LEMON – an Open Source C++
Graph Template Library

Balázs Dezső¹, Alpár Jüttner², Péter Kovács¹

¹Department of Algorithms and Their Applications
Eötvös Loránd University, Budapest, Hungary
{deba, kpeter}@inf.elte.hu

²Department of Operations Research
Eötvös Loránd University, Budapest, Hungary
alpar@cs.elte.hu

WGT 2010 – Workshop on Generative Technologies
Paphos, Cyprus, March 27, 2010
Overview

1 Introduction to LEMON
   - What is LEMON?
   - Graph Structures
   - Iterators
   - Handling Graph Related Data
   - Algorithms
   - Graph Adaptors
   - LP Interface
   - Technical Support

2 Implementation Details
   - Extending Graph Interfaces Using Mixins
   - Signaling Graph Alterations
   - Tags and Specializations

3 Performance

4 History and Statistics

5 Conclusions
1. Introduction to LEMON
Introduction to LEMON

What is LEMON?

- **LEMON** is an abbreviation for **Library for Efficient Modeling and Optimization in Networks**.
- It is an open source C++ template library for optimization tasks related to graphs and networks.
- It provides highly efficient implementations of common data structures and algorithms.
- It is maintained by the EGRES group at Eötvös Loránd University, Budapest, Hungary.
- [http://lemon.cs.elte.hu](http://lemon.cs.elte.hu)
What is this talk about?

- The basic **design concepts** and **features** of LEMON are presented.
- Selected **implementation details** are also presented demonstrating the use of C++ templates and other techniques.
- The **performance** of the library is compared to **BGL** (Boost Graph Library) and **LEDA**, the two major competitors of LEMON.
- BGL is open source, LEDA is a commercial library.
Design Goals

- **Genericity:**
  - clear separation of data structures and algorithms.
Design Goals

- **Genericity:**
  - clear separation of data structures and algorithms.

- **Running time efficiency:**
  - to be appropriate for using in running time critical applications.
Design Goals

- **Genericity:**
  - clear separation of data structures and algorithms.

- **Running time efficiency:**
  - to be appropriate for using in running time critical applications.

- **Ease of use:**
  - elegant and convenient interface based on clear design concepts,
  - provide a large set of flexible components,
  - make it easy to implement new algorithms and tools,
  - support easy integration into existing applications.
Design Goals

- **Genericity:**
  - clear separation of data structures and algorithms.

- **Running time efficiency:**
  - to be appropriate for using in running time critical applications.

- **Ease of use:**
  - elegant and convenient interface based on clear design concepts,
  - provide a large set of flexible components,
  - make it easy to implement new algorithms and tools,
  - support easy integration into existing applications.

- **Applicability for production use:**
  - open source code with a very permissive licensing scheme (Boost 1.0 license).
Copyright (C) 2003-2010 Egerváry Jenő Kombinatorikus Optimalizálási Kutatócsoport (Egerváry Combinatorial Optimization Research Group, EGRES).

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

...
Introductory Example

Let us build a directed graph, assign costs to the arcs and run Dijkstra’s algorithm on it.
Let us build a directed graph, assign costs to the arcs and run Dijkstra’s algorithm on it.

```cpp
typedef adjacency_list<listS, vecS,
    directedS, no_property,
    property<edge_weight_t, int> > graph_t;

graph_t g;
property_map<graph_t, edge_weight_t>::type
    length = get(edge_weight, g);

graph_traits<graph_t>::vertex_descriptor
    s = add_vertex(g), t = add_vertex(g);
    // add more vertices

graph_traits<graph_t>::edge_descriptor
    e = add_edge(s, t, g).first;
    length[e] = 8;
    // add more edges

vector<int> dist(num_vertices(g));

// BGL code
```

```cpp
BGL code
```

Programs using LEMON tend to be shorter and easier to understand.
Introductory Example

Let us build a directed graph, assign costs to the arcs and run Dijkstra’s algorithm on it.

