Contents

1

Introduction

In this chapter we introduce the reader to LEDA by showing several short, but powerful, programs, we give an overview of the structure of the LEDA system, we discuss our design goals and the approach that we took to reach them, and we give a short account of the history of LEDA.

1.1 Some Programs

We show several programs to give the reader a first impression of LEDA. In each case we will first state the algorithm and then show the program. It is not essential to understand the algorithms in full detail; our goal is to show:

- how easily the algorithms are transferred into programs and
- how natural and elegant the programs are.

In other words,

Algorithm + LEDA = Program.

The directory LEDAROOT/demo/book/Intro (see Section 1.2) contains all programs discussed in this section.

1.1.1 *Word Count*

We start with a very simple program. Our task is to read a sequence of strings from standard input, to count the number of occurrences of each string in the input, and to print a list of all occurring strings together with their frequencies on standard output. The input is ended by the string “end”.

In our solution we use the LEDA types *string* and dictionary arrays (*d_arrays*). The parametrized data type dictionary array (*d_array*<I, E>) realizes arrays with index type I and element type E. We use it with index type *string* and element type *int*.

⟨*word_count.c*⟩≡

```
#include <LEDA/d_array.h>
#include <LEDA/string.h>

main()
{ d_array<string,int> N(0);
  string s;
  while ( true )
  { cin >> s;
    if ( s == "end" ) break;
    N[s]++;
  }
  forall_defined(s,N) cout << "\n" << s << " " << N[s];
}
```

We give some more explanations. The program starts with the include statement for dictionary arrays and strings. In the first line of the main program we define a dictionary array N with index type *string* and element type *int* and initialize all entries of the array to zero. Conceptually, this creates an infinite array with one entry for each conceivable string and sets all entries to zero; the implementation of d_arrays stores the non-zero entries in a balanced search tree with key type string. In the second line we define a string s. The while-loop does most of the work. We read a string s; if the string is equal to "end", we break from the loop. Otherwise, we increment the entry $N[s]$ of the array N by one. The iteration *forall_defined*(s, N) in the last line successively assigns all strings to s for which the corresponding entry of N was touched during execution. For each such string, the string and its frequency are printed on the standard output. A new line is used for each pair. On input

```
stefan
stefan
kurt
end
```

the program will print

```
kurt 1
stefan 2
```

1.1.2 *Shortest Paths*

Dijkstra's shortest path algorithm [Dij59] takes a directed graph $G = (V, E)$, a node $s \in V$, called the source, and a non-negative cost function on the edges $cost : E \to R_{\geq 0}$. It computes for each node $v \in V$ the distance from s, see Figure 1.1. A typical text book

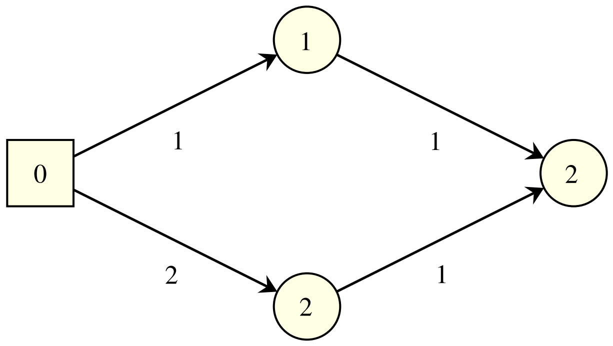

Figure 1.1 A shortest path in a graph. Each edge has a non-negative cost. The cost of a path is the sum of the cost of its edges. The source node s is indicated as a square. For each node the length of the shortest path from s is shown.

presentation of the algorithm is as follows (we will prove the correctness of the algorithm in Section 6.6):

```
set dist(s) to 0.
set dist(v) to infinity for v different from s.
declare all nodes unreached.
while there is an unreached node
{ let u be an unreached node with minimal dist-value.       (*)
  declare u reached.
  forall edges e = (u,v) out of u
     set dist(v) = min( dist(v), dist(u) + cost(e) )
}
```

The text book presentation will then continue to discuss the implementation of line (*). It will state that the pairs $\{(v, \mathit{dist}(v));\ v \text{ unreached}\}$ should be stored in a priority queue, e.g., a Fibonacci heap, because this will allow the selection of an unreached node with minimal distance value in logarithmic time. It will probably refer to some other chapter of the book for a discussion of priority queues.

