LEMON

Library for Efficient Modeling and Optimization in Networks

Alpár Jüttner, Balázs Dezső, Péter Kovács

Dept. of Operations Research
Eötvös Loránd University, Budapest

April 30, 2010

Overview

1. Introduction to LEMON
   - What is LEMON?
   - Graph Structures
   - Iterators
   - Maps
   - Algorithms
   - Undirected Graphs
   - Graph Adaptors
   - LP Interface
   - Tools
   - Technical Support
   - Users
2. Performance
   - Shortest Paths
   - Maximum Flows
   - Minimum Cost Flows
   - Planar Embedding
3. History and Statistics
4. Summary

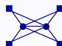
1 Introduction to LEMON

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:



Design Goals

- Genericity:
  - clear separation of data structures and algorithms,
  - excessive use of modern software development paradigms.
- 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).

Motivation

- We missed a tool like that.
  - Source code of commercial libraries cannot be examined or improved.
  - Other open source graph libraries are not well designed and lack sophisticated implementations.
- We observed a need for such a graph library in the (industrial) research.
  - Commercial tools are not always appropriate.
- We wanted something better than the existing ones.
- We wanted to attract more people to applied research.
  - Students participating in the development of LEMON
    - learn programming by implementing new features,
    - get experience in collaborative development,
    - can more easily be involved in other research projects.

LICENSE (same as BOOST)

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.

...

LEMON Components

- Efficient **data structures** for graphs and related data.
- Flexible and very fast implementations of several combinatorial **algorithms** (graph search, shortest paths, flow and matching problems etc.).
- Input-output support for various graph file formats.
  - An own flexible file format: LGF.
- A uniform and high-level interface for different LP and MIP solvers (GLPK, Clp, Cbc, CPLEX, SoPlex).
- Detailed documentation.
- Build environment.

Graph Structures

- LEMON contains highly 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.

Building 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.

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;
}
```

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) {   ← iterator
  std::cout << g.id(v) << std::endl;                ← item
}
```

Maps

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

Accessing map values

```
label[s] = "source";
cost[e] = 2 * cost[f];
```

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 the 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.

Map Concepts

- LEMON maps are not just storage classes. They are concepts of any key–value based data access.
- This design provides great flexibility.

Map Concepts

- LEMON maps are not just storage classes. They are concepts of any key–value based data access.
- This design provides great flexibility.
  - Users can write their own maps.

Example: Constant map

```
class MyMap {
public:
  typedef ListDigraph::Arc Key;
  typedef double Value;
  Value operator[](Key k) const { return 3.14; }
};
```

Map Concepts

- LEMON maps are not just storage classes. They are concepts of any key–value based data access.
- This design provides great flexibility.
  - Users can write their own maps.
  - LEMON provides several “lightweight” *map adaptor* classes in addition to the standard graph maps.

Map Adaptors

- A map adaptor class is attached to another map and performs various operations on the original data.
- This is done “on the fly” when the access operations are called.
- The original storage is neither modified nor copied.

Map Adaptors

- A map adaptor class is attached to another map and performs various operations on the original data.
- This is done “on the fly” when the access operations are called.
- The original storage is neither modified nor copied.

Standard method

```
ListDigraph::ArcMap<int> cost(g);
...  // set arc costs
algorithm(g, cost);
...  // increase arc costs by 100
algorithm(g, cost);
```

Map Adaptors

- A map adaptor class is attached to another map and performs various operations on the original data.
- This is done “on the fly” when the access operations are called.
- The original storage is neither modified nor copied.

Using map adaptor

```
ListDigraph::ArcMap<int> cost(g);
...  // set arc costs
algorithm(g, cost);
algorithm(g, shiftMap(cost, 100));
```

Algorithms I.

Graph search

- BFS, DFS

Algorithms I.

Graph search

Shortest paths

- Dijkstra (with various heaps)
- Bellman-Ford
- Suurballe

Algorithms I.

Graph search

Shortest paths

Spanning trees

- Kruskal
- Min. cost arborescence

Algorithms I.

Graph search

Shortest paths

Spanning trees

Connectivity

- Connected components
- Strongly connected components
- Topological ordering
- 2-connected components, articulation nodes

Algorithms I.

Graph search

Shortest paths

Spanning trees

Connectivity

Algorithms II.

