LEDA
A Platform for Combinatorial and Geometric Computing*

Kurt Mehlhorn
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

Stefan Näher
Fachbereich Mathematik und Informatik,
Martin-Luther-Universität Halle,
06099 Halle (Saale), Germany

Key words: abstract data type, software library, software reuse, efficient algorithms, object oriented programming, C++

Abstract

LEDA is a library of efficient data types and algorithms in combinatorial and geometric computing. The main features of the library are its wide collection of data types and algorithms, the precise and readable specification of these types, its efficiency, its extendibility, and its ease of use.

1. Introduction

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. There is no standard library of the data structures and algorithms of combinatorial and geometric computing. This is in sharp contrast to many other areas of computing. There are, e.g., packages in statistics (SPSS), numerical analysis (LINPACK, EISPACK), symbolic computation (MAPLE, MATHEMATICA), and linear programming (CPLEX).

The lack of a library severely limits the impact of the area on computer science as a whole. The continuous reimplementation of basic data structures and algorithms slows down progress within research and even more so outside. The latter is due to the fact that outside research the investment for implementing an efficient solution is frequently not made, since it is doubtful whether the implementation can be reused, and therefore methods which are known to be less efficient are used instead. As a consequence, scientific discoveries migrate only slowly into practice.

*This work was supported in part by the German Ministry for Research and Technology (Bundesministerium für Forschung und Technologie) under grant ITS 9103 and by the ESPRIT Basic Research Actions Program under contract No. 7141 (project ALCOM II).

Why is it like this? One of the major differences between combinatorial and geometric computing and other areas of computing such as statistics, numerical analysis, and linear programming is the use of complex data types. Whilst the built-in types, such as integers, reals, vectors and matrices, usually suffice in the other areas, combinatorial and geometric computing relies heavily on types like stacks, queues, lists, dictionaries, sequences, graphs, points, lines, convex hulls, ... and therefore asks for a programming language where all these types are available. Only with the advent of object-oriented programming did it become feasible to provide such an extension in a clean way.

In 1989, we started the LEDA project (Library of Efficient Data Types and Algorithms) to build a library of the data types and algorithms of combinatorial and geometric computing. The features of LEDA are:

- LEDA provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. This collection includes most of the data types and algorithms described in the text books of the area (e.g. [AHU83, Meh84, Tar83, CLR90, O'R94, Woo93, Sed91, Kin90, Wyk88, NH93]), i.e., stacks, queues, lists, sets, dictionaries, ordered sequences, partitions, priority queues, directed, undirected, and planar graphs, lines, points, planes and many algorithms in graph and network theory and computational geometry.
- LEDA gives a precise and readable specification for each of 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).
- Many data types in LEDA are parameterized, e.g., the dictionary type works for arbitrary key and information type. A specific instance D with key type **string** and information type **int** is for example defined by
  dictionary $\langle$**string**, **int**$\rangle$ D;
- LEDA contains the most efficient realizations known for its types. For many data types the user may even choose between different implementations, e.g., between ab-trees, $BB[\alpha]$-trees, dynamic perfect hashing, skip lists, ... for dictionaries.
  _dictionary $\langle$**string**, **int**, *skip_list*$\rangle$ D;
  realizes dictionary D by skip lists.
- For many efficient data structures access by position is crucial. LEDA uses a novel *item-concept* to cast positions into an abstract form.
- LEDA contains a comfortable data type graph. It offers the standard iterations such as "for all nodes v of a graph G do" (written **forall_nodes** (v, G)) or "for all neighbors w of v do" (written **forall_adj_nodes** (w, v)), it allows the addition and deletion of nodes and edges and it offers arrays and matrices

indexed by nodes and edges, ... The data type graph allows to write programs for graph problems in a form close to the typical text book presentation. We emphasize that all examples given in this paper show executable code. The goal is the equation "Algorithm + LEDA = Program".

- LEDA is realized in C++ and all its data types and algorithms are stored in the library as precompiled object-modules. Together with the fact that the high expressive power of LEDA keeps application programs short this leads to short compile times.
- LEDA offers an interface to the X11-window system so as to allow graphical output and mouse input.
- Many geometric algorithms use arbitrary precision arithmetic and are therefore free from failures due to rounding errors. Moreover, they can handle all degenerate cases.
- 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.

In this paper we describe the status of the project. We hope that LEDA will narrow the gap between algorithms research, teaching, and application. Other projects with similar goals are described in ([GOP90, Boo87].