BGL code

```cpp
typedef adjacency_list<listS, vecS, directedS, no_property,
    property<edge_weight_t, int> > graph_t;

graph_t g;

property_map<graph_t, edge_weight_t>::type length = get(edge_weight, g);

graph_traits<graph_t>::vertex_descriptor s = add_vertex(g), t = add_vertex(g);
// add more vertices

graph_traits<graph_t>::edge_descriptor e = add_edge(s, t, g).first;
length[e] = 8;
// add more edges

vector<int> dist(num_vertices(g));
dijkstra_shortest_paths(g, s,
    distance_map(&dist[0]));
```

LEMON code

```cpp
ListDigraph g;
ListDigraph::ArcMap<int> length(g);
ListDigraph::ArcMap<int> length(g);

ListDigraph::Node s = g.addNode();
ListDigraph::Node t = g.addNode();
// add more nodes

ListDigraph::Arc a = g.addArc(s, t);
length[a] = 8;
// add more arcs

ListDigraph::NodeMap<int> dist(g);
dijkstra(g, length)
    .distMap(dist).run(s);
```

Programs using LEMON tend to be shorter and easier to understand.
LEMON contains very efficient graph implementations (both in terms of running time and memory space).

They have easy-to-use interface.
LEMON contains very efficient graph implementations (both in terms of running time and memory space).

They have easy-to-use interface.

Generic design:

- C++ template programming is heavily used.
- There are generic graph concepts and several graph implementations for diverging purposes.
- The algorithms work with arbitrary graph structures.
- Users can also write their own graph classes.
Creating a graph

```cpp
using namespace lemon;
ListDigraph g;
```
Creating a graph

```cpp
using namespace lemon;
ListDigraph g;
```

Adding nodes and arcs

```cpp
ListDigraph::Node u = g.addNode();
ListDigraph::Node v = g.addNode();
ListDigraph::Arc a = g.addArc(u, v);
```
Working with Graphs

Creating a graph

```cpp
using namespace lemon;
ListDigraph g;
```

Adding nodes and arcs

```cpp
ListDigraph::Node u = g.addNode();
ListDigraph::Node v = g.addNode();
ListDigraph::Arc a = g.addArc(u,v);
```

Removing items

```cpp
g.erase(a);
g.erase(v);
```
The graph structures provide several *iterators* for traversing the nodes and arcs.

**Iteration on nodes**
```
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) {...}
```

**Iteration on arcs**
```
for (ListDigraph::ArcIt a(g); a != INVALID; ++a)
for (ListDigraph::OutArcIt a(g,v); a != INVALID; ++a)
for (ListDigraph::InArcIt a(g,v); a != INVALID; ++a)
```

*Note:* INVALID is a constant, which converts to each and every iterator and graph item type.
The graph structures provide several *iterators* for traversing the nodes and arcs.

### Iteration on nodes

```cpp
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) {...}
```
Iterators

The graph structures provide several *iterators* for traversing the nodes and arcs.

**Iteration on nodes**

```cpp
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) {...}
```

**Iteration on arcs**

```cpp
for (ListDigraph::ArcIt a(g); a != INVALID; ++a)
for (ListDigraph::OutArcIt a(g,v); a != INVALID; ++a)
for (ListDigraph::InArcIt a(g,v); a != INVALID; ++a)
```

*Note*: `INVALID` is a constant, which converts to each and every iterator and graph item type.
Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use `operator*()`.
This provides a more convenient interface.
The program context always indicates whether we refer to the iterator or to the graph item.
Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use `operator*()`.
This provides a more convenient interface.
The program context always indicates whether we refer to the iterator or to the graph item.

Example: printing node identifiers

```cpp
for (ListDigraph::NodeIt v(g); v != INVALID; ++v)
    std::cout << g.id(v) << std::endl;
```
Iterators

- Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use `operator*()`.
- This provides a more convenient interface.
- The program context always indicates whether we refer to the iterator or to the graph item.

