaryConverter = BinaryConverter())
```

**skipMap():** Skips a whole map in the input stream and places the pointer right after that. Because the count of the item in maps is specified in the header of the block, it must be passed through as input parameter (`itemCount`). There is one more optional parameter, `readCaption` telling to the method whether the caption of the map has been already read or not.

```
void skipMap(int itemCount, bool readCaption = true)
```

**skipBlock():** Skips the current block and all the maps in it. As the structures of blocks may differ(e.g. Attributes block dos not store maps), it takes the type of the block as a parameter.

```
void skipBlock(_blgf_bits::BlgfBlockType type)
```

**readNodesBinary():** Read maps from a nodes block. First of all, it reads the size of the block(i.e. count of items in each map), then it reads the `Label` map and stores the labels. The labels are stored as std::strings using the default converter to guarantee LGF and BLGF compatibility. After that, the maps are read one after the other, and their items are stored in one of the map storage classes which also provides conversion of the values by calling the `setBinary(...)` method. If there is no rule for reading a map, it is skipped.

```
void readNodesBinary()
```

**readArcsBinary():** Its working is pretty similar to the `readNodesBinary` described above. The difference is, that, after reading the size of the block it reads the `From-To` map, that contains the source and the destination nodes for each arc. If there is a `Label` map, it has to be placed right after the `From-To` map, and then the rest of the maps are read.

```
void readArcsBinary()
```

**readAttributesBinary():** The attributes block has no size specified and it stores only value-key pairs, not maps, so the functioning of this method is quite different from the previous two. It reads a key and if there is a rule for it, stores its converted value in a `ValueStorage`.

```
void readAttributesBinary()
```

### 4.4.2 DigraphWriter

The DigraphWriter class providing the writing of directed graphs to an input stream. For information about using this class see section 3 or `http://lemon.cs.elte.hu` [5].

#### Changes made to DigraphWriter

- Changed `run()` method
- New overloaded `arcMap(...)` method
- New overloaded `nodeMap(...)` method

- New private `writeNodesBinary()` method
- New private `writeArcsBinary()` method
- New private `writeAttributesBinary()` method

**run():** This method starts the batch processing. The signature of this method has been changed.

```
void run(bool binary = false, bool append = false))
```

New parameters have been added to determine the file format (`binary`), and to tell the writer whether the binary file should be appended or not (`append`), i.e. no signature is written in case of appending. These parameters are optional, and the default value of both parameters is false, so all the existing functionality of the method remains the same.

**arcMap() and nodeMap():** Similarly to the `DigraphReader`, methods `arcMap` and `nodeMap` for adding writing rules have been overloaded, with a new optional template parameter added, a specialized binary value converter.

```
template <typename Map, typename Converter, typename BinaryConverter>
DigraphWriter& nodeMap(const std::string& caption, const Map& map,
  const Converter& converter = Converter(),
  const BinaryConverter& binaryConverter = BinaryConverter())

template <typename Map, typename Converter, typename BinaryConverter>
DigraphWriter& arcMap(const std::string& caption, const Map& map,
  const Converter& converter = Converter(),
  const BinaryConverter& binaryConverter = BinaryConverter())
```

**writeNodesBinary():** Writes a `nodes` block to the stream and guarantees that the `label` map is written in the first place.

```
void writeNodesBinary()
```

**Steps of the algorithm:**

- writes out the header of the block

- writes out the `label` map
- writes out the rest of the maps added by `nodeMap(...)` function

**writeArcsBinary():** Writes an `arcs` block to the stream. Guarantees the right order of maps, i.e. the `From-To` map is written out first and if there is a `label` map, it is written out second.

```
void writeArcsBinary()
```

**Steps of the algorithm:**

- writes out the header of the block
- writes out the `From-To` map
- if there is a a `label` map added to the writer, writes it out
- writes out the rest of the maps added by `nodeMap(...)` function

**writeAttributesBinary():** Writes an `attributes` block to the stream containing all the keys and values added by `arc(...)`, `attribute(...)` and `node(...)` methods.