The structure of this paper is as follows. Section 2 gives an overview of LEDA by way of four examples, section 3 discusses the scope of the library, section 4 discusses its efficiency, and section 5 reports about experiences and gives pointers to further information.

LEDA is available by anonymous ftp (ftp.mpi-sb.mpg.de, directory pub/LEDA) and can be used freely for purposes of research and teaching. The library can be used with any C++ compiler supporting templates (e.g. cfront 3.0 and g++-2.5). Further information can be obtained from leda@mpi-sb.mpg.de.

2. Elegance and Ease of Use

We give an overview of LEDA by way of four examples. They examplify different parts of LEDA: data structures, graphs, geometry, and graphics. All examples demonstrate how easily an algorithm can be translated into a running LEDA-program. The examples also demonstrate that LEDA is a platform on which applications can be built and not only a library of data types and algorithms.

<u>Word Count:</u> Here our task is to read a sequence of strings from the standard input, to count the number of occurrences of each string in the input, and to print a list of all occuring strings together with their frequences on the standard output. The appropriate LEDA types to use are strings and dictionary arrays. The parametrized data type dictionary array ($\mathbf{d\_array}\langle \mathbf{I}, \mathbf{E}\rangle$) realizes arrays with index type $\mathbf{I}$ and

element type **E**. We use it with index type **string** and element type **int**. The full program is given in Figure 1. It starts with the include statement for dictionary arrays. 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_array**s stores the non-zero entries in a balanced search tree with key type string.) In the second line we define a string s. The third line does all the work. The expression $(cin \gg s)$ returns true if the input stream is non-empty and false otherwise. In the former case the first string is removed from the input stream and assigned to s. Then the entry $N[s]$ of array N is increased 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.

```
#include <LEDA/d_array.h>
  main( )
  {
    d_array<string, int> N(0);
    string s;
    while (cin >> s) N[s]++;
    forall_defined (s, N) cout << s << " " << N[s] << "\n";
  }
```

Figure 1: A program that counts the number of occurrences of each string in a sequence of strings.

Shortest Paths in Graphs: A directed graph consists of a set V of nodes and a set E of edges. An edge $e = (v, w)$ is a directed connection from its source node v to its target node w. Assume also that for each edge e we are given the travel time $cost[e]$ in minutes across the edge and that our task is to compute the minimum travel time from a specified node to any other node of the graph, cf. Figure 2 for an example.

Dijkstra [Dij59] found a simple algorithm to solve this problem. At a general step of the algorithm there will be some set $S \subseteq V$ of nodes for which the exact minimum travel time is known whilst for all nodes not in S only a feasible travel time is known. For any node v, we use $dist[v]$ to denote the smallest travel time from s to v currently known. Initially, $dist[s] = 0$, $dist[v] = \infty$ for any $v \neq s$, and $S = \emptyset$. At every step one selects a vertex $u \in V \setminus S$ with smallest value of $dist[u]$ (initially, $u = s$) and adds u to S. For any edge $e = (u, w)$ emanating from u we decrease $dist[v]$ to $dist[u] + cost[e]$ if this value is smaller than the current value of $dist[v]$. In our example, the first node added to S is s. When s is added $dist[a]$ is set to 2 and $dist[c]$ is set to 6. Then a is added to S and $dist[b]$ is set to 3 and $dist[c]$ is reduced to 5, ... A proof of correctness of Dijkstra's algorithm is beyond the scope of this paper (the idea is to show that for all nodes v the label $dist[v]$ is always the shortest travel time from s to v along a path where all but the last node belong to S). How can one find the