**Example: printing node identifiers**

```cpp
for (ListDigraph::NodeIt v(g); v != INVALID; ++v)
    std::cout << g.id(v) << std::endl;
```

On the other hand, BGL iterators strictly follow the STL concepts:

**BGL example 1**

```cpp
traits_t::vertex_iterator vi, vend;
for (tie(vi, vend) = vertices(g); vi != vend; ++vi)
    std::cout << *vi << std::endl;
```

This can be made much simpler using special macros:

**BGL example 2**

```cpp
BGL_FORALL_VERTICES(v, g, graph_t)
    std::cout << v << std::endl;
```
Example: printing node identifiers

```cpp
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) ← iterator
    std::cout << g.id(v) << std::endl; ← item
```

On the other hand, BGL iterators strictly follow the STL concepts:

```
traits_t::vertex_iterator vi, vend;
for (tie(vi, vend) = vertices(g); vi != vend; ++vi)
    std::cout << *vi << std::endl;
```

This can be made much simpler using special macros:

```
BGL_FORALL_VERTICES(v, g, graph_t)
    std::cout << v << std::endl;
```
Iterators

Example: printing node identifiers

```cpp
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) ← iterator
    std::cout << g.id(v) << std::endl; ← item
```

On the other hand, BGL iterators strictly follow the STL concepts:

BGL example 1

```cpp
traits_t::vertex_iterator vi, vend;
for (tie(vi, vend) = vertices(g); vi != vend; ++vi)
    std::cout << *vi << std::endl;
```
Iterators

Example: printing node identifiers

```cpp
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) ← iterator
    std::cout << g.id(v) << std::endl; ← item
```

On the other hand, BGL iterators strictly follow the STL concepts:

**BGL example 1**

```cpp
traits_t::vertex_iterator vi, vend;
for (tie(vi, vend) = vertices(g); vi != vend; ++vi)
    std::cout << *vi << std::endl;
```

This can be made much simpler using special macros:

**BGL example 2**

```cpp
BGL_FORALL_VERTICES(v, g, graph_t)
    std::cout << v << std::endl;
```
In LEMON, the graph classes represent only the pure structure of the graph.

All associated data (e.g. node labels, arc costs or capacities) are stored separately using so-called *maps*.

```cpp
ListDigraph::NodeMap<string> label(g);
ListDigraph::ArcMap<int> cost(g);
```

Accessing map values:

```cpp
label[v] = "source";
cost[e] = 2 * cost[f];
```
In LEMON, the graph classes represent only the pure structure of the graph.

All associated data (e.g. node labels, arc costs or capacities) are stored separately using so-called maps.

Creating maps

```cpp
ListDigraph::NodeMap<string> label(g);
ListDigraph::ArcMap<int> cost(g);
```
In LEMON, the graph classes represent only the pure structure of the graph.

All associated data (e.g. node labels, arc costs or capacities) are stored separately using so-called maps.

Creating maps

```
ListDigraph::NodeMap<string> label(g);
ListDigraph::ArcMap<int>    cost(g);
```

Accessing map values

```
label[v] = "source";
cost[e] = 2 * cost[f];
```
Benefits of Graph Maps

**Efficient.** Accessing map values is as fast as reading or writing an array.
Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.

- **Dynamic.** You can create and destruct maps freely.
  - Whenever you need, you can allocate a new map.
  - When you leave its scope, the map will be deallocated automatically.
  - The lifetimes of maps are not bound to lifetime of the graph.
Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.

- **Dynamic.** You can create and destruct maps freely.
  - Whenever you need, you can allocate a new map.
  - When you leave its scope, the map will be deallocated automatically.
  - The lifetimes of maps are not bound to lifetime of the graph.

