LEDA

A Library of Efficient Data Types and Algorithms *

Kurt Mehlhorn and Stefan Näher

A 04/89

Fachbereich Informatik, Universität des Saarlandes
D-6600 Saarbrücken, Federal Republic of Germany

Abstract

LEDA is a library of efficient data types and algorithms. At present, its strength is graph algorithms and r lated data structures. The computational geometry part is evolving. The main features of the library are
1) a clear separation of specification and implementation
2) parameterized data types
3) a comfortable data type graph,
4) its extendibility, and
5) its ease of use.

* This research was supported by the ESPRIT II Basic Research Action Program, ESPRIT P. 3075 – ALCOM, and by the DFG, grant SPP Me 620/6-1.

I. Introduction

There is no standard library of the data structures and algorithms of combinatorial 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 (MACSYMA, SAC-2) and linear programming (MPSX).

In fact the situation is worse, since even within small groups, say the algorithms group at our home institution, software frequently is not shared. Rather, each researcher starts from scratch and e.g. develops his own version of a balanced tree. Of course, this continuous "reimplementation of the wheel" slows down progress, within research and even more so outside. This is due to the fact that outside research the investment for implementing an efficient solution frequently is not made, because it is doubtful whether the implementation can be reused, and therefore methods which are known to be less efficient are used instead. Thus the scientific discoveries migrate only slowly into practice.

One of the major differences between combinatorial 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 computing relies heavily on types like stacks, queues, dictionaries, sequences, sorted sequences, priority queues, graphs, points, planes, ...

One year ago, we started a project (called LEDA for Library of Efficient Data types and Algorithms) to build a small, but growing library of data types and algorithms in a form which allows them to be used by non-experts. We hope that the system will narrow the gap between algorithms research, teaching, and implementation. In this paper we report on our difficulties and achievements. The main features of the library are:

1) A clear separation between (abstract) data types and the data structures used to implement them, cf. section II. This distinction frequently is not made in the combinatorial algorithms literature, but is crucial for a library. Note that we stated above that each researcher implemented his own version of a balanced tree, i.e., a data structure, and not his own version of a dictionary, i.e., a data type. The data types currently available are stack, queue, list, set, dictionary, ordered sequence, priority queue, directed graph and undirected graph. The most difficult decision, which we faced, was the treatment of positions and pointers. For the efficiency of many data structures it is crucial that some operations on the data structure take positions in (= pointers into) the data structure as arguments. We have chosen an item concept to cast the notion of position into an abstract form, cf. section II.
2) Polymorphic types, i.e., types with parameters, cf. section II. Most of our data types have type parameters. For example, a dictionary has a key type K and an information type I and a specific dictionary type is obtained by setting, say, K to **int** and I to **real**.
3) 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 to add and delete vertices and edges and it offers arrays and matrices indexed by nodes and edges,..., cf. section III for details. The data type graph allows to write programs for graph problems in a form close to the way the algorithms are usually presented in text books. Section VIII contains a list of the algorithms which are currently in LEDA.

4) The LEDA library is extendible. Users can include own data types either by writing C++ programs (cf. section V) or by combining already existing LEDA data types (cf. section VI).
5) Ease of use: LEDA is written in C++ (cfront version 1.2.1). All data types and algorithms are precompiled modules which can be linked with application programs. All examples given in this paper show executable code.

This paper is organized as follows. In section II we discuss data types and data structures, in section III the data type graph and in section IV we discuss the interaction of graphs and data types. In section V we briefly describe the internal structure of LEDA. Section VI shows how the library can be extended by users and in section VII we report about our experiences with designing, implementing and using LEDA.

The design of LEDA is joint work by the two authors, the implementation was mostly done by the second author. LEDA is available from the authors for a handling charge of DM 100.

II. Data Types

One of the major differences between combinatorial computing and other areas of computing such as statistics, numerical analysis and linear programming is the usage of complex data types. Whilst the built-in types, such as integers, reals, vectors, and matrices, usually suffice in the other areas, combinatorial computing heavily relies on types like stacks, queues, dictionaries, sequences, sorted sequences, priority queues, graphs, points, planes, ... Programming languages do not provide these types and hence they have to be defined and implemented by the individual user. A significant improvement would result, if these data types were made available as "packages" by some users and could then be reused by others. Of course, this makes it necessary to specify the properties of the data type independently of its implementation. We next discuss the LEDA specification of the data types dictionary, priority queue and partition.

Example 1, Dictionary: A dictionary can be defined as a partial function with finite domain from some universe K of keys into some set I of informations associated with the keys. An access operation looks up the function value at an argument, an insert extends the domain of definition, ... Thus a clean definition of the data type dictionary is available and hence a dictionary module should cause no principal difficulties. In fact, the programming language Comskee [BKMRS84] includes the dictionary among its built-in data types. A (minor) problem lies in the fact that most programming languages force the user to edit source text when he wants to change the key or information type. This problem is resolved in languages with polymorphic data types such as Clu, Ada, or C++ Program 1 shows a small example using the above defined dictionary type.