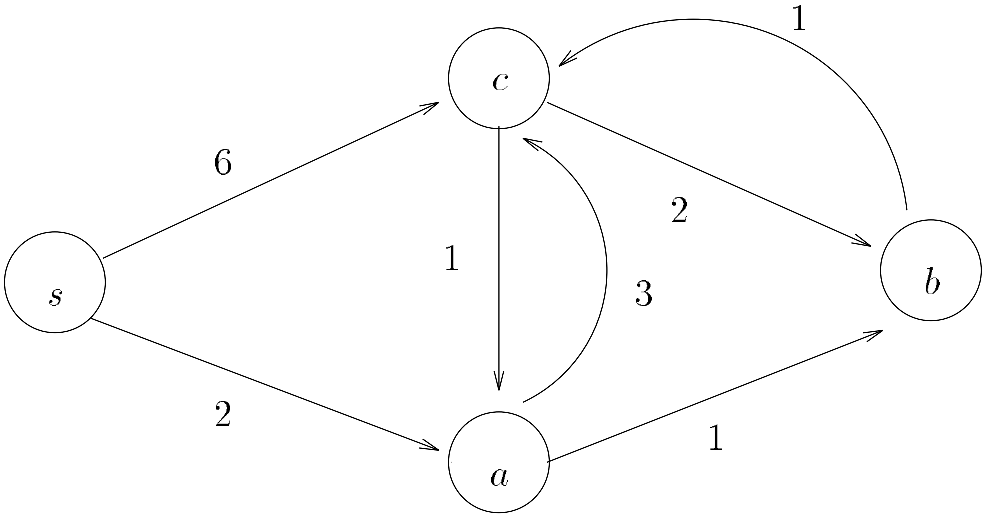

Figure 2: A graph with edge labels. The edge labels indicate the travel time (in minutes) across the edges. The shortest travel time from node s to node c is 4 minutes.

node in $V \setminus S$ with minimal *dist*-value efficiently? The appropriate data structure is a priority queue. Figure 3 shows the LEDA-implementation of Dijkstra's algorithm. Besides graphs it uses the data types **node_array**, **edge_array** and node priority queue (**node_pq**). Node-arrays and edge-arrays are arrays indexed by nodes and edges respectively. We use an **edge_array**$\langle$**int**$\rangle$ *cost* to store the travel times across edges, a **node_array**$\langle$**int**$\rangle$ *dist* to store the minimum travel times to all nodes, and a **node_pq**$\langle$**int**$\rangle$ $PQ(G)$. The parameter G in the definition of PQ tells LEDA that the underlying graph is G. The node priority queue PQ always contains the nodes of G in $V \setminus S$ together with their current integer *dist*-values. Initially, all nodes are inserted into PQ. At every iteration the node with smallest associated value is deleted from PQ ($u = PQ.del\_min(\,)$). All edges e emanating from u (**forall_adj_edges** (e, u)) are scanned and for each such edge the *dist*-value of the target node is reduced if appropriate ($PQ.decrease\_inf(v, c)$).

We want to stress the strong similarity of the LEDA-program and the description of the algorithm given above. The same similarity holds for most graph algorithms. In this sense, LEDA realizes the equation "Algorithm + LEDA = Program".

We want to stress another fact. An implementor of Dijkstra's algorithm does not need to know about the inner workings of graphs and node priority queues; a knowledge of the relevant manual pages suffices. Figure 4 shows the manual page for node priority queues. In its last section the manual page lists the execution times for the various operations on priority queues: constant time for *insert*, *empty* and *decrease_inf*, and logarithmic time for *del_min*. In a graph with n nodes and m edges, Dijkstra's algorithm requires n *insert*s, *empty*-tests, and *del_min*s and at most m *decrease_inf*s. Its running time is therefore proportional to $m + n \log n$.

Convex Hulls: Imagine a finite set L of points in the plane enclosed by a rubber

```
#include <LEDA/graph.h>
#include <LEDA/node_pq.h>
  void DIJKSTRA(const graph &G, node s, const edge_array<int>
          &cost, node_array<int> &dist)
  {
    node_pq<int> PQ(G);
    node v;
    edge e;
    forall_nodes (v, G) {
      if (v ≡ s) dist[v] = 0; else dist[v] = MAX_INT;
      PQ.insert(v, dist[v]);
    }
    while (¬PQ.empty( )) {
      node u = PQ.del_min( );
      forall_adj_edges (e, u) {
        v = target(e);
        int c = dist[u] + cost[e];
        if (c < dist[v]) { PQ.decrease_inf(v, c); dist[v] = c; }
      }
    }
  }
```

Figure 3: Dijkstra's algorithm for the single source shortest path problem

band. The rubber band relaxes to the so-called convex hull of L. Figure 5 shows an example. The convex hull is one of the basic structures of computational geometry. It is frequently used in image processing and pattern recognition as an approximation to the shape of a set of points. We discuss how to compute the convex hull. For simplicity, we restrict ourselves to the so-called upper hull. If one cuts the convex hull at the leftmost and rightmost point of L, the convex hull of L is split into the upper and lower hull of L, cf. Figure 5; here a point p is left of a point q if either the x-coordinate of p is smaller than the x-coordinate of q or the two x-coordinates are equal and the y-coordinate of p is smaller.

