LEDA

A Library of Efficient Data Types and Algorithms *

Stefan Näher and Kurt Mehlhorn

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 related data structures. The computational geometry part is evolving. The main features of the library are

- a clear separation of specification and implementation
- parameterized data types
- its extendibility
- its ease of use.

At present, the data types stack, queue, list, set, dictionary, ordered sequence, priority queue, directed and undirected graph and partition are available. Based on these data types a variety of network algorithms (shortest paths, matchings, network flow, planarity testing and embedding, ...) and geometric algorithms (plane sweep, Voronoi digrams, ...) are included.

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

* 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.

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 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 the fall of 1988, 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. The main features of the library are:

1) A clear separation between (abstract) data types and the data structures used to implement them. This distinction is frequently 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. In LEDA, specifications are given using standard mathematical terminology, e.g., a dictionary is defined as a function of finite support from some set K to some set I. We did not expect any difficulties, when we started to write LEDA specifications for dictionaries, priority queues, ... However, already priority queues turned out to be non-trivial. For the efficiency of several recent implementations of priority queues it is crucial, that operations take pointers into the data structure as arguments, a fact, which at first sight seems to exclude a specification independently of the implementation. To overcome this difficulty we introduced the abstract concept of a pointer, which we call item in LEDA. In the case of priority queues, we have pq_items. An insertion Q.insert(k, i) of a pair of key k and information i into a priority queue Q returns a pq_item it. The user of the queue can store this item and later use it to access the pair, e.g., in a decrease_inf operation: Q.decrease_inf(it, j) will reduce the information of the pair stored in item it to j. In this way we have access by position independently of the implementation.

2) Generic data types: Most of the data types in LEDA 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) LEDA is extendible: Users can include own data types either by implementing data structures from scratch in C++ or by combining already existing LEDA data types as described in [1].

4) Ease of use: All data types and algorithms are precompiled C++ modules which can be linked with application programs.

```
(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)}
```

Figure 1: Dijkstra's algorithm

Figure 1 shows an example (Dijkstra's algorithm for the single source shortest paths problem in digraphs with non-negative edge costs, cf. [AHU83], [M84, section IV.7.2], [T83]). The algorithm uses the data types graph and priority queue (lines (1) and (2)). In line (3), the parameterized data type priority queue is specialized to the type priority_queue(node,int), and in line (4), the parameterized data type node_array is specialized to node_array(pq_item); unfortunately C++ forces us to use different identifiers for the declare macro with different number of arguments.

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. 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).

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. Figure 1 is very similar to the way Dijkstra's algorithm is presented in textbooks ([AHU83], [M84], [T83]). The main difference is that Figure 1 shows executable code whilst the textbooks still require the reader to fill in (non-trivial) details.

LEDA offers the data types stack, queue, list, set, dictionary, ordered sequence, priority queue, partition, several graph types (undirected, directed. planar) and data types related to graphs. Also, a variety of graph and network algorithms, e.g. for connectivity, shortest paths, matchings, network flow, planarity testing and planar embedding, are included in the library.

The two authors started the LEDA project in the fall of 1988. Most of the implementation was done by the first author; he is sole project leader since the summer of 1989. More detailed informations about LEDA can be found in [1] or [3]. The user manual ([2]) lists the specifications of all data types currently contained in LEDA and gives many example programs. LEDA is available from the first author for a handling charge of DM 100.

References

[1] K. Mehlhorn and S. Näher: "LEDA, a Library of Efficient Data Types and Algorithms", MFCS 89, LNCS Vol. 379, 88 - 106, 1989

[2] S. Näher: "LEDA1.0 User Manual", Technical Report A 05/89, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1989

[3] S. Näher and K. Mehlhorn: "LEDA, a Library of Efficient Data Types and Algorithms", Technical Report A 04/89, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1989