```
void writeAttributesBinary()
```

## 4.5 Reading and writing undirected graphs

The classes and algorithms in this section are similar to those described in the previous subsection, so only the important things and the differences between them are listed here.

### 4.5.1 GraphReader and GraphWriter

These classes have almost the same functionality as the `DigraphReader` and `DigraphWriter`. The difference is that they work with undirected graphs and they can handle edges and edge maps as well as arcs and arc maps. For more information see `http://lemon.cs.elte.hu/`

**Changes made to GraphReader**

- Changed `run()` method
- New overloaded `arcMap(...)` method
- New overloaded `edgeMap(...)` method
- New overloaded `nodeMap(...)` method
- New private `skipMap(...)` method
- New private `skipBlock(...)` method
- New private `readNodesBinary()` method
- New private `readEdgesBinary()` method
- New private `readAttributesBinary()` method

**Changes made to GraphWriter**

- Changed `run()` method
- New overloaded `arcMap(...)` method
- New overloaded `edgeMap(...)` method
- New overloaded `nodeMap(...)` method
- New private `writeNodesBinary()` method
- New private `writeArcsBinary()` method
- New private `writeAttributesBinary()` method

All the changes are analogue to the changes that have been made to `DigraphReader` and `DigraphWriter` (Subsection 4.4.1 ). The `skipMap(...)` and `skipBlock(...)` methods are exactly the same as in the `DigraphReader`.

## 4.6 Further development

During the implementation of the BLGF, a lot of emphasis was placed on possibility of further development. The fact, that the most of the classes are templates, and they are handled through common interfaces, makes the development quite simple. New default value converters as well as new built in BLGF types can be easily added to the implementation.

**Some concrete ideas for further development**

- The standardization of real types to increase the portability of BLGF. It was not realized, because currently, there is no standard for transferring floating point values.
- Implementation of long double data type. Since some compilers may use long double type which does not conform to IEEE floating-point standard, this feature has not been implemented yet.
- Implementation of a utility, that can automatically convert LGF to BLGF and backwards.
- As the storage of different types of data coupled together is sometimes very useful, the implementation of a `pair` type for BLGF, that will be able to store `std::pair` type could be a good improvement.

## 4.7 Result of the testing

The correctness of the algorithms, and throwing of the appropriate exceptions was one of the main criteria, therefore, several tests have been made. This section deals with the description of all the tests and their results.

### 4.7.1 Default value converters

Source and test files: conv_test.cpp

- Default integer converters: All integer values were converted correctly. Correct exceptions were thrown when numeric overflow happened or when tried to convert negative value to unsigned type.

- Default real converters: Real types was converted correctly. The conversion between float and double worked and appropriate exception was thrown when there was a numeric overflow.

- Default string converter: Conversion between C-style strings and std::string worked correctly as well as the conversion of fundamental types to std::string.

### 4.7.2 The proper functionality of the reader and writer classes

Source and test files: funct_test.cpp, graph_funct_test.h, digraph_funct_test.h, customs.h, conv_test.cpp, digraph.lgf, graph.lgf

- DigraphReader and DigraphWriter: Both classes made correct results when used in binary way. The usage of previously constructed node and arc sets was tested too, as well as skipping one of the node or arc blocks. Writing more then one arcs block to one file worked correctly, too.

- GraphReader and GraphWriter: The testing was same as the previous one, with the only difference that the correct handling of both arcs and edges was tested, too. All the results were the expected ones.

- Custom converters: The custom value converters behaved correctly.

### 4.7.3 Size and performance

Source and test files: size_performance.cpp

- Size: In some extraordinary cases, the BLGF may produce bigger files as the LGF(e.g. storing only one digit numbers). In the worst scenario, when all the values were big and the real values had a long fraction parts, the BLGF produced a file whose size was about the half of the size of the file written in LGF. As the LGF is a very compact file format, it requires much less space then formats based on XML. The size of a file produced by BLGF is incomparably smaller than a file produced by one of the XML based file formats.

- Performance: The BLGF files were written about 4 times faster than the LGF files. The reading of BLGF file was not much faster than reading of LGF. It took about 25% less time.

# 5 Summary

The development of a new binary file format for LEMON was successful. All the tests had proved that BLGF satisfies all the conditions for graph storing file formats and it provides most of the functionality of the existing LGF (Lemon Graph Format) file format. The implementation of BLGF was successfully integrated to LEMON, keeping its former functionality unchanged.

**Main benefits of BLGF:**

- Requires much less space then other file formats, especially the XML based ones.
- Compared to other formats, reading and writing operations take significantly less time.
- Common primitive types are converted automatically.
- When using user defined types, there is an option for defining specialized value converters.
- The interface of the reading and writing classes remained almost the same, so there is no need for reimplementation of applications that use older versions of LEMON.

**Limitations of BLGF:** As the BLGF stores data in binary way, it is less flexible than LGF, so it has some limitations compared to LGF. Most of them are already in development.

- There can be portability problems using real types with BLGF.
- Some widely used, but non-standardized types are not implemented yet (e.g. long double).
- Using BLGF, nodes and arcs can be labeled only by primitive type values.

# References


[1] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2001. ISBN 0-201-88954-4.

[2] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: Complete Guide*. Addison-Wesley, 2002. ISBN 0-201-73484-2.

[3] Dimitri van Heesch. Doxygen. `http://www.stack.nl/~dimitri/doxygen/`, 2010-06-02.

[4] Installation Guide. `http://lemon.cs.elte.hu/trac/lemon/wiki/InstallGuide`, 2010-06-02.

[5] Library for Efficient Modeling and Optimization in Networks (LEMON). `http://lemon.cs.elte.hu`, 2010-06-02.

[6] The GraphML File Format. `http://graphml.graphdrawing.org/`, 2010-05-03.

[7] GXL – Graph eXchange Language. `http://www.gupro.de/GXL/`, 2010-05-06.

[8] Standard for binary floating-point arithmetic (IEEE 754-1985) description, 1985.

[9] The C++ Resources Network. `http://www.cplusplus.com/`, 2010-06-02.

[10] C++ reference. `http://www.cppreference.com/wiki/`, 2010-06-02.

# Declaration of Authorship

I hereby confirm that I have authored this Bachelor's thesis independently and without the use of others than the indicated sources. All passages which are literally or in general matter taken out of publications or other sources are marked as such.

Budapest, June 10, 2010

Frantisek Csajka