Here is how to compute the upper hull of a set L of points. First sort L according to the left-to-right ordering of points. Let $p_1, p_2, \ldots, p_n$ be the sorted sequence of points. We construct the upper hull of the points $p_1, \ldots, p_i$ incrementally for all i, $1 \leq i \leq n$. Initialization is simple, the upper hull of p_1 is p_1. Assume now that we have already constructed the upper hull of $p_1, \ldots, p_i$ and next want to add p_{i+1}. If $p_i = p_{i+1}$ then there is nothing to do. If $p_i \neq p_{i+1}$ then we simply have to delete the last point of the current upper hull as long as the current upper hull has at least two points and the last two points together with the new point does not form a right-turn, cf. Figure 6. We then add p_{i+1} to the upper hull; this completes the update step. Figure 7 shows the LEDA-implementation of this algorithm. Again there is

Node priority queues (node_pq)

1. Definition

An instance Q of the parametrized data type **node_pq**$\langle$**I**$\rangle$ is a set of pairs (v, i), where v is a node of some graph G and i belongs to some linearly ordered type **I**; i is called the information associated with node v. For any node v of G there can be at most one pair $(v, \ )$ in Q.

2. Creation

node_pq$\langle$**I**$\rangle$ $Q(G)$;

creates an empty instance Q of type **node_pq**$\langle$**I**$\rangle$ for the nodes of graph G.

3. Operations

void	$Q.insert($**node** v, **I** $i)$	adds the node v with information i to Q. *Precondition*: There is no pair $(v, \ )$ in Q.
I	$Q.inf($**node** $v)$	returns the information of node v.
bool	$Q.member($**node** $v)$	returns true if (v, i) in Q for some i, false otherwise.
void	$Q.decrease\_inf($**node** v, **I** $i)$	makes i the new information of node v. (*Precondition*: $i \leq Q.inf(v)$).
node	$Q.find\_min()$	returns a node with the minimal information(**nil** if Q is empty).
void	$Q.del($**node** $v)$	removes the pair $(v, \ )$ from Q.
node	$Q.del\_min()$	removes a node with minimal information from Q and returns it (**nil** if Q is empty).
int	$Q.size()$	returns the number of pairs in Q.
void	$Q.clear()$	makes Q the empty node priority queue.
bool	$Q.empty()$	returns true if Q is the empty node priority queue, false otherwise.

4. Implementation

Node priority queues are implemented by fibonacci heaps and node arrays. Operations *insert*, *del_node*, *del_min* take time $O(\log n)$, $find\_min$, $decrease\_inf$, *empty* take time $O(1)$ and *clear* takes time $O(m)$, where m is the size of Q. The space requirement is $O(n)$, where n is the number of nodes of G.