We next give the corresponding LEDA program; it is very similar to the pseudo-code above. In fact, after some experience with LEDA you should be able to turn the pseudo-code into code within a few minutes.

⟨*DIJKSTRA.c*⟩≡

```
#include <LEDA/graph.h>
#include <LEDA/node_pq.h>
void DIJKSTRA(const graph &G, node s,
              const edge_array<double>& cost,
              node_array<double>& dist)
{ node_pq<double> PQ(G);
  node v; edge e;
  forall_nodes(v,G)
```

```
  { if (v == s) dist[v] = 0; else dist[v] = MAXDOUBLE;
    PQ.insert(v,dist[v]);
  }
  while ( !PQ.empty() )
  { node u = PQ.del_min();
    forall_out_edges(e,u)
      { v = target(e);
        double c = dist[u] + cost[e];
        if ( c < dist[v] )
        {  PQ.decrease_p(v,c);  dist[v] = c;  }
      }
  }
}
```

We give some more explanations. We start by including the graph and the node priority queue data types. The function *DIJKSTRA* takes a graph G, a node s, an *edge_array cost*, and a *node_array dist*. Edge arrays and node arrays are arrays indexed by edges and nodes, respectively. We declare a priority queue *PQ* for the nodes of graph G. It stores pairs $(v, dist[v])$ and is initially empty. The *forall_nodes*-loop initializes *dist* and *PQ*. In the main loop we repeatedly select a pair $(u, dist[u])$ with minimal distance value and then iterate over all out-going edges to update distance values of neighboring vertices.

We next incorporate the shortest path program into a small demo. We generate a random graph with n nodes and m edges and choose the edge costs as random number in the range [0 .. 100]. We call the function above and report the running time.

⟨*dijkstra_time.c*⟩≡

```
⟨DIJKSTRA.c⟩
main()
{ int n = read_int("number of nodes = ");
  int m = read_int("number of edges = ");
  graph G;
  random_graph(G,n,m);
  edge_array<double> cost(G);
  node_array<double> dist(G);
  edge e; forall_edges(e,G) cost[e] = ((double) rand_int(0,100));
  float T = used_time();
  DIJKSTRA(G,G.first_node(),cost,dist);
  cout << "\n\nThe shortest path computation took " <<
          used_time(T) << " seconds.\n\n";
}
```

On a graph with 10000 nodes and 100000 edges the computation takes less than a second.

1.1.3 *Curve Reconstruction*

The reconstruction of a curve from a set of sample points is an important problem in computer vision. Amenta, Bern, and Eppstein [ABE98] introduced a reconstruction algorithm which they called CRUST. Figure 1.2 shows a point set and the curves reconstructed by

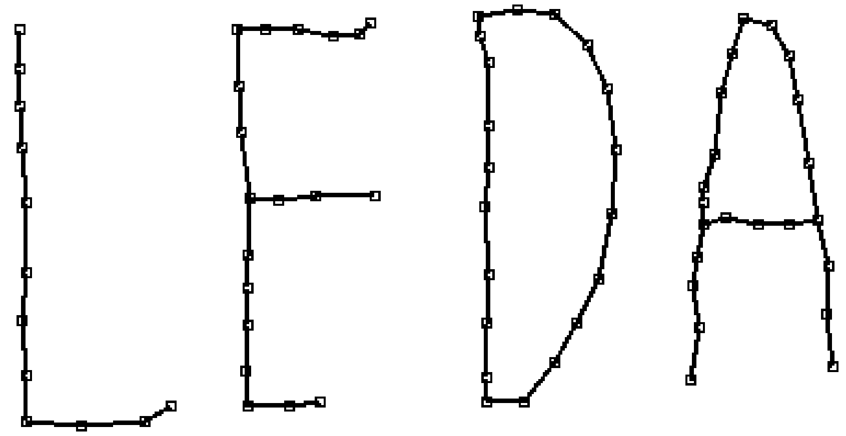

Figure 1.2 A set of points in the plane and the curve reconstructed by CRUST. The figure was generated by the program presented in Section 1.1.3.

