THE LEDA PLATFORM FOR COMBINATORIAL AND GEOMETRIC COMPUTING

K Mehlhorn

Rapporteur: Ian Welch

The LEDA Platform for Combinatorial and Geometric Computing

Kurt Mehlhorn* Stefan Näher† Christian Uhrig‡

September 1, 96

Abstract

We give an overview of LEDA and an account of its development. We discuss our motivation for building LEDA and to what extent we have reached our goals. We also discuss some recent theoretical developments. This paper contains no new technical material. It is intended as a guide to further papers about the system. We also refer the reader to `http://www.mpi-sb.mpg.de/LEDA/leda.html`, `http://www/informatik.uni-halle.de/~naeher`, and `http://www.mpi-sb.mpg.de/~mehlhorn` for more information.

1 What is LEDA?

LEDA [13, 16] aims at providing a comprehensive software platform for combinatorial and geometric computing. Combinatorial and geometric computing is a core area of computer science. In fact, most CS curricula contain a course in data structures and algorithms. The area deals with objects such as graphs, sequences, dictionaries, trees, shortest paths, flows, matchings, points, segments, lines, convex hulls, and Voronoi diagrams and forms the basis for application areas such as discrete optimization, scheduling, traffic control, CAD, and graphics. We discuss different aspects of the LEDA system.

Coverage: LEDA provides a sizable collection of data types and algorithms. This collection includes most of the data types and algorithms described in the text books of the area ([1, 10, 21, 6, 18, 23, 19, 9, 22, 17]). In particular, it includes stacks, queues, lists, sets, dictionaries, ordered sequences, partitions, priority queues, directed, undirected, and planar graphs, lines, points, planes, and polygons, and many algorithms in graph and network theory and computational geometry, e.g., shortest paths, matchings, maximum flow, min cost flow, planarity testing, spanning trees, biconnected and strongly connected components, segment intersection, convex hulls, Delaunay triangulations, and Voronoi diagrams.

*Max-Planck-Insitut für Informatik, Im Stadtwald, 66123 Saarbrücken

†Matin-Luther-Universität Halle-Wittenberg, FB Mathematik und Informatik, Weinbergweg 17, 060099 Halle

‡LEDA Software GmbH, Postfach 15 11 01, 66041 Saarbrücken

Ease of Use: The library is easy to use. In fact, only a small fraction of our users are algorithms experts and many of our users are not even computer scientists. LEDA supports applications in a broad range of areas. It has already been used in such diverse areas as code optimization, VLSI design, robot motion planning, traffic scheduling, machine learning and computational biology.

The LEDA manual [16] gives precise and readable specifications for the data types and algorithms mentioned above. 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).

In many cases LEDA programs are very close to the typical text book presentation of the underlying algorithms. The goal is the equation

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

We give an example. Dijkstra's shortest path algorithm takes a directed graph $G = (V, E)$, a node $s \in V$, called the source, and a non-negative cost function on the edges $cost : E \to \mathbb{R}_{\geq 0}$. It computes for each node $v \in V$ the distance from s. A typical text book presentation of the algorithm is as follows.

```
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 $\{(v, dist(v)); v \text{ unreached}\}$ 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 now give the corresponding LEDA program; it is very similar to the presentation above.

