LEDA

A Library of Efficient Data Types and Algorithms [*]

Kurt Mehlhorn and Stefan Näher

Max-Planck-Institut für Informatik
D-6600 Saarbrücken, Federal Republic of Germany

Key words: abstract data type, reusable software, efficient algorithms, object oriented programming

Abstract

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

– a sizable collection of data types and algorithms,
– the precise and readable specification of these types,
– the inclusion of many of the most recent and efficient implementations,
– a comfortable data type graph,
– its extendibility, and
– its ease of use.

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

[*] This research was supported by the ESPRIT II Basic Research Action Program, ESPRIT P. 3075 – ALCOM. Most of this work was carried out while the authors were affiliated with Fachbereich Informatik, Universität des Saarlandes.

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, ...

In 1989, we started a project (called LEDA, for Library of Efficient Data types and Algorithms) to build a library of the data types and algorithms of combinatorial 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. In the first version of LEDA, which has been completed in the spring of 1990, this collection included most of the data types and algorithms described in the text books of the area(e.g. [AHU83],[M84],[T83],[CLR90]), i.e., stacks, queues, lists, sets, dictionaries, ordered sequences, partitions, priority queues, directed, undirected, and planar graphs, lines, points, planes and basic algorithms in graph and network theory and computational geometry. We refer the reader to the LEDA manual([N90]) for the list of data types and algorithms available as of Fall 1990.

- 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). For many efficient data structures access by position is important. In LEDA, we use an item concept to cast positions into an abstract form. We introduce this concept in section II and demonstrate its usefulness on several examples. We mention that most of the specifications given in the LEDA manual use this concept, i.e., the concept is adequate for the description of many data types.

- LEDA contains efficient implementations for each of the data types, e.g., Fibonacci heaps and redistributive heaps for priority queues, red-black trees and dynamic perfect hashing for dictionaries, ...

- 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 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 typical text book presentation. We emphasize that all examples given in this paper show executable code. A long term goal is the equation "Algorithm + LEDA = program".

In this paper we describe the status of the project. We hope that LEDA will narrow the gap between algorithms research, teaching, and implementation. Other projects with similar goals are described in ([B87],[Lin89],[So89]); however, all of these projects settle for a considerably smaller collection of data types and algorithms than LEDA. Our approach to reach the goals of LEDA is standard. Similarly to the other projects we employ

1. a strict separation between abstract data types and the concrete data structures used to implement them,

2. parameterized data types, and
3. object oriented programming.

The implementation language of LEDA is C++ . We have chosen C++ because of its flexibility and availability. All data types and algorithms are precompiled modules which can be linked with application programs. Polymorphic types are implemented as described in ([St86]).

This paper is organized as follows. In section II we discuss data types and data structures, and in section III the data type graph and the interaction of graphs and other data types. In section IV we briefly comment on the extendibility and in section V on the internal structure of LEDA. Section VI gives a short summary.

The design of LEDA is joint work by the two authors, the implementation was mostly done by the second author. The library can be used under UNIX with the C++ compilers AT&T cfront 2.0, cfront 2.1 and GNU g++ (version 1.37). It is available from the authors for a handling charge of DM 100.

II. Data Types and Items

In this section we discuss the specification of data types in LEDA on the basis of three examples: dictionaries, priority queues, and partitions.

Example 1: Dictionaries

A popular specification of dictionaries defines a dictionary as a partial function from some type K to some other type I, or alternatively, as a set of pairs from $K \times I$, i.e., as the graph of the function. In an implementation each pair (k,i) in the dictionary is stored in some location of the memory. Efficiency dictates that the pair (k,i) cannot only be accessed through the key k but also through the location where it is stored, e.g., we might want to lookup the information i associated with key k (this involves a search in the data structure), then compute with the value i a new value i', and finally associate the new value with k. This either involves another search in the data structure or, if the lookup returned the location where the pair (k,i) is stored, can be done by direct access. Of course, the second solution is more efficient and we therefore wanted to provide it in LEDA.

In LEDA, the abstraction of a position or location is called an item. An item is a container which can hold an object relevant for the data type. In the case of dictionaries a *dic_item* contains a pair consisting of a key and an information. An object of type *dictionary*(K,I), where K and I are types, is thus defined as a collection of items (of type *dic_item*) where each item contains a pair in $K \times I$. We use $< k,i >$ to denote an item with key k and information i and require that for each $k \in K$ there is at most one $i \in I$ such that $< k,i >$ is in the dictionary. In mathematical terms this definition may be rephrased as follows: A dictionary D is a partial function from the set *dic_item* to the set $K \times I$. Moreover, for each $k \in K$ there is at most one $i \in I$ such that the pair (k,i) is in the range of D.

