LEMON – an Open Source C++ Graph Template Library [1]

Balázs Dezső[2]

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

Alpár Jüttner[2]

Centre for Wireless Network Design
University of Bedfordshire
Luton, United Kingdom

Department of Operations Research
Eötvös Loránd University
Budapest, Hungary

Péter Kovács[2]

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

Abstract

This paper introduces LEMON, a generic open source C++ library of graph and network algorithms and related data structures. It is a package of highly efficient and versatile tools with simple and convenient interface, targeting both computer scientists and the operations research community, as well as serving educational purposes. In this paper, the basic design concepts, features and performance of LEMON are compared to similar software packages, namely BGL (Boost Graph Library) and LEDA. Various implementation details are also discussed demonstrating the sophisticated use of C++ templates and other techniques. Due to its clear concepts and design, the ease of use and impressive performance, LEMON proved to be a remarkable alternative to similar open source or commercial libraries.

Keywords: C++, library, graph, network, generic, template, algorithm

1 Introduction

LEMON [16] is a C++ template library with focus on combinatorial optimization tasks connected mainly with graphs and networks. Its name is an abbreviation of **L**ibrary for **E**fficient **M**odeling and **O**ptimization in **N**etworks. LEMON is an open

[1] The LEMON project is supported by EGRES [9].

[2] E-mail: deba@inf.elte.hu, alpar@cs.elte.hu, kpeter@inf.elte.hu

source software project of Egerváry Research Group on Combinatorial Optimization (EGRES) [9] at Department of Operations Research, Eötvös Loránd University, Budapest, Hungary. It is also a member of the COIN-OR initiative [6], a collection of open source projects related to operations research. Its clear design and the permissive licensing scheme make LEMON favorable for commercial and non-commercial software development, as well as research purposes.

The goal of the library is to provide efficient, easy-to-use and well-cooperating software components, which help solving complex real-life optimization problems. These components include graph implementations and related data structures, fundamental graph and network algorithms (such as graph search, shortest path, spanning tree, matching and network flow algorithms) besides various auxiliary tools. Furthermore, the library provides a common high-level interface for several linear programming (LP) and mixed integer programming (MIP) [7] solvers.

The basic motivation for developing LEMON was to support researchers and practitioners working in the area of graph theory and network optimization by establishing an open source library that is more suitable for them than other alternatives on the market. At present, LEMON is extensively used for research, including network design, traffic routing and general graph theory [1,4,13,20], as well as in education at Eötvös Loránd University and Budapest University of Technology and Economics. It is also used in several commercial applications.

The rest of this paper is organized as follows. Section 2 provides an overview of the main concepts and features of LEMON compared to BGL [2] and LEDA [15], the most widely known and highly regarded C++ graph libraries. Section 3 describes selected details of the implementation demonstrating the applied language techniques. In Section 4, the performance of the discussed libraries is compared through benchmark tests of fundamental algorithms. Finally, some general conclusions are drawn in Section 5.

2 Overview

Probably, the best known C++ graph library is BGL, that is why the readers are introduced to LEMON through simple and equivalent sample codes using these two libraries. In Figure 1, both codes construct a directed graph, assign lengths to the arcs and run Dijkstra's algorithm. These examples also show that programs using LEMON tend to be shorter and easier to understand, since the user does not have to deal with various template techniques (like template metaprogram lists and traits classes).

2.1 Graph Structures

Although LEMON is a generic library, the main graph structures are not template classes, which is made possible by an important design decision. Namely, all data assigned to nodes and arcs are stored separately from the graph structures (see Section 2.3).

The example in Figure 1 uses `ListDigraph`, which is a doubly-linked adjacency list based directed graph implementation. Another important digraph structure is

```
typedef adjacency_list<listS, vecS,
  directedS, no_property,
  property<edge_weight_t, int> > graph_t;
typedef graph_traits<graph_t> traits_t;
graph_t g;
property_map<graph_t, edge_weight_t>::type
  length = get(edge_weight, g);

traits_t::vertex_descriptor
  s = add_vertex(g), t = add_vertex(g);
// add more vertices

traits_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]));
```

```
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);
```

Fig. 1. Sample codes using BGL and LEMON, respectively

`SmartDigraph`, which stores the nodes and arcs continuously in vectors and uses simply-linked lists for keeping track of the incident arcs of each node. Therefore, it has smaller memory footprint and it can be considerably faster, at the cost that nodes and arcs cannot be removed from it. `ListGraph` and `SmartGraph` are the undirected versions of these classes. In addition to these general structures, LEMON also contains special purpose classes for handling full graphs, grid graphs and hypercube graphs.