- **Automatic.** The maps are updated automatically on the changes of the graph.
  - If you add new nodes or arcs to the graph, the storage of the existing maps will be expanded and the new slots will be initialized.
  - If you remove items from the graph, the corresponding values in the maps will be properly destructed.
LEMON provides efficient and flexible implementations of several algorithms.

Basically, all algorithms are implemented as template classes.

However, function-type interface is also available for some of them. It provides more convenient but less flexible usage.
Algorithm Interfaces

Class interface

Function-type interface
Algorithm Interfaces

Class interface

- Complex initializations.
- Flexible execution control:
  - step-by-step execution,
  - multiple execution,
  - custom stop conditions.
- Complex queries.
- The used data structures (maps, heaps, etc.) can be changed.

Function-type interface
Algorithm Interfaces

Class interface
- Complex initializations.
- Flexible execution control.
- Complex queries.
- The used data structures (maps, heaps, etc.) can be changed.

Function-type interface
- Single execution: “this is the input”, “put the results here”.
- Simpler usage:
  - template parameters do not have to be given explicitly,
  - arguments can be set using named parameters,
  - temporary expressions can be passed as reference parameters.
- It provides less flexibility in the initialization, execution and queries.
Class interface

```cpp
Dijkstra<ListDigraph> dijkstra(g, length);
dijkstra.distMap(dist);
dijkstra.run(s);
```
Using Algorithms

Class interface

```cpp
Dijkstra<ListDigraph> dijkstra(g, length);
dijkstra.distMap(dist);

dijkstra.init();
dijkstra.addSource(s1); dijkstra.addSource(s2);
dijkstra.start();
```
Class interface

Dijkstra<ListDigraph> dijkstra(g, length);
dijkstra.distMap(dist);
dijkstra.init();
dijkstra.addSource(s1); dijkstra.addSource(s2);

while (!dijkstra.emptyQueue()) {
    ListDigraph::Node n = dijkstra.processNextNode();
    std::cout << dijkstra.dist(n) << std::endl;
}
Using Algorithms

Class interface

```cpp
Dijkstra<ListDigraph> dijkstra(g, length);
dijkstra.distMap(dist);
dijkstra.init();
dijkstra.addSource(s1); dijkstra.addSource(s2);
```

```
while (!dijkstra.emptyQueue()) {
    ListDigraph::Node n = dijkstra.processNextNode();
    std::cout << dijkstra.dist(n) << std::endl;
}
```

Function-type interface

```cpp
dijkstra(g, length).distMap(dist).run(s);
```
Besides standard graph structures, LEMON also provides *graph adaptor* classes.

They serve for considering other graphs in different ways using the storage and operations of the underlying structure.
Besides standard graph structures, LEMON also provides graph adaptor classes.

They serve for considering other graphs in different ways using the storage and operations of the underlying structure.

The adaptors also conform to the graph concepts, so they can be used like standard graph structures.

Another view of a graph can be obtained without having to modify or copy the actual storage.

This technique yields convenient and elegant codes.
Using Graph Adaptors

Obtaining a subgraph

SubDigraph adaptor

Original digraph
Using Graph Adaptors

Combining adaptors

Original digraph
Undirector adaptor
SubDigraph adaptor
Original digraph
LEMON provides a convenient, high-level common interface for linear programming (LP) and mixed integer programming (MIP) solvers.

Currently supported software packages:

- **GLPK**: open source (GNU license)
- **Clp, Cbc**: open source (COIN-OR LP and MIP solvers)
- **CPLEX**: commercial
- **SoPlex**: academic license

Additional wrapper classes for other solvers can be implemented easily.
Using LP Interface