The program reads a sequence of integers and counts the number of occurrences of each integer in the sequence. The number of occurrences is then listed for each integer in the sequence. The details are as follows. In line (1) the dictionary type dictionary(int,int) is defined. Note that the dictionary module provides dictionaries from K to I where K and I are type variables. A specific dictionary type is introduced as shown in line (1). Line (3) introduces D as the name of an object of type dictionary(int,int); D is initialized with the empty function (by a C++ constructor). Line (5) to (9) step through the input sequence. In

```
(1) declare2(dictionary,int,int);        //K = I =int
(2) main()
(3) {  dictionary(int,int) D;
(4)    int k;
(5)    while (cin >> k)
(6)    {  if (D.defined(k))
(7)          D.change_inf(k,D.lookup(k)+1);
(8)       else D.insert(k,1);
(9)    }
(10)   forall_keys(k,D) cout << k << " " << D.lookup(k) << "\n";
(11) }
```

Program 1: Counting the number of occurrences of each element of a sequence of integers

Program 1

line (6) we test whether k belongs to the domain of D and then either increase the value at argument k by one or insert the pair $(k,1)$ into the dictionary. Finally, line (10) steps through all keys k in the domain of D and prints k and the value of D at k.

Remark: The above specified dictionary type allows only access by key. It is not possible to store the positions or locations of keys and informations in the dictionary for later use. Thus in program 1 three access operations are necessary in lines 6 and 7 to change the information associated with a key. The LEDA library (version 1.0) contains a different dictionary type (see LEDA1.0 User Manual). It allows access by position as discussed in the next examples for priority queues and partitions.

LEDA is written in C++ which is an extension of the C programming language. In addition to the facilities of C, C++ provides data abstraction, data hiding, object oriented programming, operator overloading and many other features suitable for implementing abstract data types. For details of the C++ language see "The C++ Programming Language" [St86]. Every data type in LEDA is implemented as a C++ class. A class consists of private data and a collection of functions which can be applied to the data. We use these so called member functions of the class to realize the operations on instances of the corresponding data type. The known syntax for record field access is used to invoke such a functions, e.g., if D is an object (variable) of a class with a member function $f()$ then $D.f()$ calls f to operate on D.

Example 2, Priority Queues: Priority queues are frequently used in network algorithms; cf. section III for an example. It is tempting to define a priority queue as a dictionary with the additional operations *findmin* and *decrease_inf* (We require a linear order on I). The operation PQ.findmin() returns the element v in the domain of PQ with minimal function value and PQ.decrease_inf(v,i) decreases the function value at argument v to i. The operation raises an error if i is larger than the old value at argument v. It is clear that a decrease_inf operation involves a lookup and hence must be at least as costly as a lookup operation in a dictionary. However, several recent papers, e.g. [FT84] and [AMOT88] show that the decrease_inf operation can be realized in time $O(1)$ if its first argument is the *position* of the pair $(v, PQ(v))$ in the data structure realizing the priority queue PQ. This led to improvements in several network algorithms.

We conclude that the argument of a decrease_inf operation cannot be the key v. But what should it be? It cannot be the position in a data structure since we want to clearly separate data type and data structure and therefore cannot use a notion of the particular implementation in the definition of the data type. What we need is an abstraction of position.

In LEDA, we call an abstract position an *item*. For most data types XYZ there is also a type XYZ*_item*. This type represents the set of positions in objects of data type XYZ. So, together with the data type *priority_queue* there is also a type *priority_queue_item* or briefly *pq_item*. Each *pq_item* contains or has associated with it a pair (v, i) with $v \in K$ and $i \in I$. We use $< v, i >$ to denote the item which contains the pair (v, i). A priority queue PQ over K and I is then nothing but a collection of *pq_items* where each item contains a pair (v, i).

Remark 1: In mathematical terms a priority queue in LEDA is simply a partial function from the set of pq_items to the cartesian product $K \times I$. In computer science terminology, we may view a *pq_item* as the name ($\hat{=}$ address) of a container holding an argument-value pair (v, i).

Remark 2: Access by position can now be achieved as follows. When we insert a pair (v, i) into a priority queue PQ, the operation PQ.insert(v, i) returns an item, say *it*, i.e. the name of a container which holds the pair (v, i). The user of the priority queue remembers this name and can now access the pair (v, i) in two different ways. He can either access it through the item *it* which was returned by the insert (this is access by position) or through the key v (this is access by key). There is a crucial difference between the two modes of access. In the latter case, the key v identifying the pair (v, i) was provided by the user and hence access involves a dictionary look-up, in the former case, the name *it* identifying the pair (v, i) was produced by the data type and therefore can give direct access. The decrease_inf operation uses access by position, i.e. whenever the user wants to decrease the value associated with v, he calls PQ.decrease_inf and gives it the name of the container as an argument. In this way, the item concept captures the notion of access by position but is nevertheless independent of implementation.

The complete LEDA specification of the data type priority queue follows.

Priority Queues (priority_queue)

An instance Q of the data type *priority_queue* is a collection of items (*pq_item*). Every item contains a key from a type K and an information from a type I. K is called the key type of Q and I is called the information type of Q. The number of items in Q is called the size of Q. If Q has size zero it is called the empty priority queue. We use $< k, i >$ to denote the *pq_item* with key k and information i. There must exist a linear order on I.

1. Declaration of a priority queue type

declare2(*priority_queue*, K, I)

introduces a new data type with name *priority_queue*(K, I) consisting of all priority queues with key type K and information type I.

2. Creation of a priority queue

priority_queue(K, I) Q;

creates an instance Q of type *priority_queue*(K, I) and initializes it with the empty priority queue.