BGL implements a single adjacency list based graph class, but it can be customized with template parameters that specify the internal storage structures for nodes and arcs. BGL also contains a matrix based graph type, which is missing from LEMON, but the full graph structures with subgraph adaptors (see Section 2.5) provide reasonable substitutes for that.

In LEMON, the undirected `Graph` concept also fulfills the `Digraph` concept, in such a way that each *edge* of a graph can also be regarded as two oppositely directed *arcs*. As a result, all directed graph algorithms automatically run on undirected graphs, as well (provided it makes sense). Furthermore, some algorithms can be implemented simpler (e.g., planar graph algorithms), because we can distinguish the undirected edges from their directed variants. This solution entirely differs from BGL, where directed and undirected graphs have the same interfaces, but with different semantics. Using undirected graphs in BGL, the edges are usually considered undirected, but they have directions in some cases (for example, in iterations), which could be confusing. Moreover, BGL does not allow to define a property map whose keys are the directed variants of the edges, although it would also be useful in algorithms.

The LEDA graph implementation is closed source, but it is probably quite similar to the general graph structures of LEMON. The main difference is that digraphs and graphs are implemented in the same class in LEDA and there are member functions to switch between the two modes.

2.2 Iterators

LEMON iterators do not adhere to the STL compatibility, and so they provide a more convenient interface. They are initialized to the first item in the traversed

range by their constructors and their validity is checked by comparing them to a global constant `INVALID`. In addition, iterators are convertible to the corresponding item types, without having to use `operator*()`. This is not confusing, since the program context always indicates whether we refer to the iterator or to the graph item (they do not have conflicting functionalities).

Recall the examples shown in Figure 1. In LEMON, the computed distances of the nodes can be printed to the standard output like this.

```
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) {
  cout << "d[" << g.id(v) << "] = " << dist[v] << endl;
}
```

In contrast with this, BGL iterators strictly follow the STL requirements of input iterators. They must be dereferenced with the `operator*()` function in order to get the corresponding item descriptors. The `tie()` function can be used to make the code more compact and avoid programmer failures. In the BGL code, the node distances can be printed as follows.

```
traits_t::vertex_iterator vi, vend;
for (tie(vi, vend) = vertices(g); vi != vend; ++vi) {
  cout << "d[" << *vi << "] = " << dist[*vi] << endl;
}
```

BGL also provides several macros that simplify traversing graph items and define the loop variables only in the scope of the loop.

```
BGL_FORALL_VERTICES(v, g, graph_t) {
  cout << "d[" << v << "] = " << dist[v] << endl;
}
```

LEDA supports similar macros, but they do not allow to define the loop variables in the scope of the loop.

```
node v;
forall_nodes(v, g) {
  cout << "d[";
  g.printNode(v);
  cout << "] = " << dist[v] << endl;
}
```

2.3 Handling Graph Related Data

Beyond the graph structures, most graph algorithms need additional data associated to the nodes and the arcs. For example, shortest path algorithms require a length function on the arcs and they store the computed distance labels for the nodes. Graph libraries support handling these associated values in various ways. The data structures used for this purpose are typically called *maps* (not to be confused with `std::map`, which provides $O(\log n)$ time access to the elements). Since they are among the most frequently used data structures, maps have to be highly efficient and convenient.

LEMON library uses only external storage graph maps, but they are updated automatically on the changes of the graph (see Section 3.2). The main advantage of the external maps is that they can be constructed and destructed freely, their lifetimes are not determined by the lifetime of the graph. Moreover, separate storage could result in better caching properties, especially using several maps on a huge graph. In LEMON, you can declare maps like this.