Building and solving an LP problem

```cpp
Lp lp;
Lp::Col x1 = lp.addCol();
Lp::Col x2 = lp.addCol();

lp.max();
lp.obj(10 * x1 + 6 * x2);

lp.addRow(0 <= x1 + x2 <= 100);
lp.addRow(2 * x1 <= x2 + 32);

lp.colLowerBound(x1, 0);

lp.solve();
std::cout << "Solution: " << lp.primal() << std::endl;
std::cout << "x1 = " << lp.primal(x1) << std::endl;
std::cout << "x2 = " << lp.primal(x2) << std::endl;
```
Using LP Interface

Building and solving an LP problem

```cpp
Lp lp;
Lp::Col x1 = lp.addCol();
Lp::Col x2 = lp.addCol();

lp.max();
lp.obj(10 * x1 + 6 * x2);
lp.addRow(0 <= x1 + x2 <= 100);
lp.addRow(2 * x1 <= x2 + 32);
lp.colLowerBound(x1, 0);

lp.solve();
std::cout << "Solution: " << lp.primal() << std::endl;
std::cout << "x1 = " << lp.primal(x1) << std::endl;
std::cout << "x2 = " << lp.primal(x2) << std::endl;
```

Mathematical formulation

max \(10x_1 + 6x_2\)

\[0 \leq x_1 + x_2 \leq 100\]

\[2x_1 \leq x_2 + 32\]

\[x_1 \geq 0\]
Technical Support

- Extensive documentation:
  - Reference manual (generated using Doxygen)
  - Tutorial
- Mailing lists.
- Version control (Mercurial).
- Bug tracker system (Trac).
- Build environment:
  - Autotools (Linux)
  - CMake (Windows)
- Support of different compilers:
  - GNU C++
  - Intel C++
  - IBM xIC
  - Microsoft Visual C++
2. Implementation Details
A graph concept should be:

- **Convenient and flexible**: to support various use cases, which usually requires overlapping functionalities.
- **Simple**: to make the implementation of new graph structures as easy as possible.

These requirements are clearly contradictory. Therefore, two-level graph concepts were developed in LEMON.
A graph concept should be:

- **Convenient and flexible**: to support various use cases, which usually requires overlapping functionalities.
- **Simple**: to make the implementation of new graph structures as easy as possible.

These requirements are clearly contradictory.

Therefore, two-level graph concepts were developed in LEMON.
Extending Graph Interfaces Using Mixins

- The *low-level* graph concepts define only the very basic graph functionalities:
  - Node and Arc classes,
  - simple function-based iteration, etc.
- These simple interfaces are extended to the *user-level* concepts, which define a wide range of member functions and nested classes.

**Low-level graph interface**

```cpp
class DigraphBase {
public:
    // Node and Arc classes
    class Node { ... };
    class Arc { ... };

    // Basic iteration
    void first(Node& node) const;
    void next(Node& node) const;
    ...
};
```
The template *Mixin* strategy is used: if *DigraphBase* implements the low-level interface, then *DigraphExtender<DigraphBase>* will fulfill the user-level concept.
The graph maps are external, auto-updated structures. To ensure efficient data access, they are implemented using arrays or `std::vector`s.

These structures have to be extended when new nodes or arcs are added to the graph. The graph and map classes implement the Observer design pattern. The graph maps guarantee strong exception safety. If a node or arc is inserted into a graph, but an attached map cannot be extended, then each map extended earlier is rolled back to its original state.
The graph maps are external, auto-updated structures.

To ensure efficient data access, they are implemented using arrays or `std::vector`.

These structures have to be extended when new nodes or arcs are added to the graph.

The graph and map classes implement the *Observer* design pattern.
The graph maps are external, auto-updated structures.
To ensure efficient data access, they are implemented using arrays or `std::vector`s.
These structures have to be extended when new nodes or arcs are added to the graph.
The graph and map classes implement the `Observer` design pattern.
The graph maps guarantee strong exception safety.
If a node or arc is inserted into a graph, but an attached map cannot be extended, then each map extended earlier is rolled back to its original state.
The performance and the functionality of generic libraries can be further improved by template specializations.

