Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 264 (5) (2011) 23–45
www.elsevier.com/locate/entcs

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

Balázs Dezső[a,2] Alpár Jüttner[b,3] Péter Kovács[a,4]

[a] *Department of Algorithms and Applications*
Eötvös Loránd University
H-1117 Budapest, Hungary

[b] *Department of Operations Research*
Eötvös Loránd University
H-1117 Budapest, Hungary

Abstract

This paper introduces LEMON, a generic open source C++ library providing easy-to-use and efficient implementations of graph and network algorithms and related data structures. The basic design concepts, features, and performance of LEMON are compared with similar software packages, namely BGL (Boost Graph Library) and LEDA. LEMON turned out to be a viable alternative to these widely used libraries, and our benchmarks show that it typically outperforms them in efficiency.

Keywords: C++, library, design, graph, network, template

1 Introduction

LEMON [29] is a C++ template library with a focus on combinatorial optimization tasks related mainly to 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 source software project of Egerváry Research Group on Combinatorial Optimization (EGRES) [14] at the Department of Operations Research, Eötvös Loránd University, Budapest. It is also a member of the COIN-OR initiative [9], 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 for research activities.

[1] The LEMON project is supported by EGRES [14].

[2] E-mail: deba@inf.elte.hu

[3] E-mail: alpar@cs.elte.hu

[4] E-mail: kpeter@inf.elte.hu

doi:10.1016/j.entcs.2011.06.003

The goal of the library is to provide highly 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 algorithms (such as graph search, shortest path, spanning tree, matching, and network flow algorithms) and various auxiliary tools (for example, flexible input-output support for graphs and associated data). Furthermore, the library provides a common high-level interface for several linear programming (LP) and mixed integer programming (MIP) [10,12] solvers.

LEMON is designed to be cross-platform and supports a wide range of operating systems and compilers. Up to now, it is tested on Linux, Windows, OSX, and AIX systems with the following compilers: GCC 3.3-4.4, Intel C++, IBM xlC, Visual C++ 2005, 2008, and 2010, MinGW. Due to the CMake [8] based build environment, LEMON integrates well with various IDEs, such as Visual Studio, CodeBlocks or Eclipse.

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. LEMON strives for simpler design and interface besides providing a wider variety of complex algorithms and achieving highest possible overall performance. At present, LEMON is extensively used for research purposes, including network design, traffic routing, and general graph theory [2,6,25,37], as well as in education at Eötvös Loránd University and Budapest University of Technology and Economics. Furthermore, it is also used in commercial applications, for example [13].

Between 2003 and 2007, a series of development versions of LEMON were released with an increasing set of features but without a stable API. Since 2008, stable releases have been developed with version numbers 1.x. They ensure full backward compatibility and feature a smaller but more matured set of tools, which have been improved both in terms of the interface and efficiency. This paper is based on LEMON 1.2, the latest major release at the time of writing.

The rest of this paper is organized as follows. Section 2 provides an overview of the main features of LEMON compared with similar C++ graph libraries. Section 3 describes selected implementation details. Section 4 compares the performance of the discussed libraries by benchmark tests of fundamental algorithms. Section 5 outlines the main further plans for developing LEMON. Finally, the conclusions are drawn in Section 6.

2 Overview

The Boost Graph Library (BGL) is probably the best known C++ graph library, this is why the readers are introduced to LEMON through simple and equivalent sample codes using these two libraries. Both programs construct a directed graph, assign lengths to the arcs and run Dijkstra's algorithm starting from a source node.

Figures 1 and 2 briefly demonstrate the basic tools of a graph library. The

```
typedef adjacency_list<listS, vecS, bidirectionalS,
  no_property, int> graph_t;
graph_t g;

graph_t::vertex_descriptor s = add_vertex(g);
graph_t::vertex_descriptor t = add_vertex(g);
... // add more vertices

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

vector<int> dist(num_vertices(g));
dijkstra_shortest_paths(g, s,
  weight_map(get(edge_bundle, g))
  .distance_map(&dist[0]));

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

Fig. 1. Sample code demonstrating the usage of BGL.

```
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);

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

Fig. 2. Sample code demonstrating the usage of LEMON.

subsequent parts of this section discuss in detail all fundamental features of LEMON and compare the library with its two main competitors, namely BGL [4,32] and LEDA [28,30], in terms of the user interface, the main concepts, and design decisions. Note that BGL is an open source software, while LEDA is a commercial library.

2.1 Graph Data Structures

Although LEMON is a generic library, its main graph types 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 data structures (see Section 2.3).

The example in Figure 2 uses `ListDigraph`, which is a general directed graph implementation based on doubly-linked adjacency lists. Another important digraph type is `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 (see Section 3.1). Therefore, it has smaller memory footprint than `ListDigraph` and 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 data structures.

LEMON follow the generic programming paradigm, as BGL and LEDA do, and describes the requirements of generic components by means of *concepts*. These concepts play the same role as in STL: they define the supported functionality of data types, along with their user interfaces and semantics.

LEMON defines two graph concepts, `Digraph` and `Graph`, which describe the requirements for directed and undirected graphs, respectively. The undirected `Graph`

concept is designed to also satisfy the requirements of the `Digraph` concept in such a way that each *edge* of an undirected graph can also be viewed as a pair of oppositely directed *arcs*. Therefore, each undirected graph, without any transformation, can be considered as a directed graph at once.

The main benefit of this design is that all directed graph algorithms automatically work for undirected graphs, as well. In most cases, this also means that there is no need for separate algorithm implementations. However, particular algorithms could require specialization for undirected graphs. Such a special method is the checking of the Eulerian property, which is discussed in detail in Section 3.4.

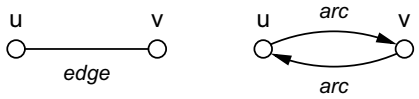

Fig. 3. Illustration of the undirected graph concept of LEMON. Each undirected edge can also be viewed as two oppositely directed arcs.