Network flows

- Maximum flow
- Feasible circulation
- Minimum cost flow
- Minimum cut

Algorithms II.

Network flows

- Maximum flow
  - Preflow push-relabel (with various heuristics)
- Feasible circulation
- Minimum cost flow
- Minimum cut

Algorithms II.

Network flows

- Maximum flow
- Feasible circulation
    - generalized version (push-relabel algorithm)
- Minimum cost flow
- Minimum cut

Algorithms II.

Network flows

- Maximum flow
- Feasible circulation
- Minimum cost flow
  - Cycle-canceling (3 versions)
  - Capacity scaling
  - Cost scaling (3 versions with various heuristics)
  - Network simplex (5 different pivot rules)
- Minimum cut

Algorithms II.

Network flows

- Maximum flow
- Feasible circulation
- Minimum cost flow
- Minimum cut
  - Hao-Orlin
  - Gomory-Hu

Algorithms II.

Network flows

Matching

- Max. cardinality matching
- Max. weighted matching
- Fractional matching

Algorithms II.

Network flows

Matching

Minimum mean cycle

- Karp
- Hartmann-Orlin
- Howard

Algorithms II.

Network flows

Matching

Minimum mean cycle

Planar graphs

- Planar embedding
- Planar drawing

Algorithms II.

Network flows

Matching

Minimum mean cycle

Planar graphs

Auxiliary algorithms

- Radix sort

Algorithms II.

Network flows

Matching

Minimum mean cycle

Planar graphs

Auxiliary algorithms

Others

Data structures

- Graphs and maps
- Path structures
- Priority queues (binary, D-ary, Fibonacci, pairing, radix, bucket, etc. heaps)
- Union-find structures
- Auxiliary data structures

Others

Data structures

Input-output support

- LEMON Graph Format (LGF)
- Other graph formats: DIMACS, Nauty
- EPS exporting

Others

Data structures

Input-output support

General optimization tools

- High-level common interface for LP and MIP solvers

Others

Data structures

Input-output support

General optimization tools

Mersenne-Twister pseudo-random generator

- Best statistical properties and very fast
- Uniform, Gauss, Exponential, Pareto, Weibull, etc. distributions

Others

Data structures

Input-output support

General optimization tools

Mersenne-Twister pseudo-random generator

Auxiliary tools

- Time measuring tools and counters
- Command line argument parser
- etc.

Others

Data structures

Input-output support

General optimization tools

Mersenne-Twister pseudo-random generator

Auxiliary tools

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> dijk(g, length);
dijk.distMap(dist);


dijk.run(s);


std::cout << dist[t] << std::endl;
```

Using Algorithms

Class interface

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

dijk.init();
dijk.addSource(s1); dijk.addSource(s2);
dijk.start();

std::cout << dist[t] << std::endl;
```

Using Algorithms

Class interface

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

dijk.init();
dijk.addSource(s1); dijk.addSource(s2);
dijk.start();

std::cout << dist[t] << std::endl;
```

Function-type interface

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

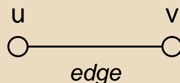
Using Algorithms

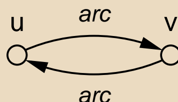
Example: Number of connected components

```
template <typename GR>
int numberOfComponents(const GR &g) {
  int c = 0;
  Bfs<GR> bfs(g);
  bfs.init();
  for (typename GR::NodeIt v(g); v != INVALID; ++v) {
    if (!bfs.reached(v)) {
      c++;
      bfs.addSource(v);
      bfs.start();
    }
  }
  return c;
}
```

Undirected Graphs

- An *undirected* graph is also considered as a *directed* one at the same time.
- Each undirected *edge* can also be regarded as two oppositely directed *arcs*.
- As a result, each algorithm working on a directed graph naturally runs on an undirected graph, as well.





Graph Adaptors

- LEMON also provides *graph adaptor* classes (similarly to map adaptors).
- They serve for considering other graphs in different ways using the storage and operations of the original structure.
- Another view of a graph can be obtained without having to modify or copy the actual storage.
- This technique yields convenient and elegant codes.

Graph Adaptors

- LEMON also provides *graph adaptor* classes (similarly to map adaptors).
- They serve for considering other graphs in different ways using the storage and operations of the original structure.
- Another view of a graph can be obtained without having to modify or copy the actual storage.
- This technique yields convenient and elegant codes.

Working with the original graph