3. Operations on a priority_queue instance Q

K	Q.key(*pq_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in Q.
I	Q.inf(*pq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in Q.
pq_item	Q.insert(K k, I i)	adds a new item $< k, i >$ to Q and returns *it*.
pq_item	Q.find_min()	returns the item with minimal information (nil if Q is empty)
void	Q.del_item(*pq_item it*)	removes the item *it* from Q. *Precondition*: *it* is an item in Q.
K	Q.del_min()	removes the item with minimal information from Q and returns its key. *Precondition*: Q is not empty.
pq_item	Q.decrease_inf(*pq_item it*, I i)	makes i the new information of item *it* *Precondition*: *it* is an item in Q and i is not larger then $inf(it)$.
void	Q.clear()	makes Q the empty priority queue
bool	Q.empty()	returns true, if Q is empty, false otherwise
int	Q.size()	returns the size of Q.

4. Implementation

Priority queues are implemented by Fibonacci Heaps. Operations insert, del_item, del_min take time $O(\log n)$, find_min, decrease_inf, key, inf, empty take time $O(1)$ and clear takes time $O(n)$, where n is the size of Q. The space requirement is $O(n)$.

Example 3, Partitions or Disjoint Sets: In this example we discuss partitions of finite sets. An application of partitions is shown in program 5 of section IV. Partitions are also used to discuss some implementation details of LEDA in section V.

An object P of type *partition* consists of a finite set of partition_items and a partition of this set into blocks. The declaration *partition P* introduces P as the name of a partition and initializes it to the empty partition. There are three operations.

partition_item	*P.make_block*()	returns a new partition_item *it* and adds the block $\{it\}$ to the partition P.
int	*P.same_block*(*partition_item p, q*)	returns true if p and q belong to the same block of the partition P.
void	*P.union_blocks*(*partition_item p, q*)	unites the blocks of P containing items p and q.

We want to stress that the make_block-operation has no parameter. It is not given an object which it is supposed to add to the partition P (this would imply access by key lateron), but the operation itself chooses an item *it*, returns it and adds the block $\{it\}$ to the partition P. The user has no idea what *it* is and he does not need to know. The only thing, that is important to him, is the partition of the items into blocks. This usage of items is similar to the usage of atoms in SETL. ∎

Access by position instead of key is abundant in the design of efficient data structures; e.g. lists are accessed by position, all operations in the union-find problem use access by position, the decrease-inf operation in priority queues is based on it and fingers in finger trees are positions. Thus the position concept is crucial for the design of efficient data structures. In LEDA we use items as an abstraction of positions. Many operations take items as arguments or return items as results. The undefined item *nil* is often returned to indicate that an operation failed, e.g., a lookup operation in a dictionary returns *nil* if the search key is not present. The examples above show that the item concept leads to natural specifications of data types.

III. Graphs

Graph algorithms are a prime example of combinatorial computing. LEDA contains several graph types and data types related with graphs allowing elegant and efficient implementations of graph and network algorithms.

```
#include <LEDA/graph.h>
bool ACYCLIC(graph G)
{ // Tests if G is acyclic by repeatedly deleting edges starting in nodes with indeg 0.
  node_set zero; // Set of all nodes v with indeg(v) = 0
  node v, w;
  forall_nodes(v, G) if (G.indeg(v)==0) zero.insert(v);
  while ( !zero.empty() )
   { v = zero.choose();
     zero.del(v);
     forall_adj_edges(e, v)
       { w = G.target(e);
         G.del_edge(e);
         if (G.indeg(w)==0) zero.insert(w);
       }
   }
  return G.number_of_edges() == 0;
}
```

Program 2: Testing a directed graph for acyclicity.

Program 2

graph is the data type of directed graphs. It provides operations for the updating (inserting and deleting nodes and edges) and accessing internal informations (number of nodes or edges, out- and indegree of nodes, endpoints of edges, list of edges adjacent to a given node, ...) of directed graphs. Furthermore there are several iteration statements that can be used to iterate over the nodes and edges (**for_all_nodes, for_all_edges**, ...). Program 2 gives a short demonstration of some graph operations used to test a directed graph G for acyclicity. The algorithm uses the data types *graph* and *node_set* whose specifications are contained in the header file "LEDA/graph.h" which is included in line 1. In line 4 a node set *zero* for the nodes of graph G is declared. It is initialized in line 6 with all nodes of indegree 0. In lines 7-15 the algorithm repeatedly deletes all edges starting in nodes of *zero* and adds the new nodes with indegree 0 to *zero*. G is acyclic if all edges are removed in the end.

Many graph algorithms, especially network algorithms, use additional informations associated with the nodes and edges (node labels, edge costs, ...). LEDA provides two ways for associating informations with the nodes and edges of graphs:

1. **Parameterized Graphs**
   A parameterized graph G is a graph whose nodes and edges contain additional (user defined) informations. Every node contains an element of a data type *vtype*, called the node type of G and every edge contains an element of a data type *etype* called the edge type of G. All operations defined on instances of the data type *graph* are also defined on instances of any parameterized graph type $GRAPH(vtype, etype)$. For parameterized graphs there are additional operations to access or update the informations associated with its nodes and edges.

