LEMON – an Open Source C++ Graph Template Library

Balázs Dezső[1], Alpár Jüttner[2], Péter Kovács[1]

[1]Department of Algorithms and Their Applications
Eötvös Loránd University, Budapest, Hungary

{deba,kpeter}@inf.elte.hu

[2]Department 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

Introduction to LEMON

What is LEMON?

- **LEMON** is an abbreviation for

  Library for **E**fficient **M**odeling and **O**ptimization in **N**etworks.
- 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`

Sponsors: EGRES, Mobil Innovációs Központ, ERICSSON

Introduction to LEMON

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).

LICENSE (same as BOOST)

Introductory Example

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

Introductory Example

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

```
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]));
```

BGL code

Introductory Example

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

```
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]));
```

BGL code

```
ListDigraph 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);
```

LEMON code

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

Graph Structures

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

Graph Structures

- 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.

Working with Graphs

Creating a graph

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

Working with Graphs

Creating a graph

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

Adding nodes and arcs

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

Working with Graphs

Creating a graph

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

Adding nodes and arcs

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

Removing items

```
g.erase(a);
g.erase(v);
```

Iterators

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

Iterators

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

Iteration on nodes

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

Iterators

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.

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.

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

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

- 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

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

Iterators

Example: printing node identifiers

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

Iterators

Example: printing node identifiers

```
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

```
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

```
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

```
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

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

Handling Graph Related Data

- 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*.

Handling Graph Related Data

- 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);
```

Handling Graph Related Data

- 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.

Using Algorithms

Class interface

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

dijkstra.run(s);
```

Using Algorithms

Class interface

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

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

Using Algorithms

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

```
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

```
dijkstra(g, length).distMap(dist).run(s);
```

Graph Adaptors

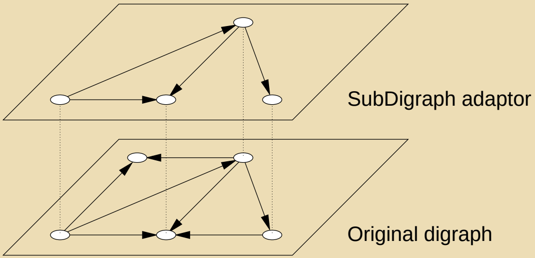
- Besides standard graph structures, LEMON also provides *graph adaptor* classes.
- They serve for considering other graphs in different ways using the storage and operations of the underlying structure.

Graph Adaptors

- 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.

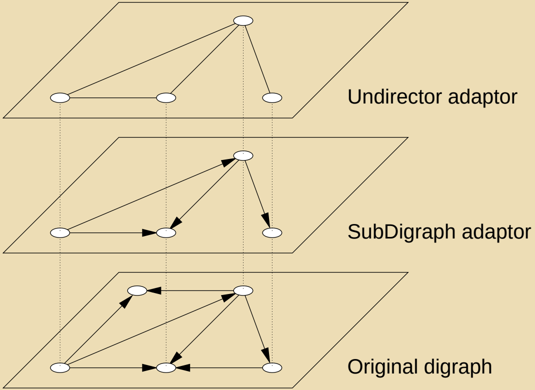
Using Graph Adaptors

Obtaining a subgraph



Using Graph Adaptors

Combining adaptors



LP Interface

- 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

```
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

```
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

$$\max 10x_1 + 6x_2$$

$$0 \leq x_1 + x_2 \leq 100$$

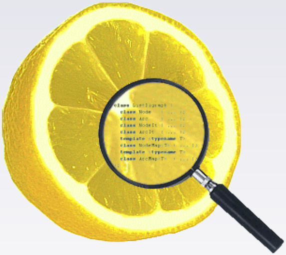
$$2x_1 \leq x_2 + 32$$

$$x_1 \geq 0$$

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 xlC
  - Microsoft Visual C++

2. Implementation Details



Design of Graph Concepts

- 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.

Design of Graph Concepts

- 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

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

  // Basic iteration
  void first(Node& node) const;
  void next(Node& node) const;
  ...
};
```

Extending Graph Interfaces Using Mixins

High-level graph interface

```
template <typename DigraphBase>
class DigraphExtender : public DigraphBase {
public:
  // Class-based iterators
  class NodeIt : public Node {
  public:
    NodeIt(const DigraphExtender& g) : _graph(g) {
      _graph.first(*this);
    }
    NodeIt& operator++() {
      _graph.next(*this);
      return *this;
    }
    ...
  private:
    const DigraphExtender& _graph;
  };
  ...
};
```

The template *Mixin* strategy is used: if `DigraphBase` implements the low-level interface, then `DigraphExtender<DigraphBase>` will fulfill the user-level concept.

Signaling Graph Alterations

- The graph maps are external, auto-updated structures.
- To ensure efficient data access, they are implemented using arrays or `std::vector`s.

Signaling Graph Alterations

- 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.

Signaling Graph Alterations

- 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.

Tags and Specializations

- 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

```
class ListDigraph {
  typedef False UndirectedTag;
  ...
};
class ListGraph {
  typedef True UndirectedTag;
  ...
};
```

Tags and Specializations

- 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.