```
algorithm(g);
```

Graph Adaptors

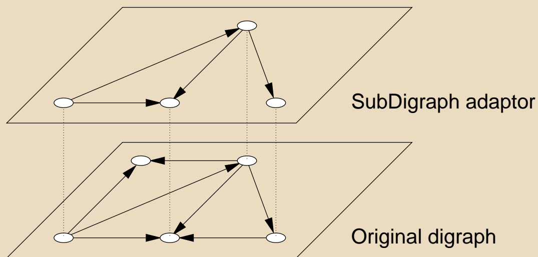
- LEMON also provides *graph adaptor* classes (similarly to map adaptors).
- They serve for considering other graphs in different ways using the storage and operations of the original structure.
- Another view of a graph can be obtained without having to modify or copy the actual storage.
- This technique yields convenient and elegant codes.

Working with the reverse oriented graph

```
algorithm(reverseDigraph(g));
```

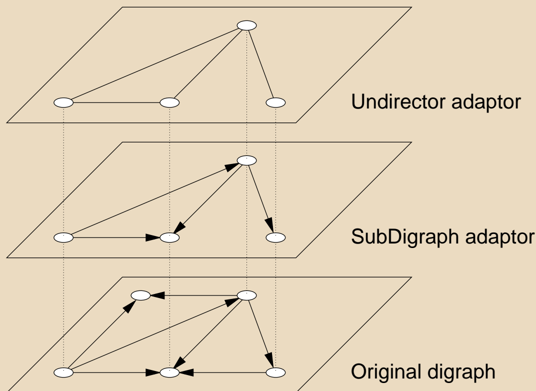
Using Graph Adaptors

Obtaining a subgraph



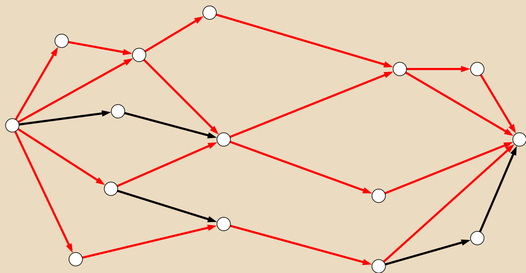
Using Graph Adaptors

Combining adaptors



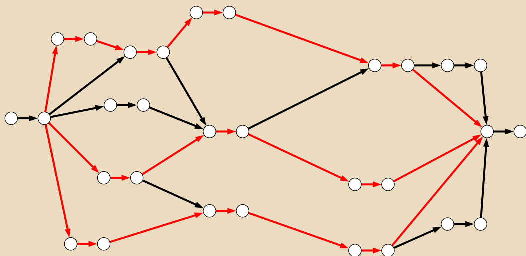
Example: Finding Disjoint Paths

- The maximum number of *arc disjoint* paths between two nodes can be found by computing a maximum flow using uniform arc capacities.



Example: Finding Disjoint Paths

- For finding *node disjoint* paths, the maximum flow algorithm can be used in conjunction with the `SplitNodes` adaptor.
- It splits each node into an *in-node* and an *out-node*.



LP Interface

- LEMON provides a convenient, high-level interface for several *linear programming* (LP) and *mixed integer programming* (MIP) solvers.
- Currently supported:

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 the 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 the 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

$$\max 10x_1 + 6x_2$$

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

$$2x_1 \leq x_2 + 32$$

$$x_1 \geq 0$$

Example graph