Undirected graphs provide an `Edge` type for the undirected edges and an `Arc` type for the arcs. This separation makes the implementation of some algorithms simpler (e.g., planar graph algorithms) because we can distinguish the undirected edges from their directed variants. On the other hand, the `Arc` type of an undirected graph is convertible to the `Edge` type, thus the corresponding edge of an arc can always be obtained conveniently, without calling any functions. As a result, all methods and data structures that are designed for edges can be used directly with both edges and arcs. This could be quite practical in several cases. For example, a property map (see Section 2.3) that assigns data to edges can be used with both edges and arcs, but an arc map can only be used with arcs.

BGL implements a single adjacency list based graph class, but it can be fully customized with template parameters that specify the internal storage data structures for nodes and arcs. Furthermore, a graph can be set to *directed*, *bidirectional* or *undirected* using another template parameter. The bidirectional graph concept is the equivalent of LEMON's `Digraph` concept. These graph types support traversing through the outgoing and incoming arcs of each node. The directed graphs of BGL store only the outgoing arc lists, thus they require less storage space than bidirectional graphs. Note that this category is missing from LEMON.

The `adjacency_list` class template of BGL implements both directed and undirected graphs by extensive use of template specializations. As a result, directed and undirected graphs have the same interfaces but different semantics in BGL. The edges of undirected graphs are usually considered undirected, but they have directions in some cases, for example, in iterations. Such an inconsistency could be confusing. Moreover, this design does not make it possible to define property maps whose keys are the directed variants of the edges, although it would also be important in certain algorithms.

LEDA's general `graph` class has closed source, but its implementation is probably similar to the general graph types of LEMON. The main difference is that LEDA implements directed and undirected graphs in the same class and provides member functions to switch between the two modes. This design is certainly convenient in

some cases, but it is less distinctive than LEMON's concepts, therefore, it has some disadvantages, similarly to BGL.

Various special purpose graph types are also implemented in the libraries, for example, full graphs, grid graphs or adjacency matrix graphs. Furthermore, all of the three libraries provide an optimized static data structure for directed graphs, which stores the nodes and arcs in arrays or vectors in such a way that the arcs are sorted by their source nodes. As the crucial operations of most directed graph algorithms iterate on the outgoing arcs of the nodes, they typically run faster using these static implementations.

2.2 Iterators

Most graph libraries provide iterator classes for traversing through the elements of the graph data structures (i.e., the nodes and arcs). LEMON defines a special iterator interface, which does not conform to the iterator concepts of the C++ Standard Template Library (STL).

The iterators of LEMON are initialized to the first element in the traversed range by their constructors, and their validity is checked by comparing them to a special constant `INVALID`. Furthermore, each iterator class is convertible to the corresponding graph element type, without having to use `operator*()`. This feature distinguishes LEMON iterators from the standard C++ iterators and makes their usage slightly simpler.

Recall the example shown in Figure 2. The computed distance of each node can be printed to the standard output as follows.

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

In the first line, all occurrences of `v` refer to the iterator itself, while the corresponding node object is referred twice inside the loop.

Note that this concept could not be applied to general iterators. For example, STL defines iterators for containers of arbitrary items. It means that the iterator type and the item type of the container could have conflicting functionality, for instance, both of them could support `operator++()`. Therefore, an iterator object and the referred object must be distinguished: `it++` affects the iterator `it`, while `(*it)++` affects the referred object `*it`.

LEMON iterator concepts, however, exploit the speciality of graphs, which can be viewed as containers of particular elements. The nodes and arcs themselves provide a strongly limited set of features, which does not conflict with the functionality of iterators. Therefore, the program context always indicates whether we refer to an iterator or to a graph element, as we have already seen in the above example.

In contrast with this, BGL iterators follow the STL requirements of input iterators. It means that they must be dereferenced with the `operator*()` function to obtain the corresponding item descriptors. Recall the BGL code shown in Figure 1. After running Dijkstra's algorithm, the node distances can be printed as follows.

```
graph_t::vertex_iterator vi, vend;
for (tie(vi, vend) = vertices(g); vi != vend; ++vi) {
```

```
  std::cout << *vi << ": " << dist[*vi] << std::endl;
}
```

The `tie()` function is used to make the code more compact and to avoid simple mistakes of the programmer.

A drawback of the above solution is that the iterator objects are defined in a wider scope than the loop itself. BGL, however, also provides several iteration macros that simplify traversing graph elements and define the loop variables only in the scope of the loop.

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

Similar macros are available in LEDA, but they do not allow to define the loop variables only in the scope of the loop.

```
node v;

forall_nodes(v, g) {
  g.printNode(v);
  std::cout << ": " << dist[v] << std::endl;
}
```

2.3 Handling Graph Related Data

In addition to the pure graph data structures, most graph algorithms need additional data associated to the nodes and arcs. For example, shortest path algorithms require a length function on the arcs and record 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 a rather slow $O(\log n)$ time access to the elements). Since they are among the most frequently used data structures, maps should be highly efficient and convenient.

The most important operation of a map data structure is the access of its elements, that is, retrieving or overwriting the value assigned to a certain node or arc. In most graph libraries, time complexity of these operations is $O(1)$. Library designers have to deal with two additional performance considerations. First, map access operations should not be virtual functions because that forbids inlining. Second, it is worthwhile to use continuous storage for maps since it usually induces faster data access due to better caching.

LEMON features only external property maps that are stored separately from the related graph data structure, but they are updated automatically on the changes of the graph (see Section 3.3). The main advantage of external maps is their great flexibility. They can be constructed and destructed freely, so their lifetimes are not bound to the lifetime of the graph. Moreover, separate storage could result in better caching properties, especially using several maps for a large graph.

Using LEMON, node and arc maps can be declared as follows.