   Instances of a parameterized graph type can be used wherever an instance of the data type *graph* can be used, e.g., in assignments and as arguments to functions with formal parameters of type *graph* or *graph&*. If a function $f(graph\& \ G)$ is called with an argument Q of type $GRAPH(vtype, etype)$ then inside f only the basic graph structure of Q (the adjacency lists) can be accessed. The node and edge informations are hidden. For example, function ACYCLIC accepts instances of any parameterized graph type as argument.

2. **Node and Edge Arrays**
   node_array (*edge_array*) is a polymorphic data type. An instance of *node_array*(XYZ) (*edge_array*(XYZ)) is an array which is indexed by the nodes (edges) of some graph and whose entries are values of type XYZ. Thus a node (edge) array is a mapping from the nodes (edges) of graph into a set of elements of type XYZ.

   Node (edge) arrays allow the passing of node and edge informations of networks to algorithms separatedly from its basic graph structure. In this way reusable network algorithms accepting instances of arbitrary graph types as arguments can be designed.

Examples for reusable network algorithms are the following programs DIJSKTRA (single source shortest paths) and MST (minimum spanning tree). We use them to illustrate LEDA's comfortable graph type and its interaction with other data types. Program 3 shows Dijkstra's algorithm (cf. [AHU83], [M84,section IV.7.2], [T83]) for the single source shortest path problem in digraphs with non-negative edge costs. The algorithm uses the data types graph and priority queue (lines (1) and (2)). The input to the algorithm is a graph G, a node s of G and a non-negative cost for each edge. It returns for each node v the length of a shortest path from s to v (array *dist*) and the last edge on such a shortest path (array *pred*). In LEDA we use edge- and node-arrays for the latter three parameters. A *node_array(edge)* is a mapping from nodes to edges. The algorithm maintains for each node v a temporary distance label $dist[v]$. Initially, $dist[s] = 0$ and $dist[v] = \infty$ for $v \neq s$, cf. lines (13)–(19). In LEDA the loop $forall\_nodes(v, G)\{\ldots\}$ can be used to iterate over all nodes v of a graph G.

```
(1)   #include <LEDA/graph.h>
(2)   #include <LEDA/prio.h>
(3)   declare2(priority_queue,node,int)
(4)   declare(node_array,pq_item)
(5)   void DIJKSTRA(graph& G, node s, edge_array(int)& cost,
(6)                         node_array(int)& dist, node_array(edge)& pred )
(7)   { priority_queue(node,int) PQ;
(8)     node_array(pq_item) I(G, nil);
(9)     pq_item it;
(10)    int c;
(11)    node u, v;
(12)    edge e;
(13)    forall_nodes(v, G)
(14)    { pred[v] = 0;
(15)      dist[v] = infinity;
(16)      I[v] = PQ.insert(v, dist[v]);
(17)    }
(18)    dist[s] = 0;
(19)    PQ.decrease_inf(I[s], 0);
(20)    while (!PQ.empty())
(21)    { it = PQ.delete_min()
(22)      u = PQ.key(it);
(23)      forall_adj_edges(e, u)
(24)      { v = G.target(e);
(25)        c = dist[u] + cost[e];
(26)        if ( c < dist[v])
(27)        { dist[v] = c;
(28)          pred[v] = e;
(29)          PQ.decrease_inf(I[v], c);
(30)        }
(31)      }
(32)    } // while
(33)  }
```

Program 3: Dijkstra's algorithm

Program 3

Dijkstra's algorithm uses a priority queue PQ. The priority queue contains pairs $(v, dist[v])$

and hence has type *priority_queue(node, int)*; cf. lines (3) and (7). Each node v of the graph needs to know the position of the item $< v, dist[v] >$ in the priority queue. We therefore declare the data type *node_array(pq_item)* in line (4) and declare *node_array(pq_item)* $I(G, nil)$ in line (8). In this declaration the parameter G tells LEDA that we want an array which is indexed by the nodes of G and the second parameter tells it that we want all entries initialized to the *pq_item* **nil**.

Initially, the items $< s, 0 >$ and $< v, infinity >$ for $v \neq s$ are put into PQ, cf. line (16). Then in each iteration we select and delete an item *it* with minimal inf from PQ, cf. line (21). Let $it =< u, dist[u] >$, cf. line (22). We now iterate through all edges e starting in edge u; cf. line (23). Let $e = (u, v)$ and let $c = dist[u] + cost[e]$ be the cost of reaching v through edge e, cf. lines (24) and (25). If c is smaller than the temporary distance label $dist[v]$ of v then we change $dist[v]$ to c and record e as the new predecessor of v and decrease the information associated with v in the priority queue., cf. lines (26) to (29).

```
#include <LEDA/graph.h>
void all_pairs_shortest_paths(graph& G, edge_array(int)& cost,
                                                  node_matrix(int)& DIST)
{
  // computes for every node pair (v, w) DIST[v][w] = cost of the least cost
  // path from v to w, the single source shortest paths algorithms BELLMAN_FORD
  // and DIJKSTRA are used as subroutines
  edge e;
  node v;
  int C = 0;
  forall_edges(e, G) C+ = cost[e];
  node s = G.new_node();                 // add s to G
  forall_nodes(v, G) G.new_edge(s, v);   // add edges (s, v) to G
  node_array(int) dist1(G);
  node_array(edge) pred(G);
  edge_array(int) cost1(G);
  forall_edges(e, G) cost1[e] = (G.source(e) == s)?C : cost[e];
  BELLMAN_FORD(G, s, cost1, dist1, pred);
  G.del_node(s);                         // delete s from G
  edge_array(int) cost2(G);
  forall_edges(e, G) cost2[e] = dist1[G.source(e)] + cost[e] - dist1[G.target(e)];
  forall_nodes(v, G) DIJKSTRA(G, v, cost2, DIST[v], pred);
  forall_nodes(v, G)
    forall_nodes(w, G) DIST[v][w] = DIST[v][w] - dist1[v] + dist1[w];
}
```