```
@nodes
label coordinate
0     (20,100)
1     (40,120)
...
41    (600,100)
@arcs
            label length
0     1     0     16
0     2     1     12
2     12    2     20
...
36    41    123   21
@attributes
source 0
caption "A shortest path problem"
```

LGF – LEMON Graph Format

Example graph

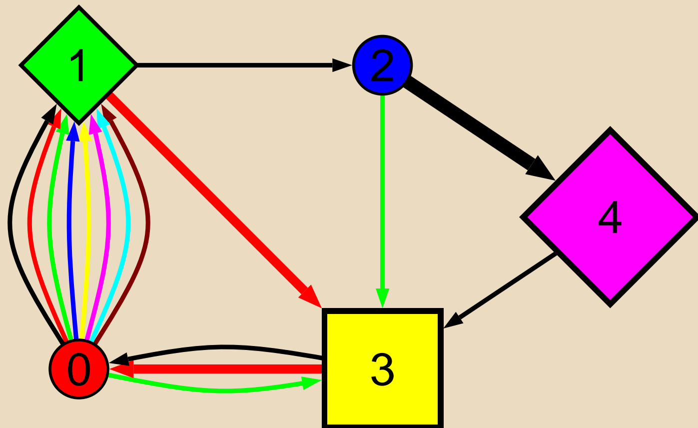
```
@nodes
label coordinate
0     (20,100)
...
41    (600,100)
@arcs
          label length
0    1    0    16
...
36   41   123  21
@attributes
source 0
caption "A shortest path problem"
```

Reading the graph

```
digraphReader(g, "input.lgf")
   .nodeMap("coord", coord)
   .arcMap("length", length)
   .attribute("caption", title)
   .node("source", src)
   .run();
```

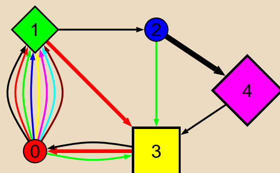
Postscript Exporting

EPS example



Postscript Exporting

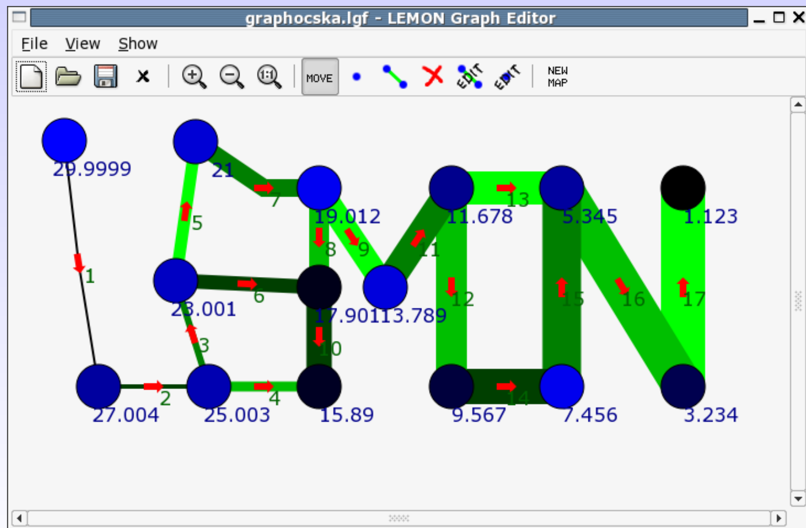
EPS example



Exporting to EPS