Figure 4: The manual page for node priority queues

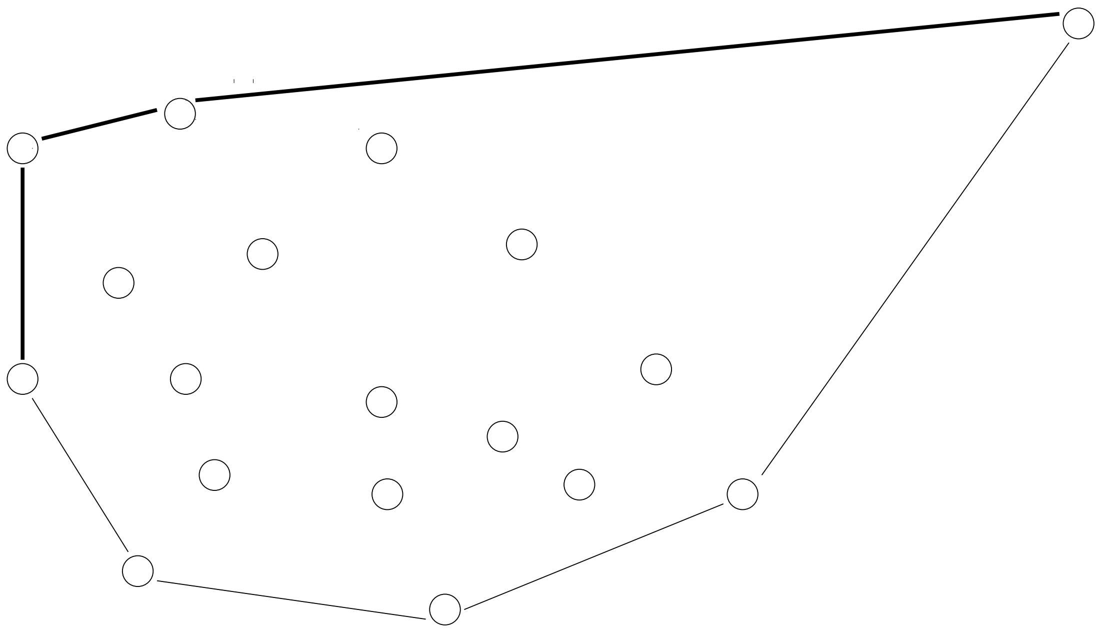

Figure 5: The convex hull of a set of points in the plane. The upper hull is shown in bold.

a striking similarity between the algorithm description and the LEDA program and only a few additional words are required to explain the program: $L.sort(\,)$ sorts the list L according to the default-ordering defined on the elements of the list. For points this is the left-to-right ordering. $L.pop(\,)$ deletes the first element from a list and returns it and $Uh.append(p)$ appends the point p to the list Uh, $L.empty(\,)$ returns true if L is empty, $Uh.length(\,)$ returns the length of list Uh, and $Uh.Pop(\,)$ deletes the last element of list Uh. In LEDA a list is viewed as a sequence of so-called items (type **list_item**) each of which contains an element of the list. $Uh.last(\,)$ returns the last item of the list Uh and for an item it of Uh the content of the item is given by $Uh[it]$ and the predecessor item is given by $Uh.pred(it)$.

There is another interesting fact about the convex hull program. A point in LEDA may have arbitrary rational coordinates and all geometric predicates are evaluated exactly, i.e., with exact rational arithmetic. Also note that the program handles multiple occurrences of the same point and collinear points correctly, situations which are frequently referred to as degeneracies. All geometric algorithms in LEDA are supposed to handle *all* degeneracies and, in fact, many of them already do; we refer the reader to [BMS94a, BMS94b, MN94b] for more details.

Graphics: The LEDA type **window** is an interface to the X11 window system. Figure 8 illustrates this data type. The program reads a sequence of points, displays them, computes their upper hull, and displays the upper hull as a polygonal line. Again only a few explanations are needed. The definition **window** W defines a graphics window and opens it for mouse input. Any click on the left mouse button inputs a point ($W \gg p$), a click on the right mouse button lets the statement $W \gg p$

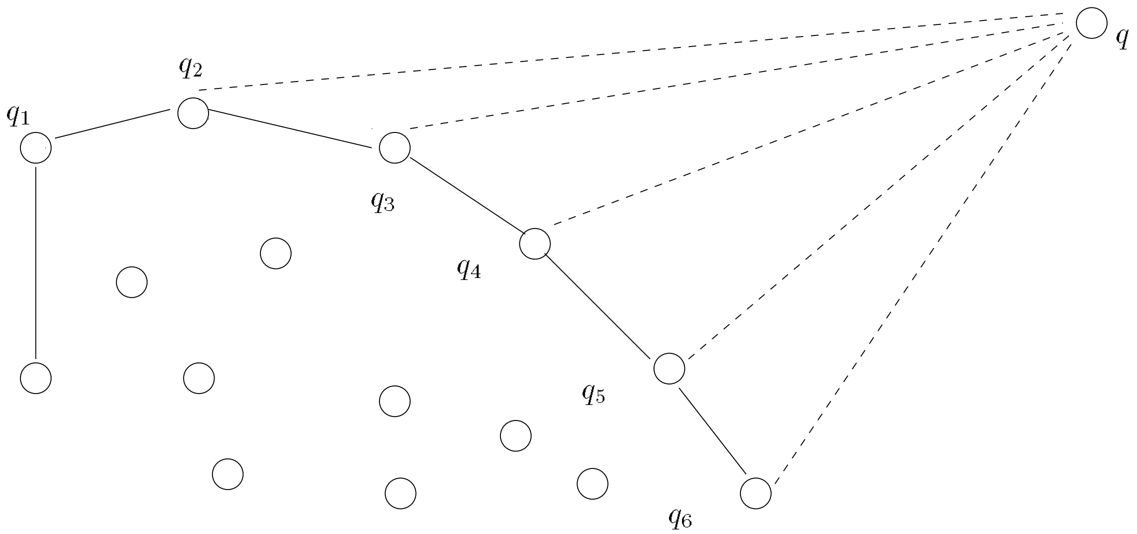

Figure 6: The upper hull after processing all but the rightmost point. When node q is added the last four points of the current hull are deleted since $(q_i,\, q_{i+1},\, q)$ is not a right-turn for $2 \leq i \leq 5$.

evaluate to false. The point is appended to list L and displayed in the window W. Then the upper hull is computed and drawn as a polygonal line. Figure 9 shows an example.