their algorithm. The algorithm *CRUST* takes a list S of points and returns a graph G. CRUST makes use of Delaunay diagrams and Voronoi diagrams (which we will discuss in Sections 9.4 and 9.5) and proceeds in three steps:

- It first constructs the Voronoi diagram *VD* of the points in S.
- It then constructs a set $L = S \cup V$, where V is the set of vertices of *VD*.
- Finally, it constructs the Delaunay triangulation *DT* of L and makes G the graph of all edges of *DT* that connect points in S.

The algorithm is very simple to implement[1].

⟨*crust.c*⟩≡

```
#include <LEDA/graph.h>
#include <LEDA/map.h>
#include <LEDA/float_kernel.h>
#include <LEDA/geo_alg.h>
void CRUST(const list<point>& S, GRAPH<point,int>& G)
{
  list<point> L = S;
  GRAPH<circle,point> VD;
  VORONOI(L,VD);
```

[1] In 97 the authors attended a conference, where Nina Amenta presented the algorithm. We were supposed to give a presentation of LEDA later in the day. We started the presentation with a demo of algorithm CRUST.

```
    // add Voronoi vertices and mark them
    map<point,bool> voronoi_vertex(false);
    node v;
    forall_nodes(v,VD)
    { if (VD.outdeg(v) < 2) continue;
      point p = VD[v].center();
      voronoi_vertex[p] = true;
      L.append(p);
    }
    DELAUNAY_TRIANG(L,G);
    forall_nodes(v,G)
      if (voronoi_vertex[G[v]]) G.del_node(v);
  }
```

We give some explanations. We start by including graphs, maps, the floating point geometry kernel, and the geometry algorithms. In CRUST we first make a copy of S in L. Next we compute the Voronoi diagram *VD* of the points in L. In LEDA we represent Voronoi diagrams by graphs whose nodes are labeled with circles. A node v is labeled by a circle passing through the defining sites of the vertex. In particular, *VD*[v].*center*( ) is the position of the node v in the plane. Having computed *VD* we iterate over all nodes of *VD* and add all finite vertices (a Voronoi diagram also has nodes at infinity, they have degree one in our graph representation of Voronoi diagrams) to L. We also mark all added points as vertices of the Voronoi diagram. Next we compute the Delaunay triangulation of the extended point set in G. Having computed the Delaunay triangulation, we collect all nodes of G that correspond to vertices of the Voronoi diagram in a list *vlist* and delete all nodes in *vlist* from G. The resulting graph is the result of the reconstruction.

We next incorporate CRUST into a small demo which illustrates its speed. We generate n random points in the plane and construct their crust. We are aware that it does really make sense to apply CRUST to a random set of points, but the goal of the demo is to illstrate the running time.

⟨*crust_time.c*⟩≡

```
  ⟨crust.c⟩
  main()
  { int n = read_int("number of points = ");
    list<point> S;
    random_points_in_unit_square(n,S);
    GRAPH<point,int> G;
    float T = used_time();
    CRUST(S,G);
    cout << "\n\nThe crust computation took " <<
            used_time(T) << " seconds.\n\n";
  }
```

For 3000 points the computation takes less than a second.

1.1.4 *A Curve Reconstruction Demo*

We use the program of the preceding section for a small interactive demo.