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

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

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

Besides the standard graph maps, LEMON also contains several "lightweight" *map adaptor* classes. They are not stand-alone maps with own data storage, but they adapt one or more other map objects and alter their data "on the fly". When the access operation of a map adaptor is called, it reads the corresponding data from the underlying maps and performs a certain operation on them, but without actually modifying or copying the original storage. These adaptor 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 object with two arc maps `length` and `speed`, which store for each arc the physical length of the corresponding road section and the maximum (or average) speed that can be achieved on it, respectively. If we are interested in the optimal traveling times, then we can call Dijkstra's algorithm as follows.

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

The `divMap()` function gives back a map adaptor object that provides the quotient of the values of the two original maps. It means that the Dijkstra algorithm receives for each arc the expressed traveling time of the corresponding road section.

Contrary to LEMON, several libraries store the associated data directly in the node and arc objects of the graphs. For example, only a limited number of internal maps of fixed types can be used in the Stanford GraphBase library [27,31]. This design allows easier implementation but strongly limits the versatility of the library.

BGL supports both internal and external storage of graph related data. The interior properties of nodes and edges can be specified as *bundled properties* or *property lists*. The bundled properties provide a much simpler interface and their use is to be preferred, whereas the latter solution is compatible with older compilers and older versions of the Boost library. Figure 1 shows a simple example for the usage of bundled properties (the lengths of the edges). If more assigned values are required for the nodes and edges, they have to be collected into specific data types, which are then passed as template parameters to the graph class.

```
struct NodeData { ... };
struct EdgeData { ... };

typedef adjacency_list<listS, vecS, bidirectionalS,
  NodeData, EdgeData> GraphType;
```

The main advantage of internal storage is that its capacity is adjusted automatically if the graph is modified, but it is not flexible as its lifetime is strictly bound to the graph object.

External property maps are also supported in BGL by wrapping standard container data structures. They are more flexible than interior properties since their lifetimes are not bound to the associated graph. However, we have to choose between efficiency and convenience if we use these maps in conjunction with a varying graph. We can apply a map that wraps a random access container (e.g., `std::vector`) to ensure rapid data access, but it must be updated manually each time the graph changes. Alternatively, we can also use an external map that is based on an as-

sociative container (e.g., `std::map`). This solution naturally adapts to any change of the graph without explicit updating, but at a significant expense of efficiency. Note that LEMON's graph maps, however, provide this flexibility and convenience without the expense of performance (see Section 3.3).

LEDA implements two kinds of external data structures for handling graph related data. The *arrays* are static data structures, but their access operations take constant time. The *map* types are more flexible as they are not invalidated when the associated graph is changed. However, they are implemented by hash tables, and so they are less efficient. Therefore, we encounter the same trade-off as with the external maps of BGL.

Although these data structures are external, LEDA makes it possible to allocate additional storage space for them in the graph objects. The newly created arrays and maps can be assigned to these slots, so the memory usage can be optimized. Apart from these solutions, LEDA also provides parameterized graph data structures, whose node and edge objects can contain arbitrary additional data, just like the bundled properties in BGL.

2.4 Algorithms

Inevitably, the most important differentiating factor between graph libraries is the range and quality of the implemented algorithms. A simple graph data structure for a specific use can be implemented rapidly, but sophisticated algorithms need careful design and lots of work from skilled programmers, especially when the efficiency is of high priority.

LEMON provides highly efficient implementations of numerous algorithms related to graph theory and combinatorial optimization. These algorithms include fundamental methods, such as breadth-first search (BFS), depth-first search (DFS), Dijkstra algorithm, Bellman-Ford algorithm, Kruskal algorithm, and methods for discovering various graph properties (connectivity, bipartiteness, Eulerian property, etc.), as well as complex algorithms for finding maximum flows, minimum cuts, feasible circulations, maximum matchings, minimum mean cycles, minimum cost flows, and planar embedding of a graph. BGL and LEDA feature similar varieties of algorithms but with different interfaces.

In LEMON, algorithms are implemented as class templates, but for the sake of convenience, function-type interfaces are also available for some of them. For instance, Dijkstra's algorithm is implemented in the `Dijkstra` class, but a `dijkstra()` function is also defined, which was used in the former examples.

The function interfaces of the algorithms are considerably simpler, but they are suitable for most practical cases due to the extensively used *named parameter* technique. This technique supports several function parameters with default values and an arbitrary set of these parameters can be specified in an arbitrary order by calling a dedicated function for each desired parameter. It means that the parameters are referred by names instead of the standard position-based reference. LEMON implements named parameters quite similarly to the Boost library [32].

The sample code in Figure 2 could also use the class interface as follows.

```
Dijkstra<ListDigraph> alg(g, length);
alg.distMap(dist);
alg.run(s);
```

This code is longer than the former one, but the execution can be controlled to a higher extent using this interface. For example, more source nodes can be specified and the algorithm can also be executed step-by-step, as the following code demonstrates.

```
alg.init();
alg.addSource(s);
while (!alg.emptyQueue()) {
  ListDigraph::Node v = alg.processNextNode();
  std::cout << g.id(v) << ": " << alg.dist(v) << std::endl;
}
```

The basic functionality of the algorithms can be greatly extended using special purpose map types for their internal data structures. For example, the `Dijkstra` class stores a `ProcessedMap`, which should be a writable node map of `bool` value type. The assigned value of a node is set to `true` when the node is processed, that is, its actual distance is found. Applying a special map, `LoggerBoolMap`, the processing order of the nodes can be recorded easily in a standard container.

Such specific map types can be passed to the algorithms using the technique of *named template parameters*. Similarly to the named function parameters, they allow specifying any subset of the parameters in arbitrary order.

```
typedef vector<ListDigraph::Node> Container;
typedef back_insert_iterator<Container> InsIterator;
typedef LoggerBoolMap<InsIterator> MyProcessedMap;

Container container;
InsIterator iterator(container);
MyProcessedMap map(iterator);
Dijkstra<ListDigraph>
  ::SetProcessedMap<MyProcessedMap>
  ::Create alg(g, length);

alg.processedMap(map);
alg.run(s);
```

Surprisingly, even the above example can be implemented using the `dijkstra()` function and named parameters as follows.