```
#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_adj_edges(e,u)
        { v = target(e);
          double c = dist[u] + cost[e];
          if ( c < dist[v] )
          {  PQ.decrease_inf(v,c);  dist[v] = c;  }
        }
    }
  }
```

We start by including the graph and the node priority queue data type. We use *edge_arrays* and *node_arrays* (arrays indexed by edges and nodes respectively) for the functions *cost* and *dist*. We delare a priority queue PQ for the nodes of graph G. It stores pairs $(v, dist[v])$ and is empty initially. 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 scan through all adjacent edges to update distance values of neighboring vertices.

Correctness: We try to make sure that the programs in LEDA are correct. We start from correct algorithms, we document our implementations carefully (at least recently), we test them extensively, and we have developed program checkers (see subsection 4.1) for some of them. We want to emphasize that many of the algorithms in LEDA are quite intricate and therefore non-trivial to implement. In the combinatorial domain it is frequently possible to obtain a correct implementation by sacrifycing efficency, e.g., by using linear search in the realization of a dictionary. In the geometric domain it is usually difficult to obtain a correct implementation even if efficiency plays no role. This is due to the so-called degeneracy and precision problem [12]. The geometric algorithms in LEDA use exact arithmetic and are therefore free from failures due to rounding errors. Moreover, they can handle all degenerate cases.

Efficiency: LEDA contains the most efficient realizations known for its types. For many data types the user may even choose between different implementations, e.g., for dictionaries he may choose between *ab*-trees, $BB[\alpha]$-trees, dynamic perfect hashing, and skip lists. 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*.

Availability: LEDA is realized in C++ and runs on many different platforms (Unix, Windows95, Windows NT, OS/2) with many different compilers. It is not in the public domain but it is availabe free of charge for academic use. See `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

History: We started the project in the fall of 1988. We spent the first 6 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 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. We are even more convinced now that our choice was the right one.

A first publication about LEDA appeared in MFCS 1989 (Lecture Note in Computer Science, Volume 379) and ICALP 1990 (Lecture Notes in Computer Science, Volume 443). Stefan Näher became the head of the LEDA project and he is the main designer and implementer of LEDA.

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). Version 2.0 allowed to use arbitrary data types (not only pointer and simple types) as actual type parameters of parametrized data types. It included a first implementation of the two-dimensional geometry library (libP) and an interface to the X-Window system for graphical input and output (data type window). Version 3.0 switched to the template mechanism to realize parametrized data types (macro substitution was used before), introduced implementation parameters that allow to choose between different implementations, extended the LEDA memory management system to user-defined classes, and further improved the efficiency of many data types and algorithms. Version 3.1 provided a more efficient graph data type and contained new data types (arbitrary precision number types and basic geometric objects) used for robust implementations of geometric algorithms and Versions 3.2 and 3.3 contained more geometry and new tools for documentation and manual production.

LEDA Software GmbH was founded in early 1995.

2 Why did we build LEDA?

We had four main reasons:

1. We had always felt that a significant fraction of the research done in the algorithms area was emminently practical. However, only a small part of it was actually used. We frequently heard from our former students that the effort needed to implement an advanced data structure or algorithm is too large to be cost-effective. We concluded that *algorithms research must include implementation if the field wants to have maximum impact.*

2. Even within our own research group we found different implementations of the same balanced tree data structure. Thus there was constant reinvention of the wheel even within our own tight group.

3. Many of our students had implemented algorithms for their master's thesis. Work invested by these students was usually lost after the students graduated. We had no depository for implementations.

4. The specifications of advanced data types which we gave in class and which we found in text books, including the one written by one of the authors, were incomplete and not sufficiently abstract. They contained phrases of the form: "Given a pointer to a node in the heap its key can be decreased in constant amortized time". This implied that a user of a data structure had to have knowledge of its implementation. As a consequence combining implementations was a non-trivial task. A case in point is the shortest path problem in graphs. We taught priority queues in the early weeks of an algorithm course and Dijkstra's algorithm for the shortest path problem in later weeks. Our students found it difficult to combine the programs.

The goal of the LEDA project is to overcome these shortcomings by creating a platform for combinatorial and geometric computing. The LEDA library should contain the major findings of the algorithms community in a form that makes them directly accessible to non-experts having only a limited knowledge in the area. In this way we hoped to reduce the gap between research and application.