Program 4: All Pairs Shortest Path

Program 4

The running time of this algorithm for a graph G with n nodes and m edges is $O(n + m + T_{declare} + n(T_{insert} + T_{Deletemin} + T_{get\_inf}) + m \cdot T_{Decrease\_key})$ where $T_{declare}$ is the cost of declaring a priority queue and T_{XYZ} is the cost of operation XYZ. With the time bounds stated in section II we obtain an $O(m + n \log n)$ algorithm.

Program 3 is very similar to the way Dijkstra's algorithm is presented in textbooks ([AHU83], [M84], [T83]). The main difference is that **program 3 is executable code** whilst the textbooks still require the reader to fill in (non-trivial) details.

Dijkstra's algorithm is a useful subroutine for the solution of the all-pairs shortest path problem in graphs with arbitrary edge costs, cf. [M84, section IV.7.4]. One uses the algorithm of Bellman-Ford to solve the single-source shortest path for some source s, then uses the solution of this computation to make all edge costs non-negative and then uses Dijkstra's algorithm to solve $n - 1$ single-source problems with non-negative edge costs. In order for this approach to work it is important that all nodes of the graph are reachable from s. The easiest way to achieve this is to add a new node s and to add edges of high cost from s to all other nodes. The details are given in program 4.

```
#include <LEDA/graph.h>
#include <LEDA/partition.h>
declare(node_array,partition_item);
int cmp(edge e1, edge e2, edge_array(int)& C) { return (C[e1] - (C[e2]); }
void MST(graph& G, edge_array(int)& cost, edgelist& EL)
// the input is an undirected graph G together with a cost function
// cost on the edges; output: list of edges EL of a minimum spanning tree
{ node v, w;
  edge e;
  partition P;
  node_array(partition_item) I(G);
  forall_nodes(v, G) I[v] = P.make_block();
  edgelist OEL = G.all_edges();
  OEL.sort(cmp, cost);
  // OEL is now the list of edges of G ordered by increasing cost
  EL.clear();
  forall(e, OEL)
  { v = G.source(e);
    w = G.target(e);
    if (!(P.same_block(I[v], I[w])))
    { P.union_blocks(I[v], I[w]);
      EL.append(e);
    }
  }
}
```

Program 5: Minimum Spanning Tree

Program 5

IV. Graphs and Data Types

We use the minimum spanning tree problem to further discuss the interaction between graphs and data types. Program 5 shows a minimum spanning tree algorithm. We do not discuss the details of the algorithm, cf. [M84, section IV.8] for the proof of correctness, but concentrate instead on the similarities of programs 3 and 5. In both cases a node_array(item) is used and in both cases the program starts by creating one item for each node of the graph. Similar statements occur in many graph algorithms.

A user of LEDA may want to incorporate all these statements into the declaration of the partition or the priority queue. He can do so (in fact we have done it already) by deriving a data type node_partition from the data type partition and similarly for priority_queue (cf. section VI). A node_partition Q consists of a *node_array(partition_item)* I and a partition P. The declaration

node_partition $Q(G)$

will then execute lines (3), (10), (11), and (12). The operations on node_partitions are also easily derived, e.g., Q.*same_block*(v, w) just calls P.*same_block*$(I[v], I[w])$. Altogether, this yields the simplified program 6.

```
(1)   #include <LEDA/graph.h>
(2)   #include <LEDA/partition.h>
(3)   int cmp(edge e1, edge e2, edge_array(int)& C) { return (C[e1] - (C[e2]); }
(4)   void MST(graph& G, edge_array(int)& cost, edgelist& EL)
(5)   { node v, w;
(6)     edge e;
(7)     node_partition Q(G);
(8)     edgelist OEL = G.all_edges();
(9)     OEL.sort(cmp, cost);
(10)    EL.clear();
(11)    forall(e,OEL)
(12)    { v = G.source(e);
(13)      w = G.target(e);
(14)      if (!(Q.same_block(v, w))
(15)      { Q.union_blocks(v, w);
(16)        EL.append(e);
(17)      }
(18)    }
(19)  }
```
Program 6: Simplified MST Program

Program 6

The reader may ask at this point why we provide the elegant types node_partition and node_priority_queue in this roundabout way. Why do we first introduce items and then show how to hide them? The reason is that in the case of graphs the ground set of the partition or priority queue is static. In general, this is not the case

