Reliable Algorithmic Software

Kurt Mehlhorn

MPI für Informatik
Saarbrücken
Germany
The Road Map

- Algorithms are at the heart of computer science; they make systems work.
The Road Map

- Algorithms are at the heart of computer science; they make systems work.
- The theory of algorithms, i.e., their design and their analysis, is a highly developed part of computer science.
The Road Map

- Algorithms are at the heart of computer science; they make systems work.
- the theory of algorithms, i.e., their design and their analysis, is a highly developed part of computer science.
- however, for many basic algorithmic tasks no reliable implementations are available
The Road Map

- Algorithms are at the heart of computer science; they make systems work.
- the theory of algorithms, i.e., their design and their analysis, is a highly developed part of computer science.
- however, for many basic algorithmic tasks no reliable implementations are available
- this is not just laziness on the side of implementers, is due to a lack of understanding (= theory)
The Road Map

- Algorithms are at the heart of computer science; they make systems work.

- The theory of algorithms, i.e., their design and their analysis, is a highly developed part of computer science.

- However, for many basic algorithmic tasks no reliable implementations are available.

- This is not just laziness on the side of implementers, is due to a lack of understanding (= theory).

- The challenge is to remedy this situation.
Algorithms are at the heart of computer science; they make systems work.

the theory of algorithms, i.e., their design and their analysis, is a highly developed part of computer science.

however, for many basic algorithmic tasks no reliable implementations are available

this is not just lazyness on the side of implementers, is due to a lack of understanding (= theory)

The challenge is to remedy this situation

- to work out the principles underlying reliable algorithmic software and
The Road Map

- Algorithms are at the heart of computer science; they make systems work.
- The theory of algorithms, i.e., their design and their analysis, is a highly developed part of computer science.
- However, for many basic algorithmic tasks no reliable implementations are available; this is not just laziness on the side of implementers, but due to a lack of understanding (= theory).
- **The challenge is to remedy this situation**
  - to work out the principles underlying reliable algorithmic software and
  - to create a comprehensive collection of reliable algorithmic software components.
State of the Art

- Popular algorithmic systems: Maple, Mathematica, STL, LEDA, CGAL, ACIS, LAPACK, MATLAB, CPLEX, Xpress, ILOG solver.

Can you trust any of them?

Most manuals evade the issue and avoid sentences which could be interpreted as guarantees.

- two basic algorithmic problems with no reliable implementation
  - Computer Aided Design (CAD), Boolean Operations on Solids
  - Linear Programming

- LEDA and CGAL are reliable: Belief or Fact?
  - LEDA = library of efficient algorithms and data types
  - CGAL = computational geometry algorithms library

- details on next slides
State of the Art: Boolean Operations on Solids

- The left-most picture shows a regular cylinder $P$, $n = 7$.
- The middle picture shows two copies of the cylinder: $Q$ was obtained by rotating $P$ by $\alpha$ degrees about its axis, $\alpha \approx 20^\circ$.
- The right-most picture shows the union of $P$ and $Q$ (= a cylinder whose base is a $4n$-gon).
existing CAD-systems are not reliable
construct a regular $n$-cylinder $P$,
obtain $Q$ by rotating $P$ by $\alpha$ degrees,
and compute the union of $P$ and $Q$.

<table>
<thead>
<tr>
<th>System</th>
<th>$n$</th>
<th>$\alpha$</th>
<th>time</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACIS</td>
<td>1000</td>
<td>1.0e-4</td>
<td>5 min</td>
<td>correct</td>
</tr>
<tr>
<td>ACIS</td>
<td>1000</td>
<td>1.0e-6</td>
<td>30 sec</td>
<td>incorrect answer</td>
</tr>
<tr>
<td>Rhino3D</td>
<td>200</td>
<td>1.0e-2</td>
<td>15 sec</td>
<td>correct</td>
</tr>
<tr>
<td>Rhino3D</td>
<td>400</td>
<td>1.0e-2</td>
<td>–</td>
<td>CRASH</td>
</tr>
</tbody>
</table>

the situation is even worse for objects with curved boundaries
Linear Programming

maximize \( c^T x \) subject to \( Ax \leq b \) \( x \geq 0 \)

- linear programming is a most powerful algorithmic paradigm
- There is no linear programming solver that is guaranteed to solve large-scale linear programs to optimality. Every existing solver may return suboptimal or infeasible solutions. There are solvers that solve small problems to optimality.