3 Did we achieve our goals?

We believe that we have reached the last goal and have at least partially reached the first three goals.

LEDA was first distributed in the summer of 1990. Its user community has grown ever since. LEDA is now used at more than 1500 academic and industrial sites in over 50 different countries world-wide. Industrial use started in 1994. Many users of LEDA are outside computer science and only a small fraction of our users are from the algorithms community. We therefore believe that we have reached our first two goals. The impact of algorithms research has increased and there is considerable use of LEDA and hence reuse of implementations. However, the gap between algorithms research and algorithms use is still quite large. In particular, many of the non-expert users of LEDA complain that a tutorial is missing. We hope that the forthcoming LEDAbook [14] will help.

We have also partially achieved our third goal. We now do have a depository for our students work. We do not have a strategy yet which allows persons outside our research groups to contribute to LEDA. We hope to have made a significant step in this direction recently. We redesigned our documentation tools and now make them publicly available with the LEDA release. This allows other reseachers to produce LEDA-style manual pages and documentations and should make it easier to develop extensions of LEDA that gain widespread use.

We have achieved our last goal. The specifications of our data types are sufficently abstract and precise so as to allow their combination without any knowledge of implementation. We have seen an example in section 1. Many of our specifications are based on the so-called *item concept* which gives an abstract treatment of pointers into a data structure. Different components of LEDA can be combined without knowledge of the implementation.

The project also had a number of positive side-effects which we did not foresee. Firstly, LEDA's wide use gives us tremendous satisfaction[1]. Secondly, the experiences with the system suggest many difficult and well motivated problems for theoretical algorithms research. We will discuss program checking and theoretical issues in the implementation of geometric algorithms below. *The system has changed the way we do algorithms research.*

4 Recent developments

A strength of the LEDA project is its strong theoretical underpinning. *We believe that only our strong theoretical background allowed us to build LEDA.* In the last two years we paid particular attention to program checking and the correct implementation of geometric programs.

4.1 Program checking

Programming is a notoriously errorprone task; this is even true when programming is interpreted in a narrow sense: going from a (correct) algorithm to a program. The standard way to guard against coding errors is program testing. The program is exercised on inputs for which the output is known by other means, typically as the output of an alternative program for the same task. Program testing has severe limitations:

- It is usually only done during the testing phase of a program. Also, it is difficult to determine the "correct" suite of test inputs.

- Even if appropriate test inputs are known it is usually difficult to determine the correct outputs for these inputs: alternative programs may have different input and output conventions or may be too inefficient to solve the test cases.

Given that program verification, i.e., formal proof of correctness of an implementation, will not be available on a practical scale for some years to come, *program checking* has been proposed as an extension to testing [2, 3]. The cited papers explored program checking in the area of algebraic, numerical, and combinatorial computing. In [15, 11, 8] we discuss program checkers for planarity testing and a variety of geometric tasks. We have also added program checkers to some of the LEDA programs, e.g., the planarity test provides a planar drawing for a planar graph and a Kuratowski subgraph for a non-planar graph. A user of the planarity algorithm has thus the possibility to verify that the output of the algorithm is correct.

4.2 Implementation of geometric algorithms

Geometric algorithms are frequently formlated under two unrealistic assumptions: computers are assumed to use exact real arithmetic (in the sense of mathematics) and inputs are assumed to be in general position. The naive use of floating point arithmetic as an approximation to exact real arithmetic very rarely leads to correct implementations. In a sequence of papers [4, 20, 12, 5, 7] we investigated the degeneracy and precision

[1]We stated above that algorithms research must include implementation to have maximal impact. We might add: without implementation algorithm research is less rewarding.

issues and extended LEDA based on our theoretical work. LEDA now provides exact geometric kernels for two-dimensional and higher dimensional computational geometry and also correct implementations for basic geometric tasks, e.g.. two-dimensional convex hulls. Delaunay diagrams, Voronoi diagrams, point location, line segment intersection. and higher-dimensional convex hulls and Delaunay diagrams.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data structures and algorithms.* Addison-Wesley, 1983.

[2] M. Blum and S. Kannan. Programs That Check Their Work. In *Proc. of the 21th Annual ACM Symp. on Theory of Computing*, 1989.

[3] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd Annual ACM Symp. on Theory of Computing*, pages 73–83, 1990.

[4] Ch. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. SODA 94*, pages 16–23, 1994.