Example: specialization using tags

```
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);
}
```

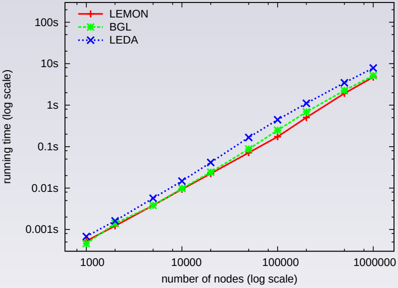
3. Performance

Performance

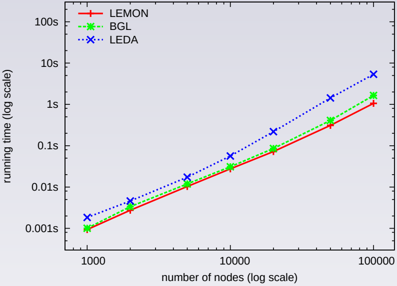
This section thoroughly compares the performance of **LEMON** to **BGL** and **LEDA**.

VS

Shortest Paths



Sparse networks

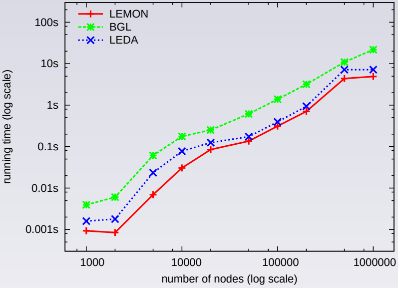


Dense networks

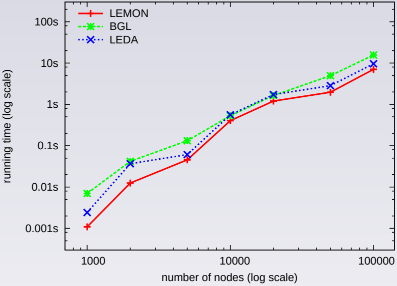
Benchmark results for **Dijkstra's algorithm**:

- **BGL** is more efficient than **LEDA**, especially on dense graphs.
- **LEMON** is even slightly faster than **BGL**.

Maximum Flows



Sparse networks

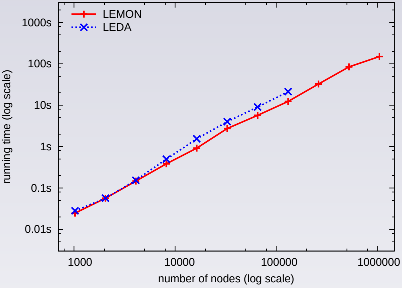


Dense networks

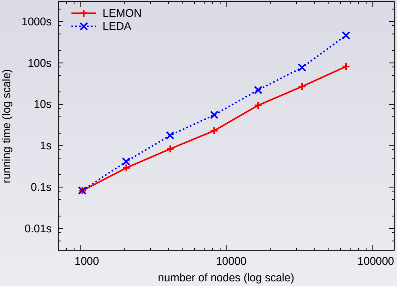
Benchmark results for the **preflow push-relabel algorithm**:

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

Minimum Cost Flows

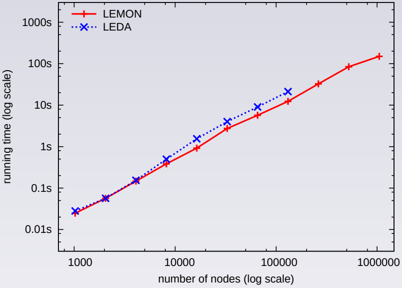


Sparse networks

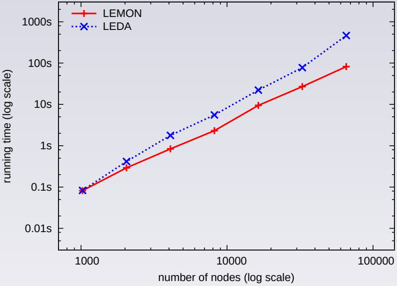


Dense networks

- **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).



Sparse networks

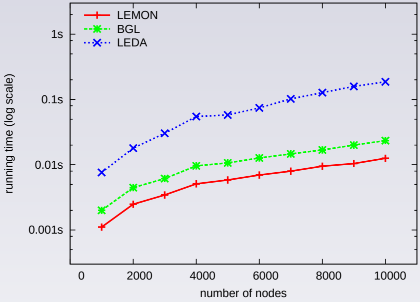


Dense networks

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.

Planar Embedding



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

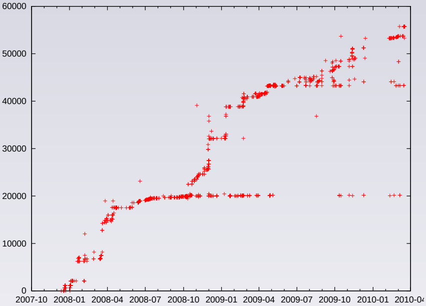
- Stable releases ensuring full reverse compatibility.
- Major versions:

  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



	lemon	test	tools	scripts	demo	Total	
C++	45,032	8340	983		238	**54,593**	(97.98%)
Python				513		**513**	(0.92%)
other			130	478		**608**	(1.09%)
Total:	45,032	8340	1113	991	238	**55,714**	

5. Conclusions

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.

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.
- 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

Thank you for the attention!

`http://lemon.cs.elte.hu`