<table>
<thead>
<tr>
<th>Problem</th>
<th>C</th>
<th>R</th>
<th>NZ</th>
<th>T</th>
<th>V</th>
<th>Res</th>
<th>RelObjErr</th>
<th>Exact Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>degen3</td>
<td>1504</td>
<td>1818</td>
<td>26230</td>
<td>8.08</td>
<td>0</td>
<td>opt</td>
<td>6.91e-16</td>
<td></td>
</tr>
<tr>
<td>etamacro</td>
<td>401</td>
<td>688</td>
<td>2489</td>
<td>0.13</td>
<td>10</td>
<td>dfeas</td>
<td>1.50e-16</td>
<td>8.79</td>
</tr>
<tr>
<td>fffff800</td>
<td>525</td>
<td>854</td>
<td>6235</td>
<td>0.09</td>
<td>0</td>
<td>opt</td>
<td>0.00e+00</td>
<td>1.11</td>
</tr>
<tr>
<td>pilot.we</td>
<td>737</td>
<td>2789</td>
<td>9218</td>
<td>3.8</td>
<td>0</td>
<td>opt</td>
<td>2.93e-11</td>
<td>4.41</td>
</tr>
<tr>
<td>scsd6</td>
<td>148</td>
<td>1350</td>
<td>5666</td>
<td>0.1</td>
<td>13</td>
<td>dfeas</td>
<td>0.00e+00</td>
<td>0.52</td>
</tr>
<tr>
<td>scsd8</td>
<td>398</td>
<td>2750</td>
<td>11334</td>
<td>0.48</td>
<td>0</td>
<td>opt</td>
<td>7.54e-16</td>
<td>1.52</td>
</tr>
</tbody>
</table>

Dhifaoui/Funke/Kwappik/M/Seel/Schömer/Schulte/Weber: SODA 03

continue with exact LP-solver
indicated vertex may be returned

- indicated vertex is not primal feasible since it violates a constraint
- indicated vertex is not dual feasible since it is not optimal for a subset of the constraints.
Are LEDA and CGAL Reliable?

- I believe so:
  - the authors are trustworthy individuals at least most of the time
  - most programs are carefully documented but not all of them
  - extensively tested
  - underlying algorithms have been shown correct
  - number types give illusion of a Real RAM
  - geometry kernels are model of geometry
  - program result checking is used

- in court the above is called circumstantial evidence

- Am I willing to bet on correctness?
  - yes, in case of the sophisticated algorithms
  - definitely no, in case of support (graphics, windows, IO)

- there are no formal proofs of correctness
First Summary

• no reliable implementations exist for fundamental algorithmic problems such as Linear Programming or Boolean Operations on Solids

• we are lacking principles: CPLEX and ACIS are state of the art.

• CGAL and LEDA are a step forward, but by far not the end of the story

• abstract challenge:
  • work out the principles underlying reliable algorithmic software
  • create a comprehensive collection of reliable algorithmic software components.

• concrete challenges:
  • a correct and efficient CAD system
  • a correct and efficient LP solver
  • a certified LEDA
  • to meet either challenge will require new theory
Approaches

- program verification
- exact computation paradigm
- program result checking
- certifying algorithms
- verification of checkers
- cooperation of verification and checking
- a posteriori analysis
- test and repair
Program Verification

- formal program verification is the obvious approach.
- obstacles
  - the mathematics underlying the algorithms must be formalized
  - verification must be applicable to languages in which algorithmicists want to formulate their algorithms
- my opinion: the direct applicability of program verification is doubtful for some time to come
- but see below: verification of checkers
The Exact Computation Paradigm

Circle: \( x^2 + y^2 = 100 \)

Line: \( y = 1 - 0.10000000000000049 \cdot x \)

existing systems approximate the coordinates (usually, 16 digits)

\[ x_0 = 9,9999999999999999950000000000049 \ldots \]

and hence cannot distinguish

\[ \text{Abstand} < 10^{-16} \]

but geometric programs branch on this case distinction \( \longrightarrow \) disaster

• exact computation paradigm: implement an efficient Real RAM
A Separation Bound for Algebraic Expressions

Let $E$ be an expression with operators $+,-,\ast$ and $\sqrt{}$ and integer operands. Let

- $u(E) =$ value of $E$ after replacing $-$ by $+$.
- $k(E) =$ number of distinct square roots in $E$.