⟨*crust_demo.c*⟩≡

```
#include <LEDA/window.h>
⟨crust.c⟩
main()
{ window W; W.display();
  W.set_node_width(2); W.set_line_width(2);
  point p;
  list<point> S;
  GRAPH<point,int> G;
  while ( W >> p )
  { S.append(p);
    CRUST(S,G);
    node v; edge e;
    W.clear();
    forall_nodes(v,G) W.draw_node(G[v]);
    forall_edges(e,G) W.draw_segment(G[source(e)], G[target(e)]);
  }
}
```

We give some more explanations. We start by including the window type. In the main program we define a window and open its display. A window will pop up. We state that we want nodes and edges to be drawn with width two. We define the list S and the graph G required for CRUST. In each iteration of the while-loop we read a point in W (each click of the left mouse button enters a point), append it to S and compute the crust of S in G. We then draw G by drawing its vertices and its edges. Each edge is drawn as a line segment connecting its endpoints. Figure 1.2 was generated with the program above.

1.1.5 *Discussion*

We hope that you are impressed by the programs which we have just shown you. In each case only a few lines of code were necessary to achieve complex functionality and, moreover, the code is elegant and readable. We conclude that LEDA is ideally suited for rapid prototyping as summarized in the equation

$$\text{Algorithm} + \text{LEDA} = \text{Program}.$$

The data structures and algorithms in LEDA are efficient. For example, the computation of shortest paths in a graph with 10000 nodes and 100000 edges and the computation of the crust of 3000 points took less than a second each. Thus

$$\text{Algorithm} + \text{LEDA} = \text{Efficient Program}.$$

```
Acknowledgements          acknowledgements
README                    information about LEDA
INSTALL                   this file
CHANGES                   most recent changes
FIXES                     bug fixes since last release
LEPS/                     LEDA extension packages
Manual/                   user manual
Makefile                  make script
confdir/                  configuration directory
lconfig                   configuration command
cmd/                      commands
incl/                     include directory
src/                      source files
demo/                     demo programs
test/                     test programs
data/                     data files
```

Figure 1.3 The top level of the LEDA root directory. Depending on the version of LEDA that is installed at your system, some of the files may be missing or empty.

1.2 The LEDA System

The LEDA system can be downloaded from the LEDA web-site.

`http://www.mpi-sb.mpg.de/LEDA/leda.html`

A commercial version of LEDA is available from Algorithmic Solutions Software GmbH.

`http://www.algorithmic-solutions.de`

At both places you will also find an installation guide.

Figure 1.3 shows the top level of the LEDA directory; some files may be missing or empty depending on the version of LEDA that is installed at your system. We use LEDAROOT to denote the path name of the LEDA directory. In this section we will discuss essential parts of the LEDA directory tree.

README and INSTALL tell you how to install the system. In the remainder of this section all path-names will be relative to the LEDA root directory.

1.2.1 *The Include Directory*

The include directory `incl/LEDA` contains:

- all header files of the LEDA system,
- subdirectory `templates` for the template versions of network algorithms,
- subdirectory `generic` for the kernel independent versions of geometric algorithms,
- subdirectory `impl` for header files of different implementations of dictionaries and priority queues,

- subdirectory `thread` for the classes needed to make LEDA thread-safe,
- subdirectory `sys` for the classes that adapt LEDA to different compilers and systems, and
- subdirectories `bitmaps` and `pixmaps` for bitmaps and pixel maps.

1.2.2 *The Source Code Directory*

The source code directory `src` contains the source code of LEDA. If you have downloaded an object code package, as you probably have, this directory will be empty. Otherwise, it has one subdirectory for each of the major parts of LEDA: basic data types, numbers, dictionaries, priority queues, graphs, graph algorithms, geometry kernels, geometry algorithms, windows, ...

1.2.3 *The LEDA Manual*

The directory `Manual` contains the LaTeX-sources of the LEDA manual. You may make the manual by typing "make" in this directory. This requires that certain additional tools are installed at your system. Alternatively, and we recommend the alternative, you may download the LEDA manual from our web-site. There are two versions of the LEDA manual available on our web-site:

- A paper version in the form of either a ps-file or a dvi-file.
- An HTML-version.