```
graphToEps(g, "graph.eps")
  .copyright("(c) 2003-2010 LEMON Project")
  .title("Sample EPS figure").coords(coords)
  .nodeScale(2).nodeSizes(sizes).nodeTexts(id).nodeTextSize(3)
  .nodeColors(composeMap(palette, colors)).nodeShapes(shapes)
  .arcColors(composeMap(palette, acolors))
  .arcWidthScale(.3).arcWidths(widths)
  .drawArrows().arrowWidth(1).arrowLength(1)
  .enableParallel().parArcDist(1)
  .run();
```

gLEMON – A Graph Editor for LEMON



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++

Users of LEMON

- Several projects at ELTE and BME, Budapest
- ZIB, Berlin
- Ericsson Research
- CPqD, Brazil
- Personal users
- etc.

- LEMON is a part of the COIN-OR software collection.
  `http://www.coin-or.org/`

User Reviews

"I use LEMON to solve the problem of tangent plane orientation in surface reconstruction... In my project I decided to use LEMON because it was easier to understand and use than other projects like "boost" and I also found that it runs fast."

"I searched long and hard for a graph library I could use. Yours was the only library that is robust, has documentation that makes sense, and is not obfuscated by excessive use of templates. For example, I spent quite some time with the Boost Graph Library and got nowhere. It builds generality upon generality."

2 Performance

Performance

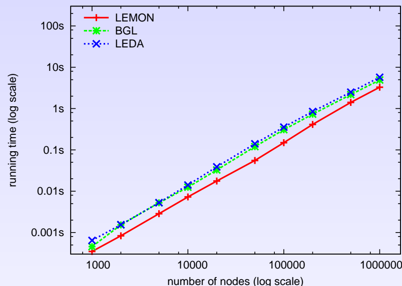
The performance of **LEMON** is compared to its major competitors:

- **BGL** – Boost Graph Library
- **LEDA** library

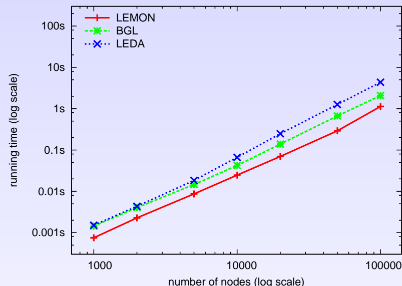
VS

Shortest Paths

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



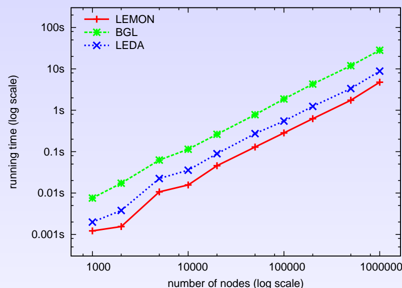
Sparse networks



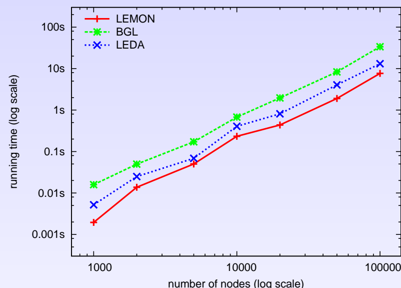
Dense networks

Maximum Flows

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



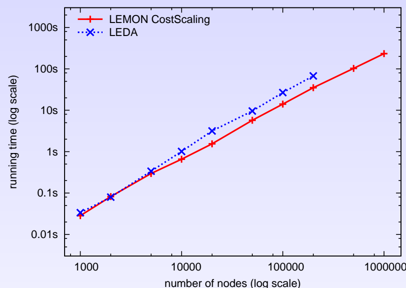
Sparse networks



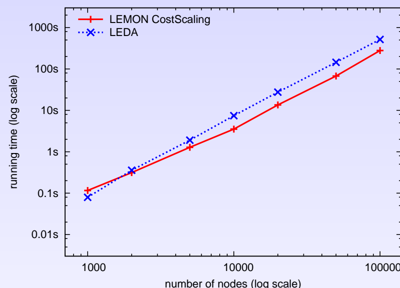
Dense networks

Minimum Cost Flows

Benchmark results for **minimum cost flow algorithms**:



Sparse networks

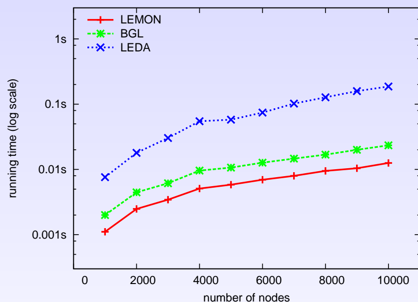


Dense networks

Note: BGL does not provide a minimum cost flow algorithm, but it has been among the plans of the developers for a long time.

Planar Embedding

Benchmark results for **planar embedding algorithms**:



Section 3

3 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

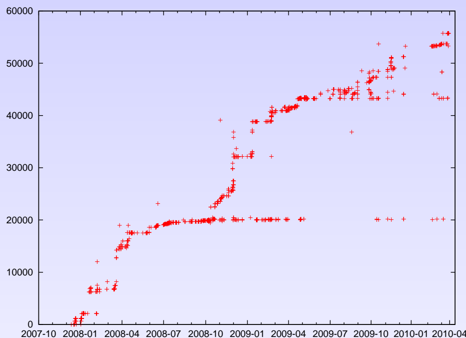
- Stable releases ensuring full backward compatibility.
- Major versions:

  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**	

4 Summary

Summary

- LEMON is a **highly efficient** C++ graph template library providing **convenient** and **flexible** components.
- It is **open source** with a very **premissive license**.
- LEMON turned out to be significantly more efficient than its two major competitors, BGL and LEDA.
- Therefore, LEMON is favorable for both research and development in the areas of combinatorial optimization and network design.

The End

Thank you for the attention!

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