In LEMON, tags are defined for several purposes, e.g. the graphs are marked with UndirectedTag.

**Tags for graphs**

```cpp
class ListDigraph {
    typedef False UndirectedTag;
    ...
};

class ListGraph {
    typedef True UndirectedTag;
    ...
};
```
For example, the function `eulerian()` is specialized for undirected graphs.

- A directed graph is Eulerian if it is connected and the number of incoming and outgoing arcs are the same for each node.
- An undirected graph is Eulerian if it is connected and the number of incident edges is even for each node.

```cpp
template<
typename GR>
typename enable_if<
typename GR::UndirectedTag, bool>::type
eulerian(const GR &g) {
    for (typename GR::NodeIt n(g); n != INVALID; ++n)
        if (countIncEdges(g, n) % 2 == 1)
            return false;
    return connected(g);
}
```
3. Performance
This section thoroughly compares the performance of LEMON to BGL and LEDA.
Benchmark results for **Dijkstra’s algorithm**:

- **BGL** is more efficient than **LEDA**, especially on dense graphs.
- **LEMON** is even slightly faster than **BGL**.
Benchmark results for the **preflow push-relabel algorithm**:

- **LEDA** is clearly faster than **BGL**, especially on sparse networks.
- **LEMON** is more efficient than both of them.
Minimum Cost Flows

- **BGL** does not provide a minimum cost flow algorithm, but it has been among their plans for a long time.
- **LEMON** and **LEDA** provide efficient implementations of the *cost scaling* algorithm (and some other methods).
Benchmark results for the **cost scaling algorithm**:

- **LEMON** clearly outperforms **LEDA**.
- **LEDA** failed on the largest sparse networks with “cost overflow” error. However, larger number type cannot be used due to the closed source.
Benchmark results for the **planar embedding method**:

- **LEDA** is much slower than **BGL**.
- **LEMON** is about two times faster than **BGL**.
4. History and Statistics
History of LEMON

2003–2007  LEMON 0.x series
- Development versions without stable API.
- Latest release: LEMON 0.7.

2008–    LEMON 1.x series
- Stable releases ensuring full reverse compatibility.
- Major versions:
  2008-10-13  LEMON 1.0 released
  2009-05-13  LEMON 1.1 released
  2010-03-19  LEMON 1.2 released

2009-03-27  LEMON joins to the COIN-OR initiative.
- http://www.coin-or.org/
SLOC – Source Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>lemon</th>
<th>test</th>
<th>tools</th>
<th>scripts</th>
<th>demo</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>45,032</td>
<td>8340</td>
<td>983</td>
<td>238</td>
<td></td>
<td>54,593</td>
</tr>
<tr>
<td>Python</td>
<td>130</td>
<td>478</td>
<td></td>
<td></td>
<td></td>
<td>608</td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>513</td>
</tr>
<tr>
<td>Total</td>
<td>45,032</td>
<td>8340</td>
<td>1113</td>
<td>991</td>
<td>238</td>
<td>55,714</td>
</tr>
</tbody>
</table>

B. Dezső, A. Jüttner, P. Kovács

LEMON – an Open Source C++ Graph Template Library
5. Conclusions
LEMON is a **highly efficient, open source** C++ graph template library having clear design and convenient interface.

Comparing to similar libraries, LEMON shows remarkable advantages both in **ease of use** and in **performance**.

Its essential algorithms turned out to be significantly more efficient than BGL and LEDA.
LEMON is a highly efficient, open source C++ graph template library having clear design and convenient interface.

Comparing to similar libraries, LEMON shows remarkable advantages both in ease of use and in performance.

Its essential algorithms turned out to be significantly more efficient than BGL and LEDA.

For these reasons, LEMON is proved to be a remarkable alternative to open source or commercial graph libraries.

LEMON is favorable for research, education and development in the area of combinatorial optimization and network design.
Thank you for the attention!

http://lemon.cs.elte.hu