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

The map values can be retrieved and modified using the corresponding overloaded versions of `operator[]()`.

```
label[v] = "source";
length[e] = 2 * length[f];
```

The LEMON maps are not only just storage classes, but also they are concepts of any key–value based data access. Beside the standard graph maps, LEMON contains several "lightweight" *map adaptor classes*, which perform various operations on the data of the adapted maps when their access operations are called, but without actually copying or modifying the original storage. These classes also conform to the map concepts, thus they can be used like standard LEMON maps.

Let us suppose that we have a traffic network stored in a LEMON graph structure with two arc maps `length` and `speed`, which denote the physical length of each arc and the maximum (or average) speed that can be achieved on the corresponding road-section, respectively. If we are interested in the best traveling times, the following code can be used.

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

BGL library uses both internal and external maps. The internal maps can be specified as property lists or bundled properties. The bundled properties have a simpler interface and their use is to be preferred. The advantage of internal maps is that they are updated automatically if the underlying graph is changed. However, the lifetimes of these maps are strictly bound to the graph.

LEDA uses only external storage for this purpose, but it provides two kinds of data structures. The *arrays* are not updated automatically, but they are implemented by vectors and their access operations take $O(1)$ time. The *map* types are more adaptable, they are not invalidated when the graph is changed. For this, the maps are implemented by hashing and their access functions take more time. Although these structures are external objects, additional space can be allocated in the graphs, which are called slots. The newly created arrays and maps can be assigned to empty slots, so the memory usage can be optimized.

2.4 Algorithms

LEMON provides several algorithms related to graph theory and combinatorial optimization. It contains well-known basic algorithms, such as breadth-first search (BFS), depth-first search (DFS), Dijkstra algorithm, Kruskal algorithm and methods for discovering graph properties like connectivity, bipartiteness or Euler property, as well as more complex optimization algorithms for finding maximum flows, minimum cuts, matchings, minimum cost flows and arc-disjoint paths. BGL and LEDA features similar algorithms, but with different interfaces.

In LEMON, the algorithms are implemented basically as classes, but for some of them, function-type interfaces are also available for the sake of convenience. For instance, the Dijkstra algorithm is implemented in the `Dijkstra` template class, but the `dijkstra()` function is also defined, which can still be used quite flexibly due to named parameters.

The basic functionality of the algorithms can be highly extended using special purpose map types for their internal data structures. For example, the `Dijkstra`

class stores a `ProcessedMap`, which has to be a writable node map of `bool` value type. The assigned value of each node is set to `true` when the node is processed, i.e., its actual distance is found. Applying a special map, `LoggerBoolMap`, the processed order of the nodes can easily be stored in a standard container.

```
vector<ListDigraph::Node> process_order;
dijkstra(g, length)
  .processedMap(loggerBoolMap(back_inserter(process_order)))
  .run(s);
```

BGL applies another approach, it implements several algorithms with visitor based interfaces. The visitor classes are the generalizations of function objects, they have more entry points by defining several callback member functions. A visitor based algorithm emits different events during the execution and calls the corresponding entry functions of the visitor. In some cases, this solution could be more convenient than the use of customized maps because all event handler operations are specified in the same class. Therefore, LEMON also contains visitor based algorithm classes for BFS and DFS.

LEDA provides much less flexibility in using algorithms. Although they are defined as template functions, only some explicit instantiations are available for the user.

2.5 Graph Adaptors

The LEMON graph adaptor classes serve for considering graphs in different ways. The adaptors can be used exactly the same as "real" graphs (i.e., they conform to the graph concepts), thus all generic algorithms can be performed on them. However, the adaptor classes use the underlying graph structures and operations when their methods are called, thus they have only negligible memory usage and do not perform sophisticated algorithmic actions. This technique yields convenient and elegant tools for the cases when a graph has to be used in a specific alteration, but copying it would be too expensive (in time or in memory usage) compared to the algorithm that should be executed on it.

The following example shows how the `ReverseDigraph` adaptor can be used to run Dijksta's algorithm on the reverse oriented graph. Note that the maps of the original graph can be used in connection with the adaptor, since the node and arc types of the adaptors convert to the original item types.

