The LEDA Platform for Combinatorial and Geometric Computing

Kurt Mehlhorn
MPI Informatik, Saarbrücken

Stefan Näher
Universität Trier

Christian Uhrig
Algorithmic Solutions GmbH, Saarbrücken

What is LEDA?

- A library of combinatorial and geometric data types and algorithms
  - covers a large part of combinatorial and geometric computing (AHU, CLR, Mehlh, PS, Ko, Ki)
  - easy to use
  - extendible
  - correct
  - efficient
- provides algorithmic intelligence for applications in GIS, VLSI-design, scheduling, traffic planning, graphics, facility planning, computational biology, . . .
- a platform on which to build applications
- a tool for teaching algorithms and algorithm engineering
- extended by AGD, LEDA-SM, CGAL

Modules in LEDA

- Basic Data Types: random source, stack, queue, map, list, set, dictionary, priority queue, ...
- Advanced Data Types: partition, sorted sequence, pq-trees, dynamic trees, range trees, interval trees, segment trees, ...
- Numbers: integer, bigfloat, real, linear algebra
- Graphs and Graph Algorithms: graphs, node and edge arrays, iterators, shortest paths, maximum flow, min cost flow, matching (weighted, unweighted, bipartite, general), assignment, components and connectivity, planarity, layout, ...
- Geometry: points, lines, circles, polygons, convex hulls, Delaunay diagrams, Voronoi diagrams, sweep-line method, ...
- Visualization and I/O: graphic windows, graph editor, persistence

other C++-libraries, e.g., STL, offer a subset of the basic data types.

Sample Applications

Application	Algorithmic Intelligence
traffic planning (Daimler-Chrysler)	graph algorithms, max flow, shortest path
geographic information systems, overlay of map (MUS)	intersection of polygons, point location
data mining (Silicon Graphics)	data structures, graph algorithms
VLSI-design (Ford)	graph algorithms, Steiner trees, scan-line algorithms

Why did we start the project?

- impact of algorithms research
- little reuse
- depository
- modularity and inadequate specifications

History

- conceived idea in fall of '88
- first release in '90, data structures and graphs
- geometry and graphs in '92
- first commercial use in '94
- LEDA Software GmbH in '95, renamed to Algorithmic Solutions GmbH in '98
- LEDA book in '99

LEDA users

- academic users (= users which do not pay)
  - 1500 installations in more than 50 countries
  - $\leq$ 50% of our users are in CS
  - $\leq$ 20% of our users are in algorithms
- commercial users

  MCI, USA, Comptel, Finnland, France Telecom, Frankreich, E-Plus Mobilfunk, Düsseldorf, ATR Telecommunications, Japan, Sun Microsystems, USA, Siemens AG, Lufthansa Systems, Kelsterbach, Daimler Benz, Berlin, Ford, USA, Silicon Graphics, USA, Digital Equipment, USA, Sony Corporation, Japan, Minolta, Japan, Shinko, Japan, Chevron Petroleum, USA, Prediction Company, USA, Commerz Financial Products, Frankfurt, IBM, Japan, and many others

Correctness

Correctness is of crucial importance
for the user of a library.

The programs in LEDA are correct because

- **they are built on a sound basis**, i.e..,
  - exact geometry kernels
  - exact number types
  - analysis of arithmetic demand
- we start from correct algorithms
- programs are well documented
- programs are intensively tested
- **program result checking is used to verify results**

The Exact Geometry Kernel

The intersection point of two lines lies on both lines.

```
count = 0; max_coord = 10000;
for (i = 0; i < 100000; i++)
{ point a, b, c, d, p;
  random_point(a,max_coord); // same for b, c, and d
  line l1(a,b), l2(c,d);
  if ( !l1.intersection(l2,p) || l1.contains(p) ) count++;
}
cout << count;              // outputs 33178 in 2.68 sec
// and with RATIONAL COORDINATES
count = 0;  max_coord = 10000;
for (i = 0; i < 100000; i++)
{ rat_point a, b, c, d, p;              // !!!!
  random_point(a,max_coord);
  rat_line l1(a,b), l2(c,d);            // !!!!
  if ( !l1.intersection(l2,p) || l1.contains(p) ) count++;
}
cout << count;               // outputs 100000 in 13.18 sec
```

Number Types

The LEDA number types give you exact and efficient arithmetic.

- *integer*, arbitrary precision integers
- *bigfloat*, arbitrary precision floating point numbers
- *rational*, rational numbers
- *real*, $+$, $-$, $*$, $/$, arbitrary roots and comparisons