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

Note that a function interface has the major advantage that temporary objects can be passed as reference parameters. In this example, both the insert iterator object and the map object are created only temporarily.

BGL implements several algorithms with *visitor-based* interfaces instead of using special purpose graph maps. The visitor classes are the generalizations of function objects: they have more entry points by defining several callback functions. A visitor-based algorithm emits different events during its execution and calls the corresponding entry functions of the associated visitor. In some cases, this technique could be more convenient than the use of customized maps, because all event handler operations are implemented in the same class. For this reason, LEMON also provides visitor-based solutions but only for the basic graph search algorithms, BFS and DFS.

LEDA provides less flexibility in using algorithms than the other two libraries. It implements a few compact function interfaces for each algorithm but without named parameters. These functions are designed for the most typical use cases and support only a limited set of configuration options.

2.5 Graph Adaptors

In typical graph algorithms and applications, we usually require a specific alteration of a graph. For example, certain nodes or arcs should be removed or the reverse oriented graph should be used. However, the actual modification of the physical storage or making a copy of the data structure along with the required maps could be rather expensive (in time or in memory usage) compared to the operations that should be performed on the altered graph. In such cases, LEMON's graph adaptor classes can be used.

Graph adaptors are special class templates that serve for considering other graph data structures in different ways. They are based on the same idea as the previously discussed map adaptors (see Section 2.3), but they are more complex. Graph adaptors can only be used in conjunction with another graph object that provides an actual storage of a graph. They do not modify the underlying data structure, they just give another view of it by utilizing the original operations. Graph adaptors conform to the graph concepts, thus they can be used the same as "real" graphs, and all generic algorithms works for them.

The following example shows how the `ReverseDigraph` adaptor can be used to run Dijkstra's algorithm on the reverse oriented graph.

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

Note that the maps of the original graph (`length` and `dist`) can also be used with the adaptor, since the node and arc types of all adaptors convert to the original item types.

As this example slightly demonstrates, graph adaptors help writing compact and elegant code and make it easier to implement complex algorithms based on reliable standard components.

Another fundamental graph alteration is the hiding of nodes and arcs, which can be achieved using one of the subgraph adaptors in LEMON. These classes store filter maps that are used by the iterators to skip the currently hidden items. Therefore, subgraph adaptors are significantly less efficient than the original graph objects.

As the adaptor classes conform to the graph concepts, we can even apply an adaptor to another one. Figure 4 illustrates a situation when a `SubDigraph` adaptor is applied to a directed graph and `Undirector` is used to make the obtained subgraph undirected.

Combinatorial optimization methods are usually based on more complex graph alterations. For example, the residual network is a particularly important model for flow and matching algorithms. `ResidualDigraph` implements this network by adapting a directed graph along with a capacity map and a flow map.

`SplitNodes` is another practical adaptor that splits each node into an in-node

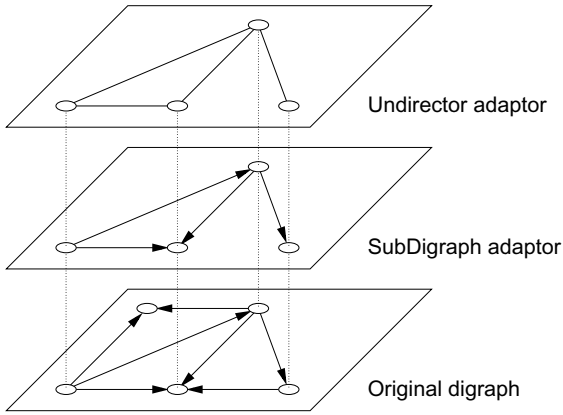

Fig. 4. Illustration of graph adaptors in LEMON.

and an out-node in a directed graph. Formally, the adaptor replaces each node v with two nodes v_{in} and v_{out}. Each arc (u, v) of the original graph will correspond to an arc (u_{out}, v_{in}). The adaptor also adds an additional bind arc (v_{in}, v_{out}) for each node v of the original graph. The aim of this construction is to assign costs or capacities to the nodes of the graph when using algorithms which would otherwise consider only arc costs or capacities.

BGL also features graph adaptors but only a few basic ones, like `reverse_graph` or `filtered_graph`. On the other hand, LEDA does not provide similar tools.

2.6 LP Interface

Linear programming (LP) is one of the most important general methods of operations research. Countless optimization problems can be formulated and solved using LP techniques. Nowadays, various efficient LP solvers are available, including both open source and commercial software. Therefore, LEMON does not implement its own solver but features wrapper classes for several LP libraries providing a common high-level interface for them.

The advantage of this design is twofold. First, LEMON applies an object oriented approach, which is quite similar to the ILOG Concert Technology [11]. This approach makes LEMON's interface more flexible than the native interfaces of several LP libraries and it could be more comfortable for those who are familiar with object oriented programming. Second, changing the underlying solver in an application that uses this common syntax needs no effort. Therefore, one can easily experiment various LP solvers in her particular application and compare their efficiency at any stage of the development.

Figure 5 demonstrates how simple it is to formalize and solve an LP problem in LEMON. `Lp::Col` represents the variables of the LP problems, while `Lp::Row` represents the constraints. The numerical operators are used to form expressions from columns and dual expressions from rows. Due to the suitable operator overloads, an LP problem can be described conveniently, directly as it is expressed in mathematics.

```
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.colUpperBound(x2, 10);

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;
```

$$
\begin{aligned}
\text{maximize} \quad & 10x_1 + 6x_2 \\
\text{subject to} \quad & 0 \leq x_1 + x_2 \leq 100 \\
& 2x_1 \leq x_2 + 32 \\
& x_1 \geq 0 \\
& x_2 \leq 10
\end{aligned}
$$

Fig. 5. Sample code demonstrating the usage of the LP interface of LEMON.

