
Introduction to LEDA

Martin Zachariasen
University of Copenhagen

Outline

- What is LEDA?
- History and Background
- Basic Concepts
- Parameterized Data Types and Algorithms
- Items
- Iteration
- Using LEDA at DIKU

What is LEDA?

LEDA (Library of Efficient Data Types and Algorithms) is a **C++ library** of combinatorial and geometric data types and algorithms.

Data Types: random sources, stacks, queues, maps, lists, partitions, dictionaries, sorted sequences, priority queues, . . .

Number Types: integers, rationals, bigfloats, algebraic numbers, linear algebra, . . .

Graphs: powerful representation of graphs, graph iterators, node- and edge-arrays, node priority queues and partitions, . . .

Graph Algorithms: spanning trees, shortest paths, flows, matchings, components, planarity, planar embeddings, . . .

Geometric Objects: points, lines, rays, circles, polygons, . . .

Geometric Algorithms: convex hulls, triangulations, Delaunay triangulations, Voronoi diagrams, segment intersection, . . .

Graphical Input and Output

History and Background

Authors: Kurt Mehlhorn and Stefan Näher

1988 LEDA project started

1990 Version 1.0

1995 Commercial version available

2000 Used at more than 1500 sites

2001 Version 4.3: Fully commercial product

Requires a license for academic and teaching purposes.

Can freely be used at DIKU (as explained later).
Installation at home under Windows or Linux requires
a student license (49 Euro).

Why was LEDA created?

- Algorithmic research often practical, but seldomly used.
- Algorithmic research requires **implementation of algorithms** in order to have maximum impact.
- The same data structure was implemented repeatedly, i.e., constant reinvention of the wheel.
- No depository of data types and algorithms available.
- LEDA: **platform for combinatorial and geometric computing**.

Example: Counting Strings

Problem: Read a sequence of strings from standard input (e.g., a file) and count the number of occurrences of each string.

```
#include <LEDA/d_array.h>
#include <LEDA/string.h>

int main()
{
    d_array<string,int> N(0);
    string s;

    while (cin >> s) N[s]++;

    forall_defined(s,N) cout << s << " " << N[s] << endl;
}
```

Example: Topological Sorting

Problem: Given an *acyclic* graph G , find an ordering of the nodes such that all edges run from smaller to higher numbered nodes.

```
bool TOPSORT(const graph& G, node_array<int>& ord)
{
    node_array<int> INDEG(G,0);
    node_list ZEROINDEG;

    int count=0;
    node v,w;

    forall_nodes(v,G) if ((INDEG[v]=G.indeg(v))==0) ZEROINDEG.append(v);

    while (!ZEROINDEG.empty())
    { v = ZEROINDEG.pop();
      ord[v] = ++count;
      forall_adj_nodes(w,v)
          if (--INDEG[w]==0) ZEROINDEG.append(w);
    }

    return count == G.number_of_nodes();
}
```

Example: Topological Sorting (cont.)

```
int main()
{
    GRAPH<string,int> G;

    node v1 = G.new_node("undershorts");
    node v2 = G.new_node("pants");
    node v3 = G.new_node("belt");
    node v4 = G.new_node("shirt");
    node v5 = G.new_node("tie");
    node v6 = G.new_node("jacket");
    node v7 = G.new_node("socks");
    node v8 = G.new_node("shoes");
    node v9 = G.new_node("watch");

    G.new_edge(v1, v2);  G.new_edge(v1, v8);
    G.new_edge(v2, v3);  G.new_edge(v2, v8);
    G.new_edge(v3, v6);
    G.new_edge(v4, v3);  G.new_edge(v4, v5);
    G.new_edge(v5, v6);
    G.new_edge(v7, v8);

    // Perform topological sort
    node_array<int> toporder(G);
    if (!TOPSORT(G, toporder)) cout << "Graph is not acyclic" << endl;

    // Print all nodes and their order
    node u;
    forall_nodes(u, G) cout << G[u] << " : " << toporder[u] << endl;
}
```

Number Types

Standard number types in **C++**:

short, int, long: Integers of limited size; fixed space is allocated for each type (typically 16, 32 and 64 bits).

float, double: Floating point numbers of limited size (typically 32 and 64 bits).

Improved number types in **LEDA**:

integer: Integers of arbitrary length.

rational, bigfloats: Rational and floating point numbers based on the **integer** type.

real: exact representation of algebraic numbers

Parameterized Data Types and Algorithms

Most data types in LEDA are defined using **templates**.

Makes it possible to specify, with a single code segment, an entire range of functions and classes.

Templates support **generic programming**:

- algorithms used for many different data types
- applications not dependent on detailed structure of data structure
(**container = templated data structure**)

No running time overhead: Compiler specializes code and performs usual type checking and code optimization.