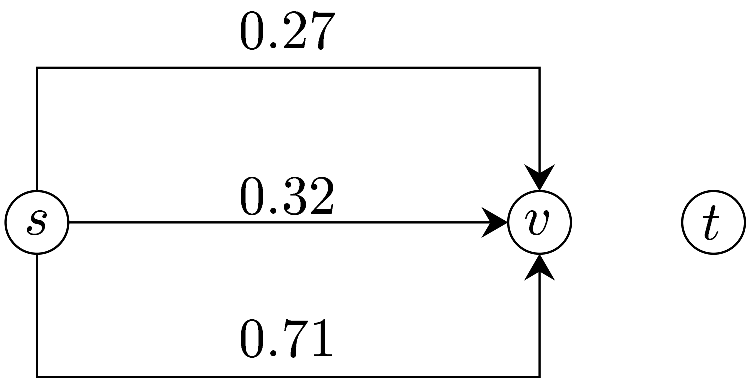
The Number Type REAL

checking the identity $(a + b) \cdot (a - b) = a^2 - b^2$.

```
int count = 0;

for (int i = 1; i < 10000; i++)
{ double x = (double) rand_int();

  double a = sqrt(x + 5), b = sqrt(x);

  if ( (a + b) * (a - b) == 5 ) count++;
}
cout << count;   // outputs 0 after 0.02 sec

// and with LEDA NUMBER TYPE REAL

count = 0;

for (int i = 1; i < 10000; i++)
{ real x = (real) rand_int();    // !!!!

  real a = sqrt(x + 5), b = sqrt(x);

  if ( (a + b) * (a - b) == 5 ) count++;
}
cout << count;   // outputs 10000 after 37.42 sec
```

Arithmetic Demand I

- *ints* may overflow, *doubles* incur round-off error.



- **Example: Preflow-Push Maxflow**
- assume floating point arithmetic with a mantissa of two decimal places and
- rounding towards infinity
- pushing out of s puts $0.27 \oplus 0.32 \oplus 0.71 = 1.3$ into v. (no cancellation).
- however, $1.3 \ominus 0.27 = 1.1$, $1.1 \ominus 0.32 = 0.78$ and $0.78 \ominus 0.71 = 0.07$. (cancellation)
- v ends up with an excess of 0.07, but no each to push across
- this has put the algorithm into an infinite loop.

Arithmetic Demand II

We need theorems concerning the arithmetic demand of network algorithms.

Thm (weighted bipartite matching alg):
If all edge weights are integers $\leq C$,
all numbers handled by the algorithm are $\leq 4 \cdot n \cdot C$.

Applications:

- *ints*: warn user if $4 \cdot n \cdot C > 2^{32}$.
- *doubles*:
  - scale edge weights into the form $0.00\ldots0xxxxxx \cdot 2^e$
  - with same exponent e for all edges and
  - sufficiently many leading zeros (namely $\log 4nC$)

Remark: All network algorithms in LEDA come with a theorem concerning their arithmetic demand and use it as outlined in applications.

Planarity Test

Input:	a graph G
Output:	yes, if G is planar
	no, if G is non-planar

Story

Planarity Test

Input:	a graph G
Output:	yes, if G is planar
	no, if G is non-planar

Story

YES: the implementor made a mistake

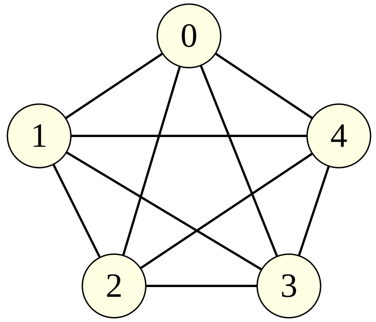
BUT: the specification is to blame

The output of a program should be "convincing".

A Convincing Planarity Test

Input: a graph G

Output:



- a planar embedding, if G is planar
- a Kuratowski subgraph K_5 or $K_{3,3}$, if G is non-planar.

- it is easy to check the witness
  - checking the embedding: checking Euler's relation: $n - m + f = 2$.
  - checking Kuratowski subgraph: check disjointness of paths
- most algorithms in LEDA give "proof" and come with a result checker
- we have not had a bug report for any of the major algs for more than 6 months

Ease of Use and Extendibility

LEDA is fun to use and easy to extend

- algorithm + LEDA = program
- platform versus library
- correct programs are easier to work with
- works well together with
  - CGAL, AGD, LEDA-SM
  - ABACUS, CPLEX, SOPLEX
  - numerous LEDA extension packages

Dijkstra's Shortest Path Algorithm

$$G = (V, E), \text{ directed graph}$$
$$s \in V, \text{ source}$$
$$cost : E \to N, \text{ edge costs}$$

set $dist(s) = 0$;
set $dist(v) = \infty$ for $v \neq s$;
declare all nodes unreached;
while there is an unreached node
do let u be an unreached node with minimal $dist$-value;
    declare u reached;
    forall edges $e = (u, v)$ out of u
    do set $dist(v) = \min(dist(v), dist(u) + cost(e))$ **od**
od

store the pairs $\{(v, dist(v)); v \text{ unreached}\}$ in a priority queue

The LEDA Program

```
void DIJKSTRA(const graph &G, node s, const edge_array<double>& cost,
              node_array<double>& dist)
{
  node_pq<double> PQ(G);
  node v; edge e;

  forall_nodes(v,G)
  { if (v == s) dist[v] = 0; else dist[v] = MAXDOUBLE;
    PQ.insert(v,dist[v]);
  }

  while ( !PQ.empty() )
  { node u = PQ.del_min();
    forall_out_edges(e,u)
      { v = G.target(e);
        double c = dist[u] + cost[e];
        if ( c < dist[v] ) { PQ.decrease_p(v,c);  dist[v] = c; }
      }
  }
}
```