1.2.4 *The Demo Directory*

The directory `demo` contains demos. All demos mentioned in this book are contained in either the subdirectory `xlman` or the subdirectory `book`. We call the demos in the former directory xlman-demos.

All xlman-demos have a graphical user interface and can be accessed through the xlman-utility, see Section 1.2.5. Of course, one can also call them directly in directory xlman. You will find many screenshots in this book; many of them are screenshots of xlman-demos.

The demos in the `book`-directory typically have an ASCII-interface and demonstrate running times. The book-directory is structured according to the chapters of this book.

1.2.5 *Xlman*

Xlman gives you on-line access to the xlman-demos and the LEDA manual (if xdvi is installed at your system). Figure 1.4 shows a screenshot of xlman.

1.2.6 *LEDA Extension Packages*

LEDA extension packages (LEPs) extend LEDA into particular application domains and areas of algorithmics not covered by the core system. Anybody may contribute a LEDA extension package. At the time of writing this there are LEDA extension packages for:

Figure 1.4 A screen shot of xlman. The upper text line shows the name of a LEDA manual page, and the lower text line shows the name of an LEP manual page (this line may be missing in your installation). The six buttons at the bottom have the following functionality: on-line access to manual pages, printing manual pages, running LEDA demos, access to LEDA documents, xlman configuration, and exit. Some of the functionality relies on other tools, e.g., xdvi, and may be missing on your system.

- abstract Voronoi diagrams (by Michael Seel),
- higher-dimensional geometry (by Kurt Mehlhorn, Michael Müller, Stefan Näher, Stefan Schirra, Michael Seel, and Christian Uhrig),
- dynamic graph algorithms (by David Alberts, Umberto Nanni, Guilio Pasqualone, Christos Zaroliagis, Pippo Cattaneo, and Guiseppe F. Italiano),
- graph iterators (by Marco Nissen and Karsten Weihe),
- external memory computations (by Andreas Crauser),
- PQ-trees (by Sebastian Leipert), and
- SD-trees (by Peter Hilpert).

LEDA extension packages must satisfy a set of basic requirements which guarantee compatibility with the LEDA philosophy; the requirements are defined on our web-page.

1.3 The LEDA Web-Site

The LEDA web-site (`http://www.mpi-sb.mpg.de/LEDA/leda.html`) is an important source of information about LEDA. We mentioned already that it allows you to download the most recent version of the system and the manual. It also gives you information about the people behind LEDA and latest news, and it contains pointers to other systems which are either built on top of LEDA or which we have used successfully together with LEDA. We will discuss some of these systems in the next section.

1.4 Systems that Go Well with LEDA

Although LEDA covers many aspects of combinatorial and geometric computing, it cannot cover all of them. In our own work we therefore also use other systems.

In the realm of exact solution of NP-complete problems we use LEDA together with ABACUS, a branch-and-cut framework for polyhedral optimization, and with CPLEX and SoPLEX, two solvers for linear programs. ABACUS was developed by Michael Jünger, Stefan Reinelt, and Stefan Thienel.

For graph drawing we use AGD, a library of automatic graph drawing, and GDToolkit , a toolkit for graph drawing. AGD is a joint effort of Petra Mutzel's group at the MPI, Stefan Näher's group in Halle, and Michael Jünger's group in Cologne. GDToolkit was developed by Guiseppe Di Battista's group in Rome. We say a bit more about AGD and GDToolkit in Section 8.1.

For computational geometry we also use CGAL, a computational geometry algorithms library. CGAL is a joint effort of ETH Zürich Freie Universität Berlin, INRIA Sophia Antipolis, Martin-Luther Universität Halle-Wittenberg, Max-Planck-Institut für Informatik and Universität des Saarlandes, RISC Linz Tel-Aviv University, and Universiteit Utrecht. We will say more about CGAL in Section 8.11.

1.5 Design Goals and Approach

We had four major goals for the design of LEDA:

- Ease of use.
- Extensibility.
- Correctness.
- Efficiency.