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

Using `ReverseDigraph` could be as efficient as working with the original graph, but not all adaptors can be so fast, of course. For example, the subgraph adaptors have to access filter maps for the nodes and/or the arcs, thus their iterators are significantly slower than the original iterators. LEMON also provides some more complex adaptors, for instance, `SplitNodes`, which can be used for splitting each node in a directed graph and `ResidualDigraph` for modeling the residual network for flow and matching problems.

BGL also features some graph adaptors, but only a few basic ones, like `reverse_graph` or `filtered_graph`. On the other hand, LEDA cannot provide adaptors due to the closed source distribution.

2.6 LP Interface

Linear programming (LP) is one of the most important general methods of operations research and LP solvers are widely used in optimization software. The interface provided in LEMON makes it possible to specify LP problems using a high-level syntax.

```
Lp lp;

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

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

lp.colLowerBound(x1, 0);
lp.colUpperBound(x2, 100);

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

cout << "Objective function value: " << lp.primal() << endl;
cout << "x1 = " << lp.primal(x1) << endl;
cout << "x2 = " << lp.primal(x2) << endl;
```

`Lp::Col` type represents the variables in the LP problems, while `Lp::Row` represents the constraints. The numerical operators can be used to form expressions from columns and dual expressions from rows. Due to the suitable operator overloads, a problem can be described in C++ conveniently, directly as it is expressed in mathematics. This solution is similar to the ILOG Concert Technology [8].

Note that LEMON does not implement an LP solver, it just wraps various libraries with a uniform high-level interface. Currently, the following linear and mixed integer programming packages are supported: GLPK [11], Clp [5], Cbc [3], ILOG CPLEX [8] and SoPlex [18]. However, additional wrapper classes for new solvers can also be implemented quite easily.

3 Implementation Details

This section presents some interesting implementation details of LEMON, along with specific code examples.

3.1 Extending Graph Interfaces Using Mixins

A fundamental problem of designing a general graph concept is that an easy-to-implement concept should require the least number of overlapping functionality, but this approach strongly limits the flexibility and efficiency of the interface. This contradiction is overcome by developing two-level graph concepts.

In LEMON, the *user-level* graph concepts define a wide range of member functions and nested classes, therefore, they support convenient and flexible use. On the other hand, the *low-level* graph concepts define only the very basic functionality, for example, simplified function-based iteration. These simple interfaces are extended to the user-level concepts using the template *Mixin* strategy [17]. Specifically, if a class `DigraphBase` implements the low-level interface, then `DigraphExtender<DigraphBase>` will fulfill the user-level `Digraph` concept.

```
class DigraphBase {
public:
```

```
  // Node and Arc classes
  class Node { ... };
  class Arc { ... };

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

The extender adds the convenient class-based iterators and the map classes to the graph, as well as the necessary members for alteration observing (see Section 3.2). If the underlying graph class also defines functions for node and arc addition and deletion, then they are overridden to handle the alteration observing, as well.

```
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;
  };
  ...
};
```

3.2 Signaling Graph Alterations

The LEMON graph maps are external, auto-updated structures (see Section 2.3). They are implemented using arrays or `std::vector`s to ensure the efficient data access, which is the most important design goal of maps. However, 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 [10] and they signal the changes of the node and arc sets. The observed events are limited to adding and removing one or several items, building the graph from scratch and removing all items from it. The observers are inherited from the corresponding `AlterationNotifier<Graph, Item>::ObserverBase` class, and they have to override the event handler functions.

The graph maps are constructed to be exception safe, in fact, they guarantee strong exception safety [19]. 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.

3.3 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, for instance, the graphs are marked with `UndirectedTag`.

```
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. On the other hand, an undirected graph is Eulerian if it is connected and the number of incident edges is even for each node.

```
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);
}
```

LEMON uses bool valued tags and `enable_if` borrowed from the Boost libraries [2,12,21] to implement the specializations. This technique allows more options in combination of rules than the simple tag based dispatching.