3. Scope

LEDA is a wide collection of data types and algorithms on these data types. In fact, most of the data structures and algorithms described in the text books of the area (e.g., [AHU83, Meh84, Tar83, CLR90, O'R94, Woo93, Sed91, Kin90, Wyk88, NH93]) are contained in LEDA. The library is organized into six logical units: 1. basic data types, 2. numbers, vectors, and matrices, 3. dictionaries and priority queues, 4. graphs, 5. windows and panels, and 6. geometry.

The basic data types are strings, lists, queues, stacks, arrays, partitions, and trees.

The number types are the built-in types int, float and double as well as the arbitrary precision versions Int and Float. The type Int is the class of integers in the mathematical sense and Float ist the class of floating point numbers with arbitrary precision mantissa and exponent. For all four number types vectors and matrices are available.

In the third unit we have priority queues, dictionaries, dictionary- and hashing-arrays, sorted sequences, and persistent dictionaries.

The graph part first of all offers different kinds of graphs: directed graphs, undirected graphs, and planar graphs.In addition, it offers data structures on graphs, e.g., arrays indexed by nodes and edges respectively, priority queues on nodes, and node

```
#include <LEDA/list.h>
#include <LEDA/plane.h>
  list<point> u_hull(list<point> L)
  {
    L.sort();       //into left-to-right order
    list<point> Uh;
    point p = L.pop();
    Uh.append(p);
    while (¬L.empty()) {
      point q = L.pop();      // pop deletes the first element from L
      if (p ≡ q) continue;
      list_item it;
      while ((Uh.length() ≥ 2) ∧ ¬right_turn(Uh[Uh.pred(it = Uh.last())],
              Uh[it], q)) Uh.Pop();      // Pop deletes the last element from Uh
      Uh.append(q);
      p = q;
    }
    return Uh;
  }
```

Figure 7: A program to compute the upper hull of a set of points

partitions, and many algorithms on graphs and networks: shortest paths, biconnected and strongly connected components, transitive closure, topological sorting, unweighted and weighted bipartite matching, unweighted general matching, network flow, min cost network flow, planarity testing, planar embedding, minimum spanning tree, etc.

The windows and panels part offers an interface to the X11-windows system. It can be used to display geometric objects and to construct interactive applications with mouse-input.

The section on geometry offers points, segments, lines, data structures on these objects (e.g. planar subdivisions and range-, segment-, and interval-trees) and some geometric algorithms (e.g. line segment intersection, Voronoi diagrams and Delaunay triangulations, and convex hulls in arbitrary dimensions).

4. Efficiency

We discuss run-time and compile-time efficiency.

All data types and algorithms in LEDA are precompiled and stored in libraries. An application program only has to include the header files of all the data types used in the application. The header files are typically short since they only include the declarations of all the member functions of the type and only very small sections of

```
#include <LEDA/window.h>
  list⟨point⟩ u_hull(list⟨point⟩);
  main( )
  {
    window W;
    list⟨point⟩ L;
    point p;
    while (W ≫ p) { L.append(p); W.draw_point(p); }
    list⟨point⟩ Uh = u_hull(L);
    p = Uh.pop( );
    while (¬Uh.empty( )) { W.draw_segment(p, Uh.head( )); p = Uh.pop( ); }
  }
```

Figure 8: A program to illustrate the upper hull algorithm and the interface to X-windows

actual code (for inline functions and for type conversion). The application programs are typically short since LEDA allows them to be formulated on a very high level. Both factors together lead to short compilation times. Only the linker has to search through the large LEDA library (about 3 MBytes).