The LP solvers are powerful general tools for solving various complex optimization problems. Let us consider the well-known maximum flow problem for example. It is to find a flow of maximum value between a source and a target node in a network with capacity constraints. Let $G = (V, A)$ denote a digraph, let $c : A \to \mathbb{R}^+$ denote a capacity function and let $s, t \in V$ denote the source and target nodes, respectively. A maximum flow is an $f : A \to \mathbb{R}$ solution of the following optimization problem.

$$
\begin{aligned}
\text{maximize} \quad & \sum_{v\,:\,(s,v)\in A} f(s,v) - \sum_{v\,:\,(v,s)\in A} f(v,s) & \\
\text{subject to} \quad & \sum_{v\,:\,(u,v)\in A} f(u,v) = \sum_{v\,:\,(v,u)\in A} f(v,u) & \forall u \in V \setminus \{s,t\} \\
& 0 \leq f(u,v) \leq c(u,v) & \forall (u,v) \in A
\end{aligned}
$$

The sample code in Figure 6 solves this problem using the LP interface of LEMON. Note that the expressions are built using simple loops that traverse the outgoing and incoming arcs of nodes. Various other graph optimization problems can also be expressed as linear programs and the interface provided in LEMON facilitates solving them easily (though usually not so efficiently as by a direct combinatorial method, if one exists).

Currently, the following linear and mixed integer programming packages are supported by LEMON: GLPK [16], Clp [7], Cbc [5], ILOG CPLEX [11], and SoPlex [34]. Additional wrapper classes for new solvers can also be implemented quite easily.

2.7 Input-Output Handling

LEMON provides a general file format for storing graphs and related node and arc maps. Such a format should be versatile, that is, it should support storing arbitrary number of maps of arbitrary value types. Furthermore, the file size and the ease of processing are also crucial to support working with huge graphs, which is a major goal of LEMON. Therefore, a flat text file format was designed instead of using structured hierarchical formats, such as GraphML [21], GXL [22] or GML [17].

```
Lp lp;
GR::ArcMap<Lp::Col> f(g);
lp.addColSet(f);

// Objective function
Lp::Expr obj;
for (GR::OutArcIt a(g, src); a != INVALID; ++a) obj += f[a];
for (GR::InArcIt a(g, src); a != INVALID; ++a)  obj -= f[a];
lp.max();
lp.obj(obj);

// Flow conservation constraints
for (GR::NodeIt v(g); v != INVALID; ++v) {
  if (v == s || v == t) continue;
  Lp::Expr expr;
  for (GR::OutArcIt a(g, v); a != INVALID; ++a) expr += f[a];
  for (GR::InArcIt a(g, v); a != INVALID; ++a)  expr -= f[a];
  lp.addRow(expr == 0);
}

// Capacity constraints
for (GR::ArcIt a(g); a != INVALID; ++a) {
  lp.colLowerBound(f[a], 0);
  lp.colUpperBound(f[a], c[a]);
}

// Solve LP
lp.solve();
```

Fig. 6. Sample code for solving the maximum flow problem with the LP interface of LEMON.

The LEMON Graph Format (LGF) comprises different sections, for example, a digraph is stored in a `@nodes` and an `@arcs` section. These parts use column oriented formats, in which each column belongs to a map in the graph. The first lines of the sections associate names to these maps, which can be used to refer them. Note that this simple idea makes it possible to extend the files with new maps (columns) at any position without having to modify the processing codes.

The `label` maps play a special role, they must store unique values, which in turn can be used to refer to the nodes and arcs in the file. The first two columns of the `@arcs` section are anonymous, they indicate the source and target nodes, respectively.

```
@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
target 41
caption "A shortest path problem"
```

This LGF file can be processed using the `digraphReader()` function with several named parameters as follows.

```
ListDigraph g;
ListDigraph::NodeMap<dim2::Point<int> > coord(g);
ListDigraph::ArcMap<int> length(g);
ListDigraph::Node src;
std::string title;

digraphReader(g, "input.lgf")
  .nodeMap("coord", coord)
  .arcMap("length", length)
```

```
.attribute("caption", title)
.node("source", src)
.run();
```

3 Implementation Details

This section presents selected implementation details of LEMON along with specific code examples that demonstrate the applied techniques.

3.1 Adjacency Lists in Vectors

The general graph types of LEMON store the adjacency lists internally in `std::vector`s and use the vector indices as identifiers of the nodes and arcs. `Node` and `Arc` objects store these indices, thus for each of them, the corresponding vector element can be looked up in constant time.

For example, the following code fragment comes from the source of `SmartDigraph`.

```
struct NodeT {
  int first_in;        // index of the first incoming arc
  int first_out;       // index of the first outgoing arc
};
struct ArcT {
  int target, source;  // indices of the endnodes
  int next_in;         // index of the next incoming arc
  int next_out;        // index of the next outgoing arc
};
std::vector<NodeT> nodes;
std::vector<ArcT> arcs;
```

The iteration on the outgoing arcs of a given node begins with the lookup of the corresponding `NodeT` item, whose `first_out` member stores the index of the first arc. After that, each step reads the `next_out` value from the current `ArcT` object to obtain the index of the next arc, or `-1` if the current arc is the last one. The incoming arcs are handled in the same way using the members `first_in` and `next_in`. It means that the incident arcs are recorded using simply-linked lists that are actually stored in a vector.

`ListDigraph` is implemented similarly, but it maintains the nodes and arcs using doubly-linked lists to support the efficient deletion of them, as well as the addition of new elements.

A major advantage of these data structures is that each node and arc is associated with a unique integer identifier, which is a crucial requirement for implementing efficient external maps (see Section 2.3). On the other hand, this design restricts the customization possibilities of the graph data structures, because the nodes and arcs must be stored in random access containers.