4 Performance

This section compares the running time performance of LEMON to BGL and LEDA. The experiments were conducted using LEMON 1.2 and Boost 1.41.0, the latest stable releases at the time of writing, and LEDA 5.0. For each library, the most efficient general graph structure was used with the default options and parameters of the algorithms.

Two fundamental problems are considered in the tests: (1) finding shortest paths from a designated source node in a graph with non-negative arc lengths; (2) finding a maximum flow between two nodes in a network with arc capacities.

All test instances were created with NETGEN [14], a popular generator for various network problems. Two different benchmark suites are used. The first one contains sparse graphs, for which m is about $n \log_2 n$, where n and m denote the number of nodes and arcs, respectively. In the second set, there are networks for which m is roughly $n\sqrt{n}$, so they are relatively dense.

The benchmark tests were performed on a 3.2GHz Intel Xeon machine with 2GB RAM and 2MB cache, running openSUSE 11.2 operating system. The codes were compiled with GCC version 4.4.1 using -O3 optimization flag. Each chart shows running times in seconds as a function of the number of nodes in the graph. To obtain suitable diagrams, logarithmic scale is used for both axes.

Figure 2 shows the performance results for finding shortest paths. All the three libraries implement Dijkstra's algorithm for this problem. The differences could mainly be induced by the efficiency of the graph structures and the applied heap representations. BGL was faster than LEDA by a factor between 1.5 and 3.5, but LEMON performed slightly better than BGL on both problem sets.

The benchmark results for the maximum flow problems are presented in Figure 3. Each library provides an implementation of the preflow push-relabel algorithm of Golberg and Tarjan [7] with various heuristics. In these tests, LEDA clearly outperformed BGL, especially on sparse networks, for which it was about 2–3 times faster. However, LEMON was even significantly more efficient than LEDA on all instances.