[5] Ch. Burnikel, K. Mehlhorn, and St. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In Springer-Verlag Berlin/New York, editor, *LNCS*, volume 855 of *Proceedings of ESA'94*, pages 227–239, 1994.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill Book Company, 1990.

[7] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL Kernel: A basis for geometric computation. *To appear at Workshop on Applied Computational Geometry (WACG96)*, 1996.

[8] C. Hundack, K. Mehlhorn, and S. Näher. A Simple Linear Time Algorithm for Identifying Kuratowski Subgraphs of Non-Planar Graphs. Manuscript, 1996.

[9] J.H. Kingston. *Algorithms and Data Structures.* Addison-Wesley Publishing Company, 1990.

[10] K. Mehlhorn. *Data structures and algorithms 1,2, and 3.* Springer, 1984.

[11] K. Mehlhorn and P. Mutzel. On the Embedding Phase of the Hopcroft and Tarjan Planarity Testing Algorithm. *Algorithmica*, 16(2):233–242, 1995.

[12] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[13] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[14] K. Mehlhorn and S. Näher. The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press. forthcoming, 1997.

[15] K. Mehlhorn. S. Näher. T. Schilz. S. Schirra. M. Seel, R. Seidel. and Ch. Uhrig. Checking Geometric Programs or Verification of Geometric Structures. In *Proc. of the 12th Annual Symposium on Computational Geometry*, pages 159–165. 1996.

[16] S. Näher and Ch. Uhrig. The LEDA User Manual (Version R 3.2). Technical Report MPI-I-95-1-002. Max-Planck-Institut für Informatik, 1995.

[17] J. Nievergelt and K.H. Hinrichs. *Algorithms and Data Structures.* Prentice Hall Inc., 1993.

[18] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[19] R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, 1991.

[20] Michael Seel. Eine Implementierung abstrakter Voronoidiagramme. Master's thesis, Max-Planck-Institut für Informatik, 1994.

[21] R.E. Tarjan. Data structures and network algorithms. In *CBMS-NSF Regional Conference Series in Applied Mathematics*, volume 44, 1983.

[22] C.J. van Wyk. *Data Structures and C programs.* Addison-Wesley Publishing Company, 1988.

[23] D. Wood. *Data Structures, Algorithms, and Performance.* Addison-Wesley Publishing Company, 1993.

DISCUSSION

Rapporteur: Ian Welch

Lecture One

Whilst Professor Mehlhorn was explaining a slide on node priority queues (NPQs) Professor Paterson asked what is the NPQ structure's relationship to the graph structure. Professor Paterson observed that from the documentation the NPQ appeared just to be a set of pairs of priorities and edges with no explicit reference to graph structure. Professor Mehlhorn replied that the priority queue data structure could be associated with anything and the NPQ was itself derived from the standard priority queue to deal with graphs, also the documentation that was shown was a précis of the full documentation included with LEDA. Professor Paterson asked if standard priority queues included the increase and decrease methods shown for the NPQ. Professor Mehlhorn replied that this was a specialisation to handle graphs.

When Professor Mehlhorn was outlining the history of LEDA Professor Randell asked what was the principle for selection of programming language for the implementation of LEDA. Professor Mehlhorn responded that his group had spent a considerable amount of time choosing the implementation language and ended choosing C++ because they found it more expressive than other languages for their purposes. For instance the group had decided that their ability to specify parameterised data types was important and at the time (during the late eighties) only C++ provided any sort of support for this. After Professor Mehlhorn had finished the lecture and invited questions from the floor Professor Randell began by asking a question about the quality control aspects to LEDA. He wondered if the LEDA library would remain faithful to the abstractions that Professor Mehlhorn's group had devised if outsiders contributed algorithms to the library. He raised the possibility that if anyone could add to the LEDA library the consistency and faithfulness to the model devised by Professor Mehlhorn's group would be lost. Professor Mehlhorn replied that the library proper will be kept a closed shop. When he had spoken about others contributing to LEDA he had meant that other people could create extension libraries to be used at the user's own risk. The extension libraries would be created using LEDA tools so they will be coherent in terms of look but there would be no guarantee that the extension libraries were better or worse quality than the core LEDA library. This will help the group maintain the considerable investment made in devising the best abstractions for the LEDA library.

Dr Bird asked if Professor Mehlhorn would redisplay the manual page. When it was redisplayed he asked if it was not a bit terse if it was the only documentation provided with LEDA. Professor Mehlhorn pointed out that the manual page is in reality twice as long and the one on display was an edited version reduced in size so it could be easily shown on a slide. Even so the group recognised the high-level nature of the documentation and were now working on a tutorial book to accompany LEDA.

Dr Larcombe asked if runtime and compile time checking could be turned off to increase the efficiency of LEDA routines. So a programmer could have full and expensive checking running during the test phase, but disable checking for the distribution version. Professor Mehlhorn replied that the more expensive checking could be disabled, and some preconditions are not checked at all with this task left up to the programmer.