The semantics of the operations

dic_item D.lookup(K k)
I D.inf(*dic_item* *it*)

void D.change_inf(*dic_item it*, $I\ i'$)

is now as follows: D.lookup(k) takes a key k and returns the item *it* with contents (k, i) (it returns *nil* if there is no such item), D.inf(*it*) extracts i from *it*, and a new value i' can be associated with k by D.change_inf(*it*, i').

Let us have a look at the insert operation for dictionaries next:
dic_item D.insert($K\ k$, $I\ i$)

There are two cases to consider. If D contains an item *it* with contents (k, i') then i' is replaced by i and *it* is returned. If D contains no such item, then a new item, i.e., an item which is not contained in any dictionary, is added to D, this item is made to contain (k, i) and is returned.

Remarks:

1. In LEDA specifications we use "T_0 XYZ.*op*($T_1\ t_1$, ..., $T_k\ t_k$)" to denote an operation *op* applied to an object XYZ with result type T_0 and arguments $t_1, \ldots,\ t_k$ of types $T_1, \ldots,\ T_k$ respectively.

2. Some readers may find it useful to interpret a *dic_item* as a pointer to a variable of type $K \times I$. The difference is that the assignment to the variable contained in a *dic_item* is restricted, e.g., the K-component cannot be changed, and that in return for this restriction the access to *dic_items* is more flexible than for ordinary variables, e.g., access through the value of the K-component is possible.

Dictionaries are implemented by leaf-oriented red-black trees. Operations insert and lookup take time $O(\log n)$, key, inf, and change_inf take time $O(1)$. Here n is the current size of the dictionary. The space requirement is $O(n)$.

We next give a simple application of the dictionary data type. Program 1 reads a sequence of strings terminated by "stop" and counts the number of occurrences of each string in the sequence. The number of occurrences is then listed for each string in the sequence.

```
 (1) declare2(dictionary, string, int)
 (2) main()
 (3) {
 (4)    dictionary(string, int) D;
 (5)    string s;
 (6)    dic_item it;
 (7)    while ( (cin >> s) && (s != "stop") )
 (8)          { it = D.lookup(s);
 (9)            if (it == nil) D.insert(s, 1);
(10)            else D.change_inf(it,D.info(it)+1);
(11)          }
(12)    forall_dic_items(it, D) cout << D.key(it) << " " << D.inf(it) << "\n";
(13) }
```

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

The details are as follows. In line (1) the dictionary type dictionary(string,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 (4) introduces D as the name of an object of type dictionary(string,int); D is initialized to the empty dictionary Lines (7) to (11) step through the input sequence. For each string s we look for an item with key s (line 8). If there is no such item in D the lookup operation returns nil and we insert the pair $(s, 1)$ (line 9). If there is such an item we increase its information (line 10). Finally, line (12) steps through all items in D and prints the corresponding key and information values.

The running time of program 1 on a sequence of n strings is $O(n \log n)$, since the program executes n lookup and at most n insert operations on D for a cost of $O(\log n)$ each and since the remaining $O(n)$ operations take constant time each. The use of items is not essential for the asymptotic running time of program 1. Even if a change_inf operation would have cost $O(\log n)$, the asymptotic running time would still be $O(n \log n)$, although with a larger constant factor. For the next example the use of items is essential.

Example 2: Priority Queues

Priority queues are a variation of dictionaries; they differ in three aspects from dictionaries: firstly, the set I of informations must be linearly ordered, secondly, there is an operation findmin() which returns an item with minimal information, and thirdly, there is a restricted change_inf operation decrease_inf(it, i) which decreases the information stored in item it to i. The operation raises an error if i is larger than the old information stored in item it. Figure 1 shows the specification of the data type priority queue as given in the LEDA manual ([N90]).

Priority queues are frequently used in network algorithms, e.g., shortest paths, minimum spanning tree, and maximum flow computations. In these applications the decrease_inf operation is typically executed more often than the other priority queue operations, e.g., in Dijkstra's algorithm for shortest paths (cf. program 3 in section III) decrease_inf is executed m times and the other operations are executed n times. Here m and n denote the number of edges and vertices of the network respectively. Note that m can be as large as n^2.

Several recent papers, e.g. [FT87] and [AMOT90], have shown that the decrease_inf operation can be realized in time $O(1)$ and all other operations in time $O(\log n)$. This led to a $O(m + n \log n)$ implementation of Dijkstra's algorithm. For the constant time implementation of decrease_inf it is absolutely crucial that the information to be changed in the data structure is accessed directly through its position in the data structure and not through the associated key. A lookup by key would necessarily have logarithmic cost. In LEDA access by position is achieved by the item concept in a natural way by simply defining priority queues as collections of pq_items and making the first argument of decrease_inf a pq_item.