BGL applies another approach: its main graph type, the `adjacency_list` class is highly customizable. Container types can be specified by template arguments separately for the node list and the incident edge lists of the nodes. Using advanced data structures for these purposes can be beneficial in certain cases. For example, storing the incident edges in an `std::set` allows logarithmic time lookup of an outgoing edge with a given target node. However, the lack of naturally assigned unique integer identifiers makes it harder to use external maps, which are frequently

required in graph algorithms for temporary storage.

3.2 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 versatility 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, and so 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 [33]. Specifically, if a class `DigraphBase` implements the low-level interface, then `DigraphExtender<DigraphBase>` will satisfy the requirements of 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 nested iterator and map classes to the graph type, as well as the required member variables for alteration observing (see Section 3.3). If the underlying graph class also defines functions for node and arc addition or deletion, then they are overridden to handle the alteration observing, as well.

```
template <typename DigraphBase>
class DigraphExtender : public DigraphBase {
public:
  // Iterator class
  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;
  };
  ...
};
```

3.3 Signaling Graph Alterations

Recall from Section 2.3 that LEMON graph maps are external, auto-updated data structures. They are implemented using arrays or `std::vector`s to ensure efficient data access, which is the most important design goal of maps. However, these data structures are extended when new nodes or arcs are added to the associated graph.

The graph and map types implement the *Observer* design pattern [15], 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.

Graphs contain instances of `AlterationNotifier<C, I>` for each item type.

```
class Graph {
  ...
protected:
  AlterationNotifier<Graph, Node> _node_notifier;
  AlterationNotifier<Graph, Arc> _arc_notifier;
  AlterationNotifier<Graph, Edge> _edge_notifier;
};
```

The graph maps are designed to be exception safe. In fact, they guarantee strong exception safety [35]. 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.

3.4 Tags and Specializations

The functionality and efficiency 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;
  ...
};
```

Let us consider the checking of the Eulerian property for example. 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. Therefore, the `eulerian()` function is specialized for undirected graphs using `UndirectedTag` as follows.