Efficiency

LEDA is a decathlete
who can compete with many specialists

- start from good algorithms
- invent new one
- explore implementation choices
- compare

- there is no overhead in using a carefully designed library
- use of library opens up new opportunities

Maximum Flow, Implementation Choices

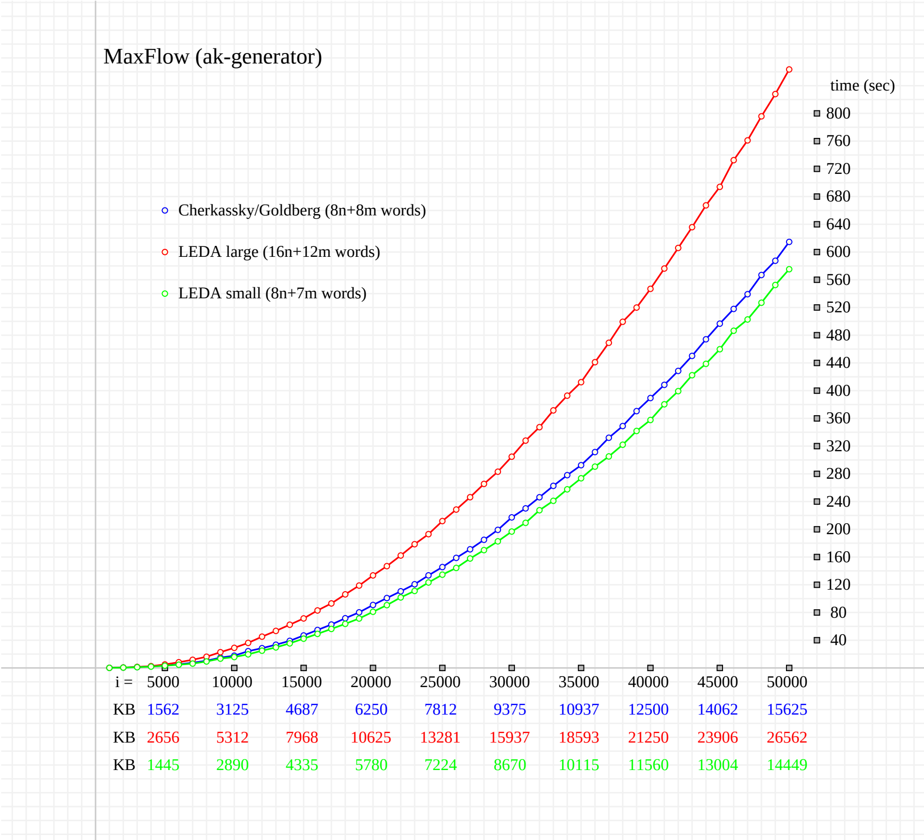
Gen	BASIC	HL	LRH	GRH	GAP	LEDA
CG1	192.8	191.5	193.7	4.87	5.34	3.28
CG2	173.9	167.9	120.5	0.3599	0.28	0.1802
AMO	62.15	65.3	6.99	0.26	0.26	0.1399

- rows correspond to different problem generators
- columns correspond to different implementation strategies

BASIC	=	text book version
HL	=	above + distinction between low and high levels
LRH	=	above + agressive local relabeling
GRH	=	above + global relabeling
GAP	=	above + gap heuristic
LEDA	=	above + improved storage layout

text books describe the implementation in the first column

Maximum Flow, Comparison



- it is a common belief that use of a library has an inherent overhead
- **this is not true**
- Goldberg's implementation is in C
- LEDA implementation uses C++ and all the nice features of LEDA

Weighted Matchings in General Graphs

Instance	Blossom IV	MS
Delaunay, $n = 40,000$	32	24
Random, $n = 40,000$, $m = 6n$	205	29
Diff, $n = 151,780$, $m = 881,317$	200,019	2,799

- Blossom IV (Applegate/Cook and Cook/Rohe) is based on the $O(n^3)$ version of Edmonds' algorithm due to Lawler and Gabow. It refines it through the use of clever heuristics.
- MS (Mehlhorn/Guido Schäfer) realizes the $O(nm \log n)$ algorithm of Ghalil, Micali, and Gabow. It uses advanced data structures (e.g., concatenable queues).
- we did not have to start from scratch.
- quantity becomes quality.

STL vs LEDA

Number of list entries: 100000

LIST<INT>	LEDA	STL
build list	0.020 sec	0.040 sec
pop and push	0.030 sec	0.030 sec
reversing	0.020 sec	0.030 sec
copy constr	0.050 sec	0.050 sec
assignment	0.020 sec	0.040 sec
clearing	0.000 sec	0.020 sec
sorting	0.130 sec	0.400 sec
sorting again	0.140 sec	0.330 sec
merging	0.030 sec	0.080 sec
unique	0.080 sec	0.080 sec
unique again	0.000 sec	0.010 sec
iteration	0.000 sec	0.000 sec
total	0.520 sec	1.110 sec