Remark: The specification of priority queues given above differs from the one given in [FT87], [AMOT90] and [CLR90]. There, insert takes a new item with a predefined contents (k, i) as argument and inserts it into the queue, i.e., it is the application program's task to create new items. This either requires the application to know the internal structure of items, which violates the information hiding principle of abstract data types, or it requires an additional operation to create items, which is, however, not part of the specification given in these papers. In LEDA the queue itself creates the items. This point of view leads to an elegant specification which completely separates data type and application program.

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 to the empty queue.

3. Operations on a priority_queue 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 an 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 an 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.

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 the queue. The space requirement is $O(n)$.

Figure 1: The specification of priority queues.

Example 3: Partitions or Disjoint Sets

In this example we discuss partitions of finite sets, frequently called the union-find problem. An application of partitions is shown in program 4 of section III.

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, 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.

Remark: In the text books ([AHU83], [M84], [T83]) make_block takes an item as input and therefore the remark made in the priority queue example applies here also. We feel that the specification given above is more natural.

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 examples given above show that the item concept leads to natural specifications of data types. More examples can be found in [N90].

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.

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 as they are 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*. More precisely, a node v is selected and deleted from *zero*. Then all edges e starting in v are removed from G. If the removal of edge e decreases the indegree of its target node w to zero, then w is added to *zero*. Finally, G is acyclic if all edges are removed in the end.

```
(1) #include <LEDA/graph.h>

(2) bool ACYCLIC(graph G)
(3) { // Tests if G is acyclic by repeatedly deleting edges starting in nodes with indegree 0.

(4)   node_set zero; // Set of all nodes v with indeg(v) = 0
(5)   node v,w;

(6)   forall_nodes(v,G) if (G.indeg(v)==0) zero.insert(v);

(7)   while ( !zero.empty() )
(8)     { v = zero.choose();
(9)       zero.del(v);
(10)      forall_adj_edges(e,v)
(11)        { w = G.target(e);
(12)          G.del_edge(e);
(13)          if (G.indeg(w)==0) zero.insert(w);
(14)        }
(15)    }

(16) return G.number_of_edges() == 0;
(17)}
```

Program 2: Testing a directed graph for acyclicity.

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&*. 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 and *edge_array* are parameterized data types. An instance of the data type *node_array*(T) (*edge_array*(T)) is an array which is indexed by the nodes (edges) of some graph and whose entries are values of type T. Thus a node (edge) array is a mapping from the nodes (edges) of a graph into a set of variables of type T. 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.

```
#include <LEDA/graph.h>
#include <LEDA/prio.h>
declare2(priority_queue,node,int)
declare(node_array,pq_item)
void DIJKSTRA(graph& G, node s, edge_array(int)& cost,
                    node_array(int)& dist, node_array(edge)& pred )
{ priority_queue(node,int) PQ;
  node_array(pq_item) I(G,nil);
  int c;
  node u,v;
  edge e;
  forall_nodes(v,G)
  { pred[v] = nil;
    dist[v] = infinity;
    I[v] = PQ.insert(v,dist[v]);
  }
  dist[s] = 0;
  PQ.decrease_inf(I[s],0);
  while (!PQ.empty())
  { u = PQ.del_min()
    forall_adj_edges(e,u)
    { v = G.target(e);
      c = dist[u] + cost[e];
      if ( c < dist[v])
      { dist[v] = c;
        pred[v] = e;
        PQ.decrease_inf(I[v],c);
       }
     }
   } // while
}
```

Program 3: Dijkstra's shortest paths algorithm

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 (12)–(18). In LEDA the loop $forall\_nodes(v, G)\{\ldots\}$ can be used to iterate over all nodes v of a graph G.

Dijkstra's algorithm uses a priority queue PQ. The priority queue contains pairs $(v, dist[v])$ and hence has type *priority_queue*(*node*, *int*). The type *priority_queue*(*node*, *int*) is defined in line (3) and in line (7) the queue PQ is created. 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 (14). Then in each iteration we select and delete an item it with minimal inf from PQ and store its key in u, cf. line (20). We now iterate through all edges e starting in node u, cf. line (21). Let $e = (u, v)$ and let $c = dist[u] + cost[e]$ be the cost of reaching v through edge e, cf. lines (22) and (23). 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 (24) to (28).

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.