Let us turn to run-time efficiency next. All data types in LEDA are realized by the asymptotically most efficient implementation known. For many data types we have even included several implementations, e.g., for dictionaries the user can choose between ab-trees, AVL-trees, BB[a]-trees, red-black trees, skip lists, and randomized search trees. The mechanism for choosing another implementation is quite convenient. Replacing the definition of dictionary array N in the word count program by
_d_array⟨**string**, **int**, *skip_list*⟩ $N(0)$;
tells LEDA to use the skip-list implementation.

Abstract data types hide the implementation details of a data structure, but the abstraction, if improperly done, might bring about a loss of efficiency. Here is an example. A dictionary with key type **K** and information type **I** is usually viewed as a mapping from **K** to **I**. Suppose now that one first wants to access the information associated with some key k, to modify it next, and finally to associate a new information with key k. In the traditional definition of dictionary this involves *two* searches: the first search locates the key k and the associated information and the second search locates k again and associates a new information with k. However, a direct use of the data structure can avoid the second search. One can simply keep a pointer to the position of key k in the data structure and replace the second search by direct pointer access. In LEDA we introduced a novel *item concept* to overcome this inefficiency. A dictionary is viewed as a collection of items (type **dic_item**) each containing a key and an information. The keys associated with different items are distinct. A lookup in a dictionary takes a key and returns an item (not an information). The information can then be accessed through the item. More importantly,

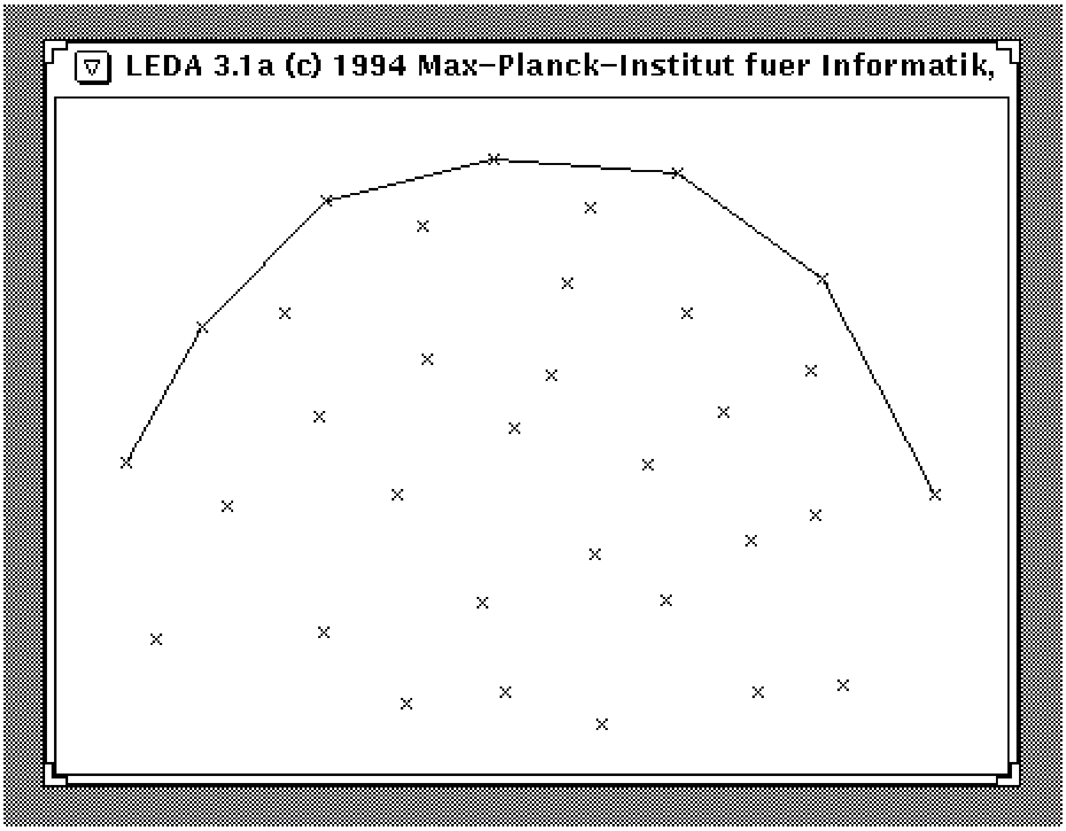

Figure 9: An output of the upper hull program

the item can be stored and a later access can be made directly through the item. In this way, items are the abstraction of a position in a data structure. They give the same efficiency and nevertheless allow complete encapsulation of the underlying data structure. We feel that LEDA's item concept is a key factor in its efficiency.