We next discuss our four goals and how we tried to reach them.

We wanted the library to reduce the gap between the algorithms community and the "rest of the world" and therefore *ease of use* was a major concern. We wanted the library to be useable without intimate knowledge of our field of research; a basic course in data structures and algorithms should suffice. We also wanted the data types and algorithms of LEDA to be useable without any knowledge of their implementation.

We invented the item concept, see Section 2.2, as an abstraction of the concept of "pointer to a container in a data type" and used it for the specification of all container-based data types. We formulated rules (see Chapter 2) that capture key concepts, such as copy constructor, assignment, and compare functions, uniformly for all data types. We introduced a powerful graph type, see Chapter 6, which supports the natural and elegant formulation of graph and network algorithms and is also the basis for many of the geometric algorithms.

Ease of use also means easy access to information. The LEDA manual, see Chapter 14, gives precise and readable specifications for the LEDA data types and algorithms. The specifications are short (typically not more than a page), general (so as to allow several implementations) and abstract (so as to hide all details of the implementation). All specifications follow a common format, see Section 2.1. We developed tools that support the production of manual pages and documentations. Finally, we wrote this book that gives a comprehensive view of LEDA.

Combinatorial and geometric computing is a diverse area and hence it is impossible for a library to provide ready-made solutions for all application problems. For this reason it is important that LEDA is easily extensible and can be used as a platform for further software development. LEDA itself is a good example for the *extensibility* of LEDA. The advanced data types and algorithms discussed in Chapters 5, 7, 8, and 9 are built on top of the basic data types introduced in Chapters 3, 4, 6, and 8. The basic data types in turn rest on a conceptual framework described in Chapter 2 and the implementation principles discussed in Chapter 13.

Incorrect software is hard to use at best and dangerous at worst. We underestimated the difficulties of achieving *correctness*. After all, any publication in our area proves the correctness of the described algorithms and going from a correct algorithm to a correct program is tedious and time-consuming, but hardly an intellectual challenge. So we thought, when we started the project. We now think differently.

Many of the algorithms in LEDA are quite intricate and therefore difficult to implement correctly. Programmers make mistakes and we are no exception. How do we guard against errors? Many of our implementations are carefully documented (this book contains many examples), we test extensively, as does our large user community, and we have recently adopted the philosophy that programs should give sufficient justification for their answers to allow checking, see Section 2.14. We have developed program checkers for many of our programs.

The correct implementation of geometric programs was particularly difficult, as the theoretical underpinning was insufficient. Geometric algorithms are typically derived under two simplifying assumptions: (1) the underlying machine model is the *real RAM* which can compute with real numbers in the sense of mathematics and (2) inputs are in general position. However, the number types *int* and *double* offered by programming languages are only crude approximations of real numbers and practical inputs are frequently degenerate. Our approach is to formulate geometric algorithms such that they work for all inputs, see Chapter 9, and to realize the real RAM (as far as it is needed for computational geometry) by exact number types, see Chapter 4, and an exact and yet efficient geometry kernel, see Chapter 8.

Efficiency was our fourth design goal. It may surprise some readers that we list it last. However, efficiency without correctness is meaningless and efficiency without ease of use is a questionable blessing. We achieve efficiency by the use of efficient algorithms and their careful implementation.

Our implementations are usually based on the asymptotically most efficient algorithms known for a particular problem. In many cases we even implemented different algorithmic approaches. For example, there are several shortest path, matching, and flow algorithms, there are several convex hull, line segment intersection, and Delaunay diagram algorithms, and there are several realizations of dictionaries and priority queues. In the case of data types, the implementation parameter mechanism allows the convenient selection of an implementation. For example, the declarations

```
dictionary<string,int> D1;
dictionary<string,int,skip_list> D2;
```

declare *D1* as a dictionary from *string* to *int* with the default implementation and select the skip list implementation for *D2*.