Templates Functions and Classes: Examples

Template function:

```
template <class E>
void SWAP(E& a, E& b) { E tmp = a; a = b; b = tmp; }
```

Template class:

```
template <class E>
class stack {
public:
    stack() {...}
    void push(E x) {...}
    E pop() {...}
    int size() const {...}
    bool empty() const {...}
    void clear() {...}
}
```

Items

Item: Address of an element in a data structure or a stand-alone element.

Is **essentially** a pointer type; items are safer since operations on them are restricted.

Used for efficient **direct** reference to elements and **iteration**.

Two item types:

Dependent Items Refer to elements in a (complex) data structure, e.g., elements in lists or nodes in a graph.

Independent Items Refer to stand-alone elements, e.g., points or lines in the plane.

Similar to iterators in the Standard Template Library (STL), but conceptually different.

Dependent Item Types

Items that refer to elements that live as part of a collection.

The collection has some combinatorial structure:

- List
- Sorted sequence
- Priority queue
- Dictionary
- Partition
- Graph
- ...

Items and elements are accessed via member functions of the corresponding **collection**.

Independent Item Types

Items that refer to stand-alone elements:

- Integer
- Rational
- Point
- Segment
- Line
- ...

LEDA design: A point is an item and this item has two attributes (point coordinates) = logically a pointer to a container that contains two doubles.

Advantages: Faster assignments, faster identity tests, transparent storage management.

Disadvantages: The attributes of an independent item are immutable.

Iteration: Collections

Iteration macros for iterating over the elements in a collection:

```
list<point> L;
...
list_item it;
forall_items(it, L)
  { /* the items in L are successively assigned to it */ }

point p;
forall(p, L)
  { /* the elements of L are successively assigned to p */ }
```

Iteration: Graphs

Iteration macros for iterating over the nodes and edges in a graph:

```
graph G;
...
node u, v;
forall_nodes(u, G)
  { /* the nodes of G are successively assigned to u */ }

forall_adj_nodes(u, v)
  { /* the nodes adjacent to v are successively assigned to u */ }

edge e;
forall_edges(e, G)
  { /* the edges of G are successively assigned to e */ }
```

STL Style Iterators

Some data types also provide an STL compatible iteration scheme. The following example shows STL iteration on lists.

```
#include <LEDA/list.h>

int main ()
{
    list<int> l1;
    list<int> l2;

    l1.append(1); l1.append(3); l1.append(6); l1.append(7);
    l2.append(2); l2.append(4);

    l1.merge(l2);
    for (list<int>::iterator i = l1.begin (); i != l1.end (); i++)
        cout << *i << " ";
    cout << endl;
}
```

Comparing Objects

Users can define linearly ordered types that can be used as keys in, e.g., dictionaries and sorted sequences.

```
class pair {
    double x;
    double y;

public:
    pair(double xi = 0, double yi = 0) : x(xi), y(yi) {}
    pair(const pair& p) { x = p.x; y = p.y; }

    friend ostream& operator>> (ostream& is, pair& p)
        { is >> p.x >> p.y; return is; }
    friend ostream& operator<< (ostream& os, const pair& p)
        { os << p.x << " " << p.y; return os; }

    friend int compare(const pair&, const pair&);
};

int compare(const pair& p, const pair& q)
{
    if (p.x < q.x) return -1;
    if (p.x > q.x) return 1;
    if (p.y < q.y) return -1;
    if (p.y > q.y) return 1;
    return 0;
}
```

Comparing Objects (cont.)

Example of using pair class as key in a sorted sequence.

```
#include <LEDA/sortseq.h>
#include <LEDA/string.h>

int main()
{
    sortseq<pair,string> P;
    P.insert(pair(3.0, 4.0), "A");
    P.insert(pair(2.0, 2.0), "B");
    P.insert(pair(3.0, 3.0), "C");

    seq_item it;
    for (it = P.min_item(); it; it = P.succ(it))
        cout << P.key(it) << " " << P[it] << endl;
}
```

Graphics using LEDA

Simple and powerful `window` class for graphical input and output.

```
#include <LEDA/window.h>

int main()
{
    window W(400,400);
    W.display();

    point p;
    while (W >> p) W << p;

    W.screenshot("draw_points");
}
```

Compiling LEDA Programs at DIKU

LEDA can be used on all Linux machines at DIKU.

1. Update your `.tcshrc` file. Environment variable `LEDAROOT` must point to the directory where LEDA is installed. Also, add LEDA directory to the shared library path.

```
setenv LEDAROOT /usr/local/projects/disk02/algorithm/LEDA-4.3
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${LEDAROOT}
```

2. Copy Makefile from LEDA installation guide to the directory where your source files are located. In order to compile and link a single source file, say `myfile.c`, type

```
make myfile
```

If you need to link several files into one executable, please follow the instructions given in the Makefile.