We use the minimum spanning tree problem to further discuss the interaction between graphs and data types. Program 4 shows the minimum spanning tree algorithm of Kruskal. It starts with a spanning forest of n isolated vertices (in line (12) a partition P is initialized with one block for each vertex of G) and then steps through the edges in order of increasing cost (line (13) constructs the list OEL of all edges of G, line (15) sorts this list and the forall statement in line (18) iterates over the edges in OEL). When an edge e is considered, it is checked whether the two endpoints v and w belong to the same tree of the spanning forest (line 21). If not, the two trees are united and the edge e is added to the spanning forest (lines 22 and 23). We refer the reader to [M84, section IV.8] for the proof of correctness.

```
(0)   #include <LEDA/graph.h>
(1)   #include <LEDA/partition.h>
(2)   declare(node_array,partition_item);
(3)   edge_array(int) *C;
(4)   int cmp(edge e1, edge e2) { return (*C[e1] - *C[e2]); }
```

```
(5)  void MST(graph& G, edge_array(int)& cost, edgelist& EL)
(6)  // the input is an undirected graph G together with a cost function
(7)  // cost on the edges; output: list of edges EL of a minimum spanning tree
(8)  { node v, w;
(9)    edge e;
(10)   partition P;
(11)   node_array(partition_item) I(G);
(12)   forall_nodes(v, G) I[v] = P.make_block();
(13)   edgelist OEL = G.all_edges();
(14)   C = &cost;
(15)   OEL.sort(cmp);
(16)   // OEL is now the list of edges of G ordered by increasing cost
(17)   EL.clear();
(18)   forall(e, OEL)
(19)   { v = G.source(e);
(20)     w = G.target(e);
(21)     if (!(P.same_block(I[v], I[w])))
(22)     { P.union_blocks(I[v], I[w]);
(23)       EL.append(e);
(24)     }
(25)   }
(26) }
```

Program 4: Kruskal's minimum spanning tree algorithm

Programs 3 and 4 show some similarities. 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 of 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. 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*$(I[v], I[w])$. For details see section IV.

IV. 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 by writing the appropriate C++ modules. This way of extending the library is suitable for users having a detailed knowledge of data structures, and with 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 III). It is implemented by combining partitions and node_arrays of partition_items as follows:

```
class node_partition {
// A node_partition is a partition of the nodes of some graph G.
// It consists of a node_array of partition items and a partition.
node_array(partition_item) I;
partition P;
public:
node_partition(graph& G)
{ // constructs a node_partition for the nodes of graph G
  node v;
  I.init(G, nil);
  forall_nodes(v, G) I[v] = P.make_block();
 }
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]); }
};
```

V. Implementation of LEDA

This section comments on the implementation of LEDA. 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 [St86] or [Lip89].

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 a class fills in the C++ code to realize the member functions.

C++ does not provide parameterized data types directly. However, [St86] discusses how the macro facilities of C++ can be used to simulate parameterized data types. The parameterized data types of LEDA are realized by Stroustrup's method.

LEDA provides various kinds of iteration statements, cf. programs 1 to 4. All iteration statements are implemented by macros which expand the iteration into more complicated for-statements.

LEDA can be used under UNIX with the C++ compilers AT&T cfront 2.0, cfront 2.1, and GNU g++ (version 1.37). All modules are precompiled and contained in 3 libraries: basic data types, graph data types and algorithms, and data types and algorithms for computational geometry. The total size of the libraries is about 2 Mbyte.

VI. Summary

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, Lin89]. We believe, that LEDA compares well with these systems because of

– its sizable collection of data types and algorithms
– the precise and readable specification of these types
– the item concept for positions and pointers
– the natural syntax, and
– the inclusion of many of the most recent and most efficient data structures and algorithms.

Acknowledgement: We want to thank our colleagues G. Hotz, J. Loeckx, W. Rülling, K. Sieber and R. Wilhelm for many helpful discussions and our students D. Basenach, J. Dedorath, E. Haak, M. Muth, M. Wenzel and W. Zimmer for implementation work.

VII. References

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

[**AMOT90**] R.K. Ahuja, K. Mehlhorn, J.B. Orlin, R.E. Tarjan: "Faster Algorithms for the Shortest Path Problem", JACM, Vol. 37, 213-233, 1990

[**B87**] G. Booch: "Software Components with Ada", Benjamin/Cummings Publ. Company, 1987

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

[**FT87**] M.L. Fredman, and R.E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms", JACM, Vol. 34, 596-615, 1987

[**Lin89**] C. Lins: "The Modula-2 Software Component Library", Springer Publishing Company, 1989

[**Lip89**] S.B. Lippman: "C++ Primer", Addsion-Wesley Publishing Company, 1989

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

[N90] S. Näher: "LEDA User Manual Version 2.0", TR A17/90, Univ. des Saarlandes, Saarbrücken, 1990

[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