Consider, for example, the standard plane sweep algorithm (cf. [M84, section VII.4.1, section VII.4.1]) for computing line segment intersections. It uses two information structures, usually called the X- and Y-structure. The Y-structure is an ordered sequence of intersections of the sweep line with the line segments and the X-structure is a priority queue. The priority queue contains an event for each line segment l of the Y-structure which intersects the succeeding line segment *lsuc* in front of the sweep line. The event occurs when the sweep line passes the intersection.

```
x_sweep = x;
sort_seq_item sit = Y_structure.insert(l, nil);
sort_seq_item sitpred = Y_structure.predecessor(sit);
sort_seq_item sitsuc = Y_structure.successor(sit);
pq_item pqit;
if (sitpred != nil)
{ if ((pqit = Y_structure.info(sitpred)) != nil) X_structure.delete_item(pqit);
     // removes the event, if any, associated with sitpred from the event queue
  line_segment lpred = Y_structure.key(sitpred);
  condpair inter = intersection(lpred, l);
  if (inter.status && (inter.x > x_sweep))
     Y_structure.change_inf(sitpred, X_structure.insert(sitpred, inter.x))
  else
     Y_structure.change_inf(sitpred, nil)
}
if (sitsuc != nil)
{ line_segment lsuc = Y_structure.key(sitsuc);
  condpair inter = intersection(l, lsuc);
if (inter.status && (inter.x > x_sweep))
  Y_structure.change_inf(sit, X_structure.insert(sit, inter.x);
}
```

Program 7: This program fragment processes the left endpoint of a line segment in the plane sweep algorithm for line segment intersection.

Program 7

In the algorithm the sweep line is moved from left to right. The sweep line stops whenever it passes through a left or right endpoint of a line segment or through an intersection. In either case the X- and the Y-structure have to be updated appropriately. Consider for example the situation where a left endpoint of some line segment l is encountered at coordinate x. The following actions have to be taken: insert l into the Y-structure, say between *lpred* and *lsuc*, remove the event, if any, associated with *lpred* from the X-structure and add the events associated with *lpred* and l, if they exist.

The appropriate LEDA types are

```
sort_seq(line_segment,pq_item) Y_structure;
priority_queue(sortseq_item,int) X_structure;
```

The Y-structure is a sequence of sortseq_items. Each item contains a line segment as its key and a pq_item as its information. The ordering is induced by the intersection of the line segments with the sweep line. Similarly, the X-structure stores for each item of the Y-structure the x-coordinate of the corresponding event. Each item in the Y-structure has direct access (through pq_item) to the associated event and each event in the X-structure has direct access (through sortseq_item) to the affected position of the Y-structure. Program 7 shows the code which processes the left endpoint of a line segment l at x-coordinate x. It makes use of a function *intersection(line_segment lpred, l)*, which given two line segments returns a pair *(status, xcoord)* (type *condpair*), where *status* indicates whether the two segments intersect and, if so, *xcoord* is the x-coordinate of the intersection. The variable *x_sweep* denotes the current position of the sweep line.

V. Inside LEDA

This section gives some of the implementation details of LEDA. The reader should be familiar with the major features of C++

1. Implementation of abstract data types

As mentioned before each data type in LEDA is realized by one or more C++ classes. The operations and operators are member functions of the corresponding classes. In C++ a class definition consists of a *declaration part* and an *implementation part*. The declaration of a class describes the interfaces of its member functions (return and parameter types) and the private data of each instance of the class. The former part of a class declaration corresponds to the abstract specification of the data type. The implementation part of the class fills in the missing C++ code to realize the member functions.

We will now treat a realization of the data type *partition* in detail. The implementation is based on the well-known union-find data structure. A *partition* is a forest of partition nodes, *partition_items* are pointers to partition nodes. All nodes of a partition are linked in a singly linked list. This list is used (by the destructor) to free storage when the scope of the partition ends.

```
typedef partition_node* partition_item;

class partition {

// private:

partition_item used_items;     // first item on linked list of used items

public: // operations

partition_item make_block();
partition_item find_block(partition_item);
int            same_block(partition_item, partition_item);
void           union_blocks(partition_item, partition_item);
void            clear();
// constructor:
partition()     { used_items = nil; }
// destructor:
˜partition()    { clear(); }
};
```

Note that the constructor is called automatically to initialize an instance of the class when it is created by a variable declaration. The destructor is called automatically when the scope of the instance ends.

```
class partition_node {
friend class partition;  // members of class partition are allowed to access private data
// private:
partition_node* father;  // parent node in the forest
partition_node* next;    // next node in the linked list of used nodes
int size;                // size of the subtree rooted at this node
public:
// constructor:
partition_node(partition_node* n) { father=nil; size=1; next=n; }
}
```

The C++ code realizing the member functions of the class partition follows. We use the union-find algorithm with weighted union rule and path compression.

```
partition_item partition::make_block()
{ // create new item and insert it into list of used item
  used_items = new partition_node(used_items);
  return used_items;
}

partition_item partition::find_block(partition_item it)
{ // return the root of the tree that contains item it
  partition_item x,root = it;
  while (root→father) root = root→father;
  // path compression:
  while (it!=root)
  { x = it→father;
    it→father = root;
    it = x;
  }
  return root;
}
```

```
int partition::same_block(partition_item a, partition_item b)
{ return find_block(a)==find_block(b); }

void partition::union_blocks(partition_item a, partition_item b)
{ // weighted union
  a = find_block(a);
  b = find_block(b);
  if(a→size > b→size)
    { b→father = a;
      a→size += b→size; }
  else{ a→father = b;
        b→size += a→size; }
}

void partition::clear()
{ // delete all used items
  partition_item p = used_items;
  while (used_items)
  { p = used_items;
    used_items = used_items→next;
    delete p;
  }
}
```

Note that only member functions or member functions of friends are allowed to access the private data of a class. This guarantees that the user can manipulate objects of a class only by member functions, i.e. only by the operations defined in the specification of the data type. This data hiding feature of C++ supports complete separation of the specification and the implementation of data types.