```
template <typename GR>
typename enable_if<typename GR::UndirectedTag, bool>::type
eulerian(const GR &g) {
  for (typename GR::NodeIt v(g); v != INVALID; ++v) {
    if (countIncEdges(g, v) % 2 == 1) return false;
  }
  return connected(g);
}
```

LEMON uses bool-valued tags and `enable_if` borrowed from the Boost libraries [4,24,38] to implement the specializations. This technique allows more options in combination of rules than the simple tag-based dispatching.

Another example for specialization can be found in the implementation of graph maps. Because the data vectors of `ListDigraph` and the corresponding maps could contain gaps, some items do not need to be constructed. To avoid the unnecessary data initializations and potential side effects, the values of the maps are constructed

with placement `new` when items are inserted into the graph. However, maps of POD value types are implemented with `std::vector`s because their constructors are cheap and do not have side effects. The values are reset when the items are removed from the graph.

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 1.43.0, the latest stable releases at the time of writing, and LEDA 5.0. The latter one is not the latest, but note that LEDA is not a free software.

Three fundamental problems were considered in the tests [1,10]: (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; (3) finding a minimum cost flow from a set of supply nodes to a set of demand nodes in a network with capacity constraints and arc costs.

All test instances were created with NETGEN [26], a popular generator for various network problems. Two different benchmark suites are considered. 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 arc capacities and costs were generated evenly from the range [1..10000]. In the minimum cost flow instances, the number of supply and demand nodes are both about $\sqrt{n}$. The largest sparse network contains one million node and about 20 million arcs, and the largest dense graph contains one hundred thousand nodes and more than 30 million arcs. Storing them takes 600–800 MB space in DIMACS format, a widely used compact text file format.

The benchmark tests were performed on a machine with AMD Opteron Dual Core 2.2 GHz CPU and 16 GB memory (1 MB cache), running openSUSE 10.1 operating system. The codes were compiled with GCC version 4.1.0 using -O3 optimization flag.

The charts in the following figures show the measured running times in seconds as a function of the number of nodes in the graph. Logarithmic scale is used for both axes to ensure suitable diagrams. In these experiments, the most efficient general graph data structure was used for each library. Namely, `SmartDigraph` was used for LEMON, `adjacency_list<vecS, vecS, directedS, ...>` was used for BGL and `graph` was used for LEDA.

Figure 7 shows the benchmark results for finding shortest paths. All the three libraries implement Dijkstra's algorithm for this problem with several priority queue representations. To obtain comparable results, the standard binary heap data structure was selected for each library. BGL was more efficient than LEDA, especially on large dense graphs, for which it turned out to be more than two times faster. However, LEMON performed significantly better than both of them on all problem instances. Since Dijkstra's algorithm is rather simple, these differences were obvi-

ously induced by the efficiency of the applied graph, map, and heap data structures.

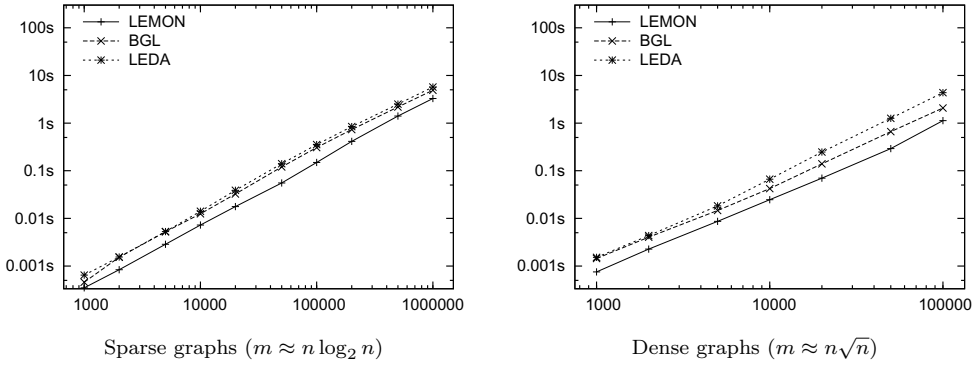

Fig. 7. Benchmark results for the Dijkstra algorithm.

The performance results for the maximum flow problem instances are presented in Figure 8. Each library provides an implementation of the *push-relabel algorithm* of Goldberg and Tarjan with various heuristics [19]. This algorithm is one of the fastest solution methods, but its practical efficiency highly depends on the applied heuristics, and the three implementations differ in this aspect. In these tests, LEDA clearly outperformed BGL, but LEMON turned out to be even more efficient than LEDA. It was about two times faster on almost all instances.

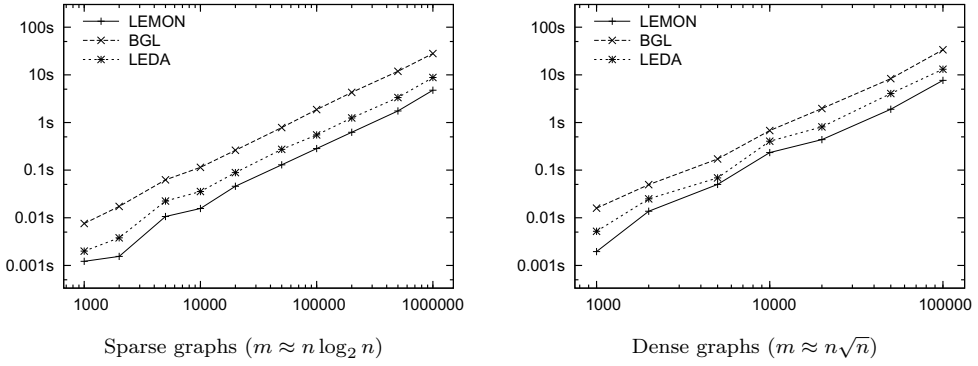

Fig. 8. Benchmark results for maximum flow algorithms.

Figure 9 shows the results for the minimum cost flow algorithms. In this case, only LEMON and LEDA could be compared because BGL does not implement a solution method for this problem, though it has been among the plans of the developers for a long time.

LEMON features various algorithms for the minimum cost flow problem. The two most efficient methods are the *cost scaling algorithm* [18,20] and the *network simplex algorithm* [1,12]. As LEDA also implements the cost scaling algorithm, the same method was chosen for LEMON. The efficiency of this algorithm also depends on the application of various practical heuristics, in which the libraries differ. According to these tests, LEDA was slower than LEMON by a factor between 1.7 and 2.1 except for the small instances. Moreover, it failed with *"cost overflow"*

error message on the largest two sparse networks, thus running time data is omitted for them.

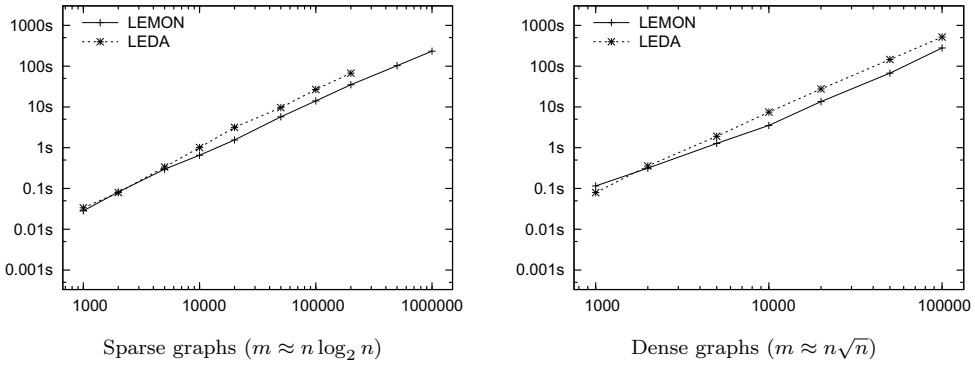

Fig. 9. Benchmark results for minimum cost flow algorithms.

Since LEMON and BGL are generic and open source libraries, we could implement graph adaptor classes that make it possible to run LEMON algorithms on BGL graph data structures and BGL algorithms on LEMON graph data structures. Table 1 contains benchmark results of such comparisons. The performance of the Dijkstra algorithm is measured on the largest problem instances for all combinations of the LEMON and BGL implementations (`SmartDigraph` and `adjacency_list<vecS, vecS, directedS, ...>` were used as before). The binary heap data structures were considered as parts of the algorithm implementations. However, the property maps are strongly related to the graph data structures, thus they were exchanged together with the graphs. Note that the differences in the design decisions of the libraries could have a huge effect on the performance of fundamental data structures (see Sections 2.1, 2.3, 3.1).

Graph type	Algorithm	Sparse graph	Dense graph
LEMON	LEMON	3.27s	1.13s
LEMON	BGL	4.36s	1.07s
BGL	LEMON	3.55s	1.56s
BGL	BGL	4.90s	2.08s

Table 1
Benchmark results for the largest instances of the shortest path problem combining LEMON and BGL implementations.

These results verify that LEMON's `SmartDigraph` implementation is significantly faster than the `adjacency_list` data structure of BGL. Moreover, the Dijkstra algorithm of LEMON also proved to be more efficient, probably because of the better implementation of the heap data structure. The BGL graph type with the BGL algorithm implementation was clearly the slowest combination.

Apart from the general graph types, all the three libraries provide more efficient static graph implementations, which were also tested. Table 2 compares the performance of Dijkstra's algorithm using the general and static graph types of the

libraries. The main conclusion of these results is that LEMON's `SmartDigraph` implementation was almost as efficient as `StaticDigraph`, while the general graph types of the other two libraries turned out to be much slower than the static representations, especially when relatively dense graphs are considered. We can also note that the differences between the libraries were smaller using the optimized graph representations. For dense graphs, the running times were practically the same.

Implementation	Sparse graph	Dense graph
LEMON with `SmartDigraph`	3.27s	1.13s
LEMON with `StaticDigraph`	3.26s	0.94s
BGL with `adjacency_list`	4.90s	2.08s
BGL with `compressed_sparse_row_graph`	4.39s	0.96s
LEDA with `graph`	5.71s	4.36s
LEDA with `static_graph`	4.52s	0.96s

Table 2
Benchmark results for the largest instances of the shortest path problem using general and static graph types.

This comparison fairly demonstrates the importance of efficient graph data structures and their effect on the overall performance of algorithms. Although the static graph types are clearly more efficient, the performance of general graph types is also important because they are used more frequently.

Numerous other experiments were also made using several compilers and 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. The fundamental algorithms and data structures of LEMON turned out to be measurably faster than the corresponding implementations of the other two libraries. This achievement is clearly one of the most important benefits of LEMON. It could be a major reason for using this library.

5 Future of LEMON

A major goal of the upcoming release LEMON 1.3 is to provide basic multi-threading support by allowing parallel execution of several algorithms on the same graph object. Following releases will also implement internally parallel graph algorithms.

Along with this, work will be continued on porting and thoroughly revising all the features that exist in the 0.x series of LEMON. An important group still waiting for porting is the bipartite graph concepts, implementations and bipartite graph related algorithms. This task is planned to be accomplished by the release of version 1.3.

Furthermore, entirely new features are also expected in the upcoming new releases, such as heuristic and approximation algorithms for hard optimization problems including, for example, the traveling salesman problem and the maximum

clique problem.

5.1 Adopting the New C++ Standard

The planned new C++ standard, unofficially called *C++0x* will contain several language improvements [3,36]. LEMON will be adjusted to exploit the benefit of the new constructs.

In the new standard, the *const lvalue* and the *rvalue* references can be distinguished [23], which is useful in many practical cases. For example, the following code is syntactically right, but it could fail with runtime error.