The description of many algorithms leaves considerable freedom for the implementor, i.e., a description typically defines a family of algorithms all with the same asymptotic worst case running time and leaves decisions that do not affect worst case running time to the implementor. The decisions may, however, dramatically affect the running time on inputs that are not worst case. We have carefully explored the available opportunities; Sections 7.6 and 7.10 give particularly striking examples. We found it useful to concentrate on the best and average case after getting the worst case "right".

LEDA has its own memory manager. It provides efficient implementations of the *new* and *delete* operators.

How efficient are the programs in LEDA? We give many tables of running times in this book which show that LEDA programs are able to solve large problem instances. We made comparisons, see Tables 1.1 and 1.2, and other people did, see for example [Ski98, Ski]. The comparisons show that our running times are competitive, despite the fact that LEDA is more like a decathlon athlete than a specialist for a particular discipline.

1.6 History

We started the project in the fall of 1988. We spent the first six months on specifications and on selecting our implementation language. Our test cases were priority queues, dictionaries, partitions, and algorithms for shortest paths and minimum spanning trees. We came up with the item concept as an abstraction of the notion "pointer into a data structure". It worked successfully for the three data types mentioned above and we are now using it for most data types in LEDA. Concurrently with searching for the correct specifications

Number of list entries: 100000

LIST<INT>	LEDA	STL
build list	0.020 sec	0.040 sec
pop and push	0.030 sec	0.030 sec
reversing	0.020 sec	0.030 sec
copy constr	0.050 sec	0.050 sec
assignment	0.020 sec	0.040 sec
clearing	0.000 sec	0.020 sec
sorting	0.130 sec	0.400 sec
sorting again	0.140 sec	0.330 sec
merging	0.030 sec	0.080 sec
unique	0.080 sec	0.080 sec
unique again	0.000 sec	0.010 sec
iteration	0.000 sec	0.000 sec
total	0.520 sec	1.110 sec

LIST<CLASS>	LEDA	STL
build list	0.090 sec	0.030 sec
pop and push	0.100 sec	0.030 sec
reversing	0.070 sec	0.030 sec
copy constr	0.140 sec	0.060 sec
assignment	0.120 sec	0.030 sec
clearing	0.080 sec	0.020 sec
sorting	0.770 sec	0.510 sec
sorting again	0.900 sec	0.380 sec
merging	0.200 sec	0.090 sec
unique	0.250 sec	0.100 sec
unique again	0.010 sec	0.000 sec
iteration	0.010 sec	0.000 sec
total	2.740 sec	1.280 sec

Table 1.1 A comparison of the list data type in LEDA and in the implementation of the Standard Template Library [MS96] that comes with the GNU C++ compiler. The upper part compares *list<int>* and the lower part compares *list<class>*, where the objects of type *class* require several words of storage. LEDA lists are faster for small objects and slower for large objects. This table was generated by the program stl_vs_leda in LEDAROOT/demo/stl. You can perform your own comparisons if your C++ compiler comes with an implementation of the STL. All running times are in seconds.

we investigated several languages for their suitability as our implementation platform. We looked at Smalltalk, Modula, Ada, Eiffel, and C++. We wanted a language that supported abstract data types and type parameters (polymorphism) and that was widely available. We wrote sample programs in each language. Based on our experiences we selected C++ because of its flexibility, expressive power, and availability.

A first publication about LEDA appeared in the conferences MFCS 1989 and ICALP 1990 [MN89, NM90]. Stefan Näher became the head of the LEDA project and he is the

Type of network	LEDA	CG
random (4000 nodes 28000 edges)	0.31	0.11
CG1 (8002 nodes 12003 edges)	9.26	4.20
CG2 (8002 nodes 12001 edges)	0.11	0.73
AMO (4001 nodes 7998 edges)	0.05	1.74

Table 1.2 A comparison of the maxflow implementation in LEDA and the one by Cherkassky and Goldberg [CG97, CG]. The latter implementation is generally considered the best code available. We used four different kinds of graphs: random graphs and graphs generated by three different generators. The generators CG1 and CG2 were suggested by Cherkassky and Goldberg and the generator AMO was suggested by Ahuja, Magnanti, and Orlin. The generators are discussed in detail in Section 7.10. All running times are in seconds. You may perform your own experiments by running the program flow_test in LEDAROOT/demo/book/Intro and following the instructions given in MAXFLOW_README in the same directory.

main designer and implementer of LEDA. Progress reports appeared in [MN92, Näh93, MN94, MN95, BKM$^+$95, MNU97, MN98].