Dr Andersson said that as the LEDA algorithms are highly efficient the cost of checks are well compensated for. In addition the structures provided by LEDA make programming a much easier task at the expense of a slightly higher execution cost. Professor Mehlhorn cited a case where Siemens research workers had built their own library that ran three times faster than LEDA but at the cost of building a library that was inflexible, and made the evolution of software very difficult.

Dr Goldberg asked the question whether there was a conflict between the aims of modularity and efficiency. For instance when processing a large data structure each LEDA module would rewalk the structure for each algorithm that was applied to the structure whereas if a custom solution was written each algorithm could be applied as the structure was walked only once. Professor Mehlhorn agreed that this was a problem. Given this modular algorithms should only be used when the problem can fit into main memory. During the fall he would be setting students the task of implementing a library to handle processing of problems that must be stored in secondary memory and he expected the resulting library would look very different to LEDA.

Dr Andersson asked how large was LEDA's function space. Is it possible to remember all the functions without recourse to documentation? Professor Mehlhorn stated that there are about 50 different data types, 5 or 6 number types, 15 basic combinatorial types, and many geometric types. Each may have multiple operations. In order to aid the user trying to remember function names the group tried to keep function names similar i.e. insert is used for similar methods but it is not always the case. There are on-line manual pages that support substring searching so lessening the reliance on memory. Professor Mehlhorn stated that he used the manual pages frequently e.g. to remember if a keyword is orient or orientation etc. Professor Mehlhorn said that the group did have some general rules but the size of the task means they cannot cater for every situation.

Lecture Two

A delegate asked about the graphical user interface. The delegate was concerned that making it too 'user friendly' actually made the interface too complicated and obscured the purpose of LEDA. He had found some 'user friendly' interfaces hard to use and was worried he might have problems with LEDA as a result. Professor Mehlhorn replied that the user interface being presented during the demonstration was just a harness for the LEDA library - LEDA was the library routines proper. LEDA does not contain any interface code. Professor Randell asked if the facilities used to create the demonstration interface for LEDA were available separately so that other user interface modules could be easily constructed. Professor Mehlhorn replied that the demonstration was supplied with LEDA, and the new LEDA book contained exercises that made use of the demonstration software but no user interface building tools were included with the current distribution of LEDA. His colleague on the project is looking into producing tools that would allow the easy building of graphical user interfaces for LEDA without the need to start from scratch every time. In response to the discussion about the use of result checker programs within LEDA a delegate asked if Professor Mehlhorn's group had carried out formal proofs for the checker programs. Professor Mehlhorn replied that no, they hadn't done strict formal analysis of the checker programs but the programs were based on algorithms available in the literature which could be assumed to be correct. In addition when the programs were written they were written in the style of 'literate programming' so the source algorithms were closely tied to the code lessening the chance of error. To help keep checker programs manageable they are kept very simple and so easily tested.

The same delegate raised his concern that the checker would be ineffective if it contained a logical error so relying upon a checker program could be dangerous. Professor Mehlhorn acknowledged that this was a possibility but as of yet it hadn't happened, and besides he wasn't sure there was any way out of the problem that the checker itself might be wrong.

Professor Randell asked if checkers were used to check non-LEDA programs as well as LEDA programs. Professor Mehlhorn pointed out that the problem was taking the output of the non-LEDA programs. Other people used different data representations making it difficult to reconcile their checkers with other people's programs.

In response to a delegate Professor Mehlhorn replied that the source code for LEDA was available as part of the normal distribution. A delegate asked was linear algebra implemented for the LEDA REAL number type. Professor Mehlhorn replied that linear algebra could be applied, for instance Gaussian elimination but in practice because of the semi-interpreted nature of the REAL data type the algorithm would be very slow. Dr Andersson said that he thought checkers were a good idea but wondered if randomised checking was needed as well as the checking of a single result. Professor Randell asked for clarification. Dr Andersson said that his interest was not purely in random use or ordering of use of functions but random choices of parameters or data and checking the algorithms against these. Professor Mehlhorn agreed that this approach was a necessary one. The problem their group had with a particular algorithm would have been detected far earlier if the planar graphs being tested were generated randomly and checked automatically. In fact they now do this and generate very large numbers of examples. Dr Bird asked what operations could be done on the LEDA real data type. Professor Mehlhorn replied arithmetic operators and sign test but not for trigonometry or anything similar.