For every data type XYZ there exists a so-called header file "XYZ.h" containing the declaration of class XYZ. Programs using XYZ have to include this file. For example, partitions can only be used after the line

```
#include <LEDA/partition.h>      (see program 6)
```

The implementation of all classes are precompiled and contained in a library which can be used by the linker.

2. Parameterized Types

Most of the data types in LEDA have type parameters. In section II we defined a dictionary to be a mapping from a key type K to an information type I, here K and I are formal type parameters. The LEDA statement

"$declare2(dictionary, t_1, t_2)$"

declares a dictionary type with name "$dictionary(t_1, t_2)$" and actual type parameters $K = t_1$ and $I = t_2$. How is this realized?

Note that the operations on a dictionary are independent of key type K and information type I. So it is possible to implement all dictionary operations (member functions) without knowing K and I. This is done by implementing a base class dictionary with $K = I = void*$. For example

```
class dictionary { // base class
// private data
⋮
public:
⋮
void insert(void* k, void* i);
void* access(void* k);
⋮
};
```

In C++ the type void* (pointer to void) is used for passing arguments to functions that are not allowed to make any assumptions about the type of their arguments and for returning untyped results from functions. To declare a concrete data type for given actual type parameters (e.g., dictionary(int,int)) a derived class of the corresponding base class (dictionary) has to be declared. This derived class inherits all operations and operators from its base class and performs all necessary type conversion:

```
class dictionary(int,int): public dictionary {
⋮
void insert(int k,int i) { dictionary::insert((void*)k, (void*)i); }
int access(int k) { return (int)dictionary::access((void*)k); }
⋮
};
```

C++ 's macro facility is used to fill in such declarations of derived classes. There are macros declare, declare2, ... to declare data types with one, two, ... type parameters. declare2(dictionary,int,int) for example just creates the above declaration of dictionary(int,int).

3. Iteration

LEDA provides various kinds of iteration statements. Examples are

for lists:

forall(x, L) { the elements of L are successively assigned to x}

for graphs:

forall_nodes(v, G) { the nodes of G are successively assigned to v}

forall_adj_nodes(w, v) { the neighbor nodes of v are successively assigned to w}

All these statements are macros that are expanded to more complicated for-statements. The list iteration macro forall is defined as follows

```
#define forall(x, L)  for(L.init_cursor(); x = L.current_element(); L.move_cursor(); )
```

Here init_cursor(), move_cursor() and current_element() are member functions of the class list that manipulate an internal cursor.

The other iteration statements are implemented similarly.

VI. Extendibility

The goal of the LEDA project is the design of a library of reusable data types and algorithms that can easily be included into user programs. Of course, such a library can never be complete, i.e., there will always be situations requiring data types not contained in the library. Therefore there should be a possibility for users to extend the library by adding new data types and algorithms

LEDA is extendible in two ways:

1. New data types can be added as described in section V (Inside LEDA). This kind of extending the library is suitable for users having a detailed knowledge of the data structures, they want to use, and having experience in C++ programming.
2. New data types can also be constructed by combining already existing LEDA data types. This method allows non-experts to build data types on a higher level of description (the level of most example programs in this paper). An example of such a combination of two LEDA data type is the data type *node_partition* (cf. section IV). It is implemented by combining partitions and node_arrays of partition_items as follows:

```
class node_partition {
node_array(partition_item) I;
partition P;
public:
void union_blocks(node v, node w) { P.union_blocks(I[v], I[w]); }
int same_block(node v, node w)     { return P.same_block(I[v], I[w]); }
// constructor
node_partition(graph& G);
};

node_partition::node_partition(graph& G)
{ // construct a node_partition for the nodes of graph G
   node v;
   I.init(G, nil);
   forall_nodes(v, G) I[v] = P.make_block();
 }
```

We give another example for combining two LEDA data types. Assume we want to construct a data type *fast_list*. An instance of the data type *fast_list* is a linear list with a fast by key access operation. Fast lists can be implemented by a linear list L of dictionary items and a dictionary D. The dictionary is used to associate with each key its position in the list. The operations of the data type *fast_list* perform the corresponding operations on the linear list L and operations on the dictionary D to store or lookup the positions of the elements in L. The details are as follows.

```
declare2(dictionary,K,list_item)
(K is the element type of fast_list)
declare(list,dic_item)
typedef dic_item fast_list_item;
class fast_list {
list(dic_item) L;
dictionary(K,list_item) D;
public:
fast_list_item insert(fast_list_item it, K k)
{ // insert key k after item it
      dic_item x = D.insert(k,nil);
      list_item p = L.insert(D.inf(it),x);
      D.change_inf(x,p);
}
void remove(fast_list_item it) { L.del(D.inf(it)); D.del(it); }
fast_list_item access(K k) { D.lookup(k); }
K entry(fast_list_item it) { return D.key(it) }
fast_list_item succ(fast_list_item it) { return L.entry(L.succ(D.inf(it))) }

:

};
```

VII. Experiences

We report on our experiences in designing, implementing and using LEDA.