In the second half of 1989 and during 1990 Stefan Näher implemented a first version of the combinatorial part (= data structures and graph algorithms) of LEDA (Version 1.0). Then there were releases 2.0, 2.1, 2.2, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7. With the appearance of this book we will release version 4.0. Each new release offered new functionality, increased efficiency, and removed bugs.

LEDA runs on many different platforms (Unix, Windows, OS/2) and with many different compilers.

In early 1995 LEDA Software GmbH was founded to market a commercial version of LEDA. Christian Uhrig became the Chief Executive Officer. The company was renamed to Algorithmic Solutions Software GmbH in late 1997 to reflect the fact that it not only markets LEDA, but also other systems like, for example, AGD and CGAL, and that it also develops algorithmic solutions for specific needs.

The research version is used at more than 1500 academic and research sites. Try the website `http://www.mpi-sb.mpg.de/LEDA/DOWNLOADSTAT` to find out whether the system is already in use at your site.

Bibliography

[ABE98] N. Amenta, M. Bern, and D. Eppstein. The crust and the β-skeleton: Combinatorial curve reconstruction. *Graphical Models and Image Processing*, pages 125–135, 1998.

[BKM+95] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proceedings of the 11th Annual Symposium on Computational Geometry (SCG'95)*, pages C18–C19, New York, NY, USA, June 1995. ACM Press.

[CG] B. Cherkassky and A. Goldberg. PRF, a Maxflow Code. `www.inter-trust.com/star/goldberg/index.html`.

[CG97] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[Dij59] E.W. Dijkstra. A note on two problems in connection with graphs. *Num. Math.*, 1:269–271, 1959.

[MN89] K. Mehlhorn and S. Näher. LEDA: A library of efficient data types and algorithms. In Antoni Kreczmar and Grazyna Mirkowska, editors, *Proceedings of the 14th International Symposium on Mathematical Foundations of Computer Science (MFCS'89)*, volume 379 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 1989.

[MN92] K. Mehlhorn and S. Näher. Algorithm design and software libraries: Recent developments in the LEDA project. In *Algorithms, Software, Architectures, Information Processing 92*, volume 1, pages 493–505. Elsevier Science Publishers B.V. North-Holland, 1992.

[MN94] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[MN95] K. Mehlhorn and S. Näher. LEDA, a Platform for Combinatorial and Geometric Computing. *Communications of the ACM*, 38:96–102, 1995.

[MN98] K. Mehlhorn and S. Näher. From algorithms to working programs: On the use of program checking in LEDA. In *Proceedings of Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 84–93, 1998.

[MNU97] K. Mehlhorn, S. Näher, and C. Uhrig. The LEDA platform for combinatorial and geometric computing. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *Lecture Notes in Computer Science*, pages 7–16, 1997.

[MS96] D.R. Musser and A. Saini. *STL tutorial and Reference Guide*. Addison-Wesley, Reading, 1996.

[Näh93] S. Näher. LEDA: a library of efficient data types and algorithms. In Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner, editors, *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*

(STACS'93), volume 665 of *Lecture Notes in Computer Science*, pages 710–723. Springer, 1993.

[NM90] S. Näher and K. Mehlhorn. LEDA: A library of efficient data types and algorithms. In Michael S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages, and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 1–5. Springer, 1990.

[Ski] S.S. Skiena. The Stony Brook Algorithm Repository. `www.cs.sunysb.edu/~algorith/index.html`.

[Ski98] S.S. Skiena. *The Algorithm Design Manual*. Springer, 1998.

Index