```
DigraphWriter writer(g, std::cout);
writer.nodeMap("map", shiftMap(map, 42));
writer.run();
```

The `shiftMap()` function creates only a temporary variable, but the `DigraphWriter` class stores a reference to this object. When this referenced object is used in the `run()` function, it could already be destroyed.

The new language feature makes it possible to decide in compile time whether the parameter is a temporary object, thus a compilation error could be enforced in such cases. Moreover, this feature also allows a smarter handling of map references. For example, the function `nodeMap()` could be specialized for temporarily created parameters. This version of the function would store the passed object instead of setting a reference. This solution would support a more flexible usage without significant performance loss.

6 Conclusions

LEMON is a highly efficient, open source C++ graph template library having a clear design and convenient interface. It provides a wide 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. According to extensive benchmark tests, essential algorithms and data structures of LEMON typically turned out to be more efficient than the corresponding tools of widely used similar libraries, namely 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] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., February 1993.

[2] M. Bárász, Z. Fekete, A. Jüttner, M. Makai, and J. 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.

[3] P. Becker. Working draft, standard for programming language C++. Technical Report N3090=10-0080, ISO/IEC, March 2010.

[4] Boost C++ Libraries. http://www.boost.org/, 2010.

[5] Cbc – Coin-Or Branch and Cut. http://projects.coin-or.org/Cbc/, 2010.

[6] T. Cinkler and L. 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.

[7] Clp – Coin-Or Linear Programming. http://projects.coin-or.org/Clp/, 2010.

[8] CMake – Cross Platform Make. http://www.cmake.org/, 2010.

[9] COIN-OR – Computational Infrastructure for Operations Research. http://www.coin-or.org/, 2010.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[11] ILOG CPLEX. http://www.ilog.com/, 2010.

[12] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[13] B. Dezső, A. Jüttner, and P. Kovács. Column generation method for an agent scheduling problem. *Electronic Notes in Discrete Mathematics*, 36(3):829–836, 2010.

[14] EGRES – Egerváry Research Group on Combinatorial Optimization. http://www.cs.elte.hu/egres/, 2010.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[16] GLPK – GNU Linear Programming Kit. http://www.gnu.org/software/glpk/, 2010.

[17] The GML File Format. http://www.infosun.fim.uni-passau.de/Graphlet/GML/, 2010.

[18] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22(1):1–29, 1997.

[19] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[20] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[21] The GraphML File Format. http://graphml.graphdrawing.org/, 2010.

[22] GXL – Graph eXchange Language. http://www.gupro.de/GXL/, 2010.

[23] H. E. Hinnant, B. Stroustrup, and B. Kozicki. A brief introduction to rvalue references. Technical Report N2027=06-0097, ISO JTC1/SC22/WG21 – C++ working group, 2006.

[24] J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, pages 25–32, June 2003.

[25] A. Jüttner, T. Cinkler, and B. 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.

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

[27] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York, NY, USA, 1993.

[28] LEDA – Library of Efficient Data Types and Algorithms. http://www.algorithmic-solutions.com/, 2010.

[29] LEMON – Library for Efficient Modeling and Optimization in Networks. http://lemon.cs.elte.hu/, 2010.

[30] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, New York, NY, USA, 1999.

[31] SGB – Stanford GraphBase. ftp://ftp.cs.stanford.edu/pub/sgb/, 2010.

[32] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[33] Y. Smaragdakis and D. 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.

[34] SoPlex – The Sequential Object-Oriented Simplex. http://soplex.zib.de/, 2010.

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

[36] B. Stroustrup. The design of C++0x – reinforcing C++'s proven strengths, while moving into the future. *C/C++ Users Journal*, May 2005.

[37] Z. Szűgyi and Z. Porkoláb. Quantitative comparison of MC/DC and DC test methods. In G. Falcone, Y.-G. Guéhéneuc, C. Lange, Z. Porkoláb, and H. A. Sahraoui, editors, *12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, QAOOSE Workshop, ECOOP 2008*, pages 1–10, 2008.

[38] I. Zólyomi and Z. Porkoláb. Towards a general template introspection library. In G. Karsai and E. 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.