We found the task of specifying a data type surprisingly difficult. The data types dictionary and priority queue were the first two examples which we tried. The dictionary was readily specified; we had, however, lengthy discussions whether a dictionary is a function from keys to variables of type I or to objects of type I. The former alternative allows array notation for dictionaries, e.g. line 8 in program 1 could be written $D[k]++$, but also allows the user to store pointers to variables in our modules. The latter alternative makes notation more cumbersome but seems to be safer. We did not resolve the conflict but now have both alternatives to gain further insight by experiments. The priority queue took us a long time. We wanted to support access by position and we wanted a complete separation of data type and data structure. We found neither the combinatorial algorithms nor the abstract data type literature very helpful. In the algorithms literature the position concept is usually only discussed in the context of concrete implementations and is then equated with an index in an array or a pointer to a node of a data structure. In this way, no abstract definition of the data type is given and the data structures are intimately tied with the applications; e.g. priority queues are tied to shortest path calculations and partitions are tied to graph algorithms. In the latter part of the literature the position concept is only discussed in simple examples, e.g. iterators in linear lists [TRE88].

We use items as an abstraction of positions. Items are similar to the atoms of SETL. We found the item approach very flexible and, once we used it for priority queues, the specification of data types like sequences, partitions and lists became easy.

The implementation of LEDA was done by the second author, in particular, lists, graphs, and sorted sequences were implemented by him. Once the standards were set, we asked students to join in and to either realize additional data types or to give alternative realizations. Implementations of various kinds of dictionaries (BB[α]-trees, red-black-trees, (a,b)-trees, dynamic perfect hashing) and priority queues (Fibonacci-heaps, C-heaps) were provided by Dirk Basenach, Jürgen Dedorath, Evelyn Haak, Michael Muth, Michael Wenzel and Walter Zimmer.

LEDA was used to write graph and geometry algorithms. Some examples are shortest paths, components of various kinds, unweighted and weighted matchings, network flows, embeddings of planar graphs, visibility graphs of line segments, Voronoi diagrams and intersection of half spaces. The graph users liked LEDA because all the required data types such as graphs, node- and edge-arrays, lists, dictionaries, ... were available and hence LEDA increased their productivity enormously. This has led to more experimental work, one of the goals of the project. The first geometry users of LEDA were much less enthusiastic because almost none of the required types such as points, lines, ... were available. Stefan Meiser implemented some of them and we are now hearing the first positive reactions from the geometry users.

VIII. Conclusions

LEDA is a library of efficient data types and algorithms. At present, its strength is graph algorithms and the data structures related to them. The computational geometry part is evolving.

There are several other projects which aim for similar goals as LEDA, e.g. [B88, So89, L89]. We believe, that LEDA compares well with these systems because of

- the clear separations between specification and implementation,
- the natural syntax, and
- the inclusion of many of the most recent and most efficient data structures and algorithms.

We close this section with a list of algorithms that we implemented using LEDA data types.

1. Graph Algorithms

All graph algorithms are part of the library. They accept instances of any user-defined graph type *GRAPH(vtype, etype)* as argument. *1.1. Basic Graph Algorithms*

- depth first search
- breadth first search
- connected components
- transitive closure

1.2. Shortest Path Algorithms

- Dijkstra's algorithm
- Bellman/Ford algorithm
- all pairs shortest paths

1.3. Matchings

- maximum cardinality bipartite matching
- maximum weight bipartite matching

1.4. Network Flow

- maximum flow algorithm of Galil/Namaad
- maximum flow algorithm of Goldberg/Tarjan

1.5. Planar Graphs

- planarity test
- triangulation
- straight line embedding

2. Computational Geometry

- intersection of half spaces
- convex hull of point sets
- construction of Voronoi diagrams
- construction of visibility graphs

Acknowledgement: We want to thank our colleagues G. Hotz, J. Loeckx, W. Rülling, K. Sieber and R. Wilhelm for many helpful discussions.

IX. References

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

[AMOT88] R.K. Ahuja, K. Mehlhorn, J.B. Orlin, R.E. Tarjan: "Faster Algorithms for the Shortest Path Problem", Technical Report No. 193, MIT, Cambridge, 1988

[B88] A. Bachem: Personal Communication, 1988

[BKMRS84] A. Müller-von Brochowski, T. Kretschmer, J. Messerschmitt, M. Ries, J. Schütz : "The Programming Language Comskee", Linguistische Arbeiten, Heft 10, SFB 100, Univ. des Saarlandes, Saarbrücken, 1984

[FT84] M.L. Fredman, and R.E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms", 25th Annual IEEE Symp. on Found. of Comp. Sci., 338-346, 1984

[L89] C. Lins: "The Modula-2 Software Component Library", Springer Publishing Company, 1989

[M84] K. Mehlhorn: "Data Structures and Algorithms", Vol. 1–3, Springer Publishing Company, 1984

[So89] J. Soukup: "Organized C", Typescript, 1988

[St86] B. Stroustrup: " The C++ Programming Language", Addison-Wesley Publishing Company, 1986

[T83] R.E. Tarjan: "Data Structures and Network Algorithms", CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 44, 1983

[TRE88] P. Thomas, H. Robinson, J.Emms: "Abstract Data Types", Oxford Applied Mathematics and Computing Science Series, 1988