It is clear that some penalty has to be paid for the generality of LEDA. U. Lauther [Lau92] has carried out an extensive comparison between graph and network algorithms implemented in LEDA and hand-coded C-versions. He reports that the LEDA-Versions are slower by a factor between 2 and 10 (typically by a factor of 4) and use about 2 to 3.5 times the storage of his versions. We believe that this is quite acceptable given the convenience that LEDA offers. We should mention that we reimplemented some of the basic data structures (most notably graphs) as a reaction to Lauther's report. This reduced the typical slowdown to about 2.

5. Experiences and Conclusion

The work on LEDA was started in 1989 and a first version of LEDA was made available for outside use in 1990. Since fall 1992 LEDA version 3.0 is available through anonymous ftp. Version 3.1 underlying this report will be released in the summer of 1994; it corrects bugs, provides a more efficient graph data type, and contains robust implementations of some geometric algorithms. Currently most users of LEDA are in universities and research institutions. Sample applications are (according to a survey conducted in spring 94): code optimization, motion planning, logic synthesis, scheduling, VLSI design, term rewriting systems, semantic nets, machine learning, image analysis, computational biology, and many others.

LEDA is not the only library of data structures around. Other libraries are NIHL [GOP90], Booch components [Boo87], cf. [Loc94] for a survey. The main feature that distinguishes LEDA from the other libraries is its scope. No other library covers so much of combinatorial and geometric computing.

We close with some pointers to further literature: [Näh93] is the LEDA Manual. The implementation of LEDA, in particular the realization of parametrized types and of implementation parameters, is discussed in [MN92]. The implementation of geometric algorithms is discussed in [MN94b, BMS94a, BMS94b]. Case studies of algorithms implemented in LEDA can be found in [MN94a, MMN93, MZ93]. All papers mentioned above are available through anonymous ftp (ftp.mpi-sb.mpg.de, directory pub/LEDA/articles).

References

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[BMS94a] Ch. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. SODA 94*, pages 16–23, 1994.

[BMS94b] Ch. Burnikel, K. Mehlhorn, and St. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *LNCS*, 1994. Proceedings of ESA'94.

[Boo87] G. Booch. *Software Components with Ada.* Benjamin/Cummings Publishing Company, 1987.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill Book Company, 1990.

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

[GOP90] K.E. Gorlen, S.M. Orlow, and P.S. Plexico. *Data Abstraction and object-oriented programming in C++.* John Wiley and Sons Ltd., 1990.

[Kin90] J.H. Kingston. *Algorithms and Data Structures.* Addison-Wesley Publishing Company, 1990.

[Lau92] U. Lauther. Untersuchung der library of efficient data types and algorithms (LEDA). Technical report, Siemens AG, ZFE, München, 1992.

[Loc94] N. Locke. C++ FTP Libraries. *C++ Report*, 6(2):61–65, 1994.

[Meh84] K. Mehlhorn. *Data Structures and Efficient Algorithms.* Springer Verlag, 1984.

[MMN93] K. Mehlhorn, P. Mutzel, and St. Näher. An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Technical Report MPI–I–93–151, Max–Planck–Institut für Informatik, Saarbrücken, 1993.

[MN92] K. Mehlhorn and St. Näher. Algorithm design and software libraries: Recent developments in the leda project. In *Algorithms, Software, Architectures, Information Processing 92*, volume 1. Elsevier Science Publishers B.V., 1992.

[MN94a] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[MN94b] K. Mehlhorn and St. Näher. The implementation of geometric algorithms. 1994. IFIP94, to appear.

[MZ93] M. Müller and J. Ziegler. An implementation of a convex hull algorithm. Technical Report MPI-I-94–105, Max-Planck-Institut für Informatik, Saarbrücken, 1993.

[Näh93] St. Näher. LEDA manual. Technical Report MPI-I-93-109, Max-Planck-Institut für Informatik, 1993.

[NH93] J. Nievergelt and K.H. Hinrichs. *Algorithms and Data Structures.* Prentice Hall Inc., 1993.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[Sed91] R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, 1991.

[Tar83] R.E. Tarjan. *Data Structures and Network Algorithms*, volume 44. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.

[Wyk88] C.J. van Wyk. *Data Structures and C programs.* Addison-Wesley Publishing Company, 1988.

[Woo93] D. Wood. *Data Structures, Algorithms, and Performance.* Addison-Wesley Publishing Company, 1993.

Index