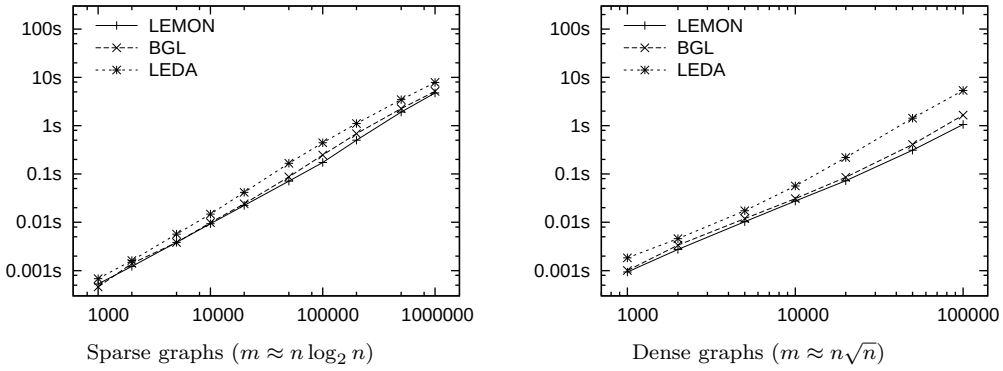

Fig. 2. Benchmark results for the Dijkstra algorithm

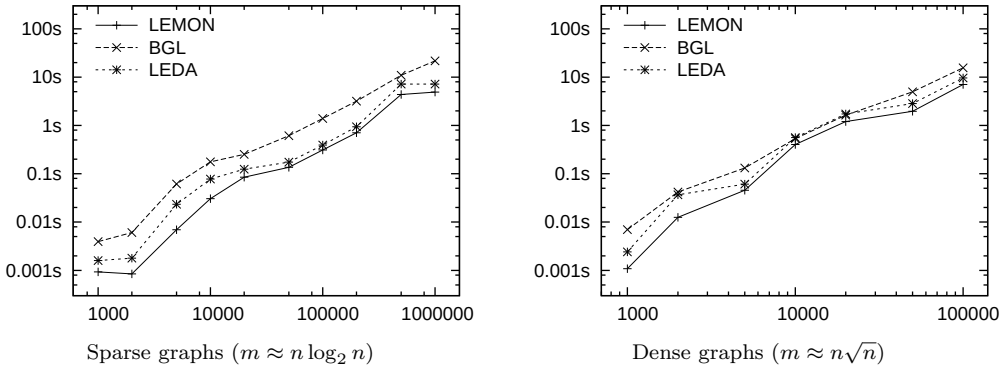

Fig. 3. Benchmark results for maximum flow algorithms

Several other experiments were also made using more algorithms applied to various generated problems and real-life networks, but they are omitted in this paper due to page limit. All comparisons showed similar relations and suggested the same conclusions. Therefore, it seems to be justified to claim that the fundamental algorithms in LEMON are typically more efficient than the corresponding implementations of the other two libraries. This achievement is certainly one of the most important benefits of LEMON, it could be a major reason for using this library.

5 Conclusions

LEMON is a highly efficient, open source C++ graph template library having clear design and convenient interface. It provides a considerable range of data structures, algorithms and other practical components, which can be combined easily for solving problems of various types related to graphs and networks. Comparing to similar libraries, LEMON shows remarkable advantages in versatility, convenience and performance. According to comprehensive benchmark tests, its essential algorithms proved to be significantly more efficient than the corresponding implementations of BGL and LEDA. For these reasons, LEMON is favorable for both research and development in the area of combinatorial optimization and network design.

References

[1] Mihály Bárász, Zsolt Fekete, Alpár Jüttner, Márton Makai, and Jácint Szabó. QoS aware and fair resource allocation scheme in transport networks. In *8th International Conference on Transparent Optical Networks (ICTON '06)*, Nottingham, United Kingdom, June 2006.

[2] Boost C++ Libraries. http://www.boost.org/, 2009.

[3] Cbc – Coin-Or Branch and Cut. http://projects.coin-or.org/Cbc/, 2009.

[4] Tibor Cinkler and László Gyarmati. MPP: Optimal Multi-Path Routing with Protection. In *Proceedings of IEEE International Conference on Communications, ICC 2008*, pages 165–169, Beijing, China, May 2008.

[5] Clp – Coin-Or Linear Programming. http://projects.coin-or.org/Clp/, 2009.

[6] COIN-OR – Computational Infrastructure for Operations Research. http://www.coin-or.org/, 2009.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] ILOG CPLEX. http://www.ilog.com/, 2009.

[9] EGRES – Egerváry Research Group on Combinatorial Optimization. http://www.cs.elte.hu/egres/, 2009.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[11] GLPK – GNU Linear Programming Kit. http://www.gnu.org/software/glpk/, 2009.

[12] Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, pages 25–32, June 2003.

[13] Alpár Jüttner, Tibor Cinkler, and Balázs Dezső. A randomized cost smoothing approach for optical network design. In *9th International Conference on Transparent Optical Networks (ICTON '07)*, volume 1, pages 75–78, Rome, Italy, July 2007.

[14] Darwin Klingman, Albert Napier, and Joel Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science*, 20:814–821, 1974.

[15] LEDA – Library of Efficient Data Types and Algorithms. http://www.algorithmic-solutions.com/, 2009.

[16] LEMON – Library for Efficient Modeling and Optimization in Networks. http://lemon.cs.elte.hu/, 2009.

[17] Yannis Smaragdakis and Don S. Batory. Mixin-based programming in C++. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 163–177, London, UK, 2001. Springer-Verlag.

[18] SoPlex – The Sequential Object-Oriented Simplex. http://soplex.zib.de/, 2009.

[19] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition, February 2000.

[20] Zalán Szűgyi and Zoltán Porkoláb. Quantitative comparison of MC/DC and DC test methods. In Giovanni Falcone, Yann-Gaël Guéhéneuc, Christian Lange, Zoltán Porkoláb, and Houari A. Sahraoui, editors, *12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, QAOOSE Workshop, ECOOP 2008*, pages 1–10, 2008.

[21] István Zólyomi and Zoltán Porkoláb. Towards a general template introspection library. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering, Third International Conference, GPCE 2004*, volume 3286 of *Lecture Notes in Computer Science*, pages 266–282, Vancouver, Canada, October 2004. Springer.