Then (BFMS, BFMSS)

$$E = 0 \quad \text{or} \quad |E| \geq \frac{1}{u(E)^{2k(E)} - 1}$$

Theorem allows us to determine signs of algebraic expressions by numerical computation with precision $(2^{k(E)} - 1) \log u(E)$.

in preceding example: compute the first 25 decimal digits of $x_0$ and you know how $x_0$ compares to 10.

related work: Mignotte, Canny, Dube/Yap, Li/Yap, Scheinermann

extensions: division, higher-order roots, roots of univariate polynomials
Discussion I

How small can $A - B\sqrt{C}$ be, if non-zero? $A, B, C \in \mathbb{N}$.

$$|A - B\sqrt{C}| = \left| \frac{(A - B\sqrt{C})(A + B\sqrt{C})}{A + B\sqrt{C}} \right| = \frac{|A^2 - B^2C|}{|A + B\sqrt{C}|} \geq \frac{1}{|A + B\sqrt{C}|} \geq \frac{1}{|A| + |B|\sqrt{C}}$$

This is a special case of the theorem

- $u(E) = |A| + |B|\sqrt{C}$
- $k = 1$
Recent Progress I

- efficient geometry kernels for linear objects in CGAL and LEDA

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\alpha$</th>
<th>time</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>6.175e-06</td>
<td>30 s</td>
<td>correct</td>
</tr>
<tr>
<td>20000</td>
<td>9.88e-07</td>
<td>141 s</td>
<td>correct</td>
</tr>
<tr>
<td>$\rightarrow \infty$</td>
<td>$\rightarrow 0$</td>
<td>$\rightarrow \infty$ sec</td>
<td>correct</td>
</tr>
</tbody>
</table>

- CORE and LEDA offer reasonably efficient computations with radicals
- ESOLID (Manocha): exact boundary evaluation of some curved solids.
- exact boolean operations on 2-dimensional curved objects of low degree

1 min for $n = 1000$

Berberich, Eigenwillig, Hemmer, Hert, M, Schömer: ESA 2002
Recent Progress II

• arrangements of ellipsoids

Geismman, Hemmer, Schömer: CompGeo 2001
Wolpert: PhD-thesis
Program Result Checking

- verification = program works for every input
  result checking = program worked for a specific input

- Blum and Kannan (89): programs that check their work

- a checker for a program computing a function $f$ takes
  - an instance $x$ and an output $y$, and
  - returns true if $y = f(x)$ and return false, otherwise

- hope: checking is simpler than computing

- example
  - multiplication problem: compute $y = x_1 \cdot x_2$, given $x_1$ and $x_2$.
  - a (probabilistic) checker gets $x_1$, $x_2$, and $y$, chooses a small random prime $p$ and
    - verifies that $(x_1 \mod p) \cdot (x_1 \mod p) \mod p = y \mod p$.

- program result checking is too restrictive to be practical
Certifying Algorithms

- certifying algorithms return additional output (= a witness) that simplifies checking.
- on input $x$, a certifying program for a function $f$ returns a value $y$ and additional information $I$ that makes it easy to check that $y = f(x)$.
- “easy to check” has twofold meaning:
  - there is a simple program $C$ that given $x$, $y$, and $I$ checks whether indeed $y = f(x)$.
  - if $y \neq f(x)$ then there should be no $I$ such that $(x, y, I)$ passes checking.
  - simple = correctness is “obvious”.
  - the running time of $C$ on inputs $x$, $y$, and $I$ should be no larger than the time required to compute $f(x)$ from scratch
  - preferably much much smaller
- observe that certifying program and checker are designed together
Example of a Certifying Algorithms

- planarity testing: given a graph $G$, decide whether it is planar
  - Tarjan (76): planarity can be tested in linear time
  - Chiba et al (85): planar embedding of a planar graph in linear time
  - a story
  - Hundack/M/Näher (97): Kuratowski subgraph of a non-planar graph in linear time

- many more examples are discussed in LEDA book
- in the LEDA system many programs are certifying.
Verification of Checkers

• the checker should be so simple that its correctness is “obvious”.
• we may hope to formally verify the correctness of the implementation of the checker

this is a much simpler task than verifying the solution algorithm
  • the mathematics required for the checker is usually much simpler that the one underlying the algorithm for finding solutions and witnesses
  • checkers are simple programs
  • algorithmicists may be willing to code the checkers in languages which ease verification

• **Remark:** for a correct program, verification of the checker is as good as verification of the program itself

• Harald Ganzinger and I are exploring the idea
Cooperation of Verification and Checking

• a sorting routine working on a set $S$
  (a) must not change $S$ and
  (b) must produce a sorted output.

• I learned the example from Gerhard Goos

• the first property is hard to check (provably as hard as sorting)
  but usually trivial to prove, e.g.,
  if the sorting algorithm uses a swap-subroutine to exchange items.

• the second property is easy to check by a linear scan over the output, but hard to prove (if the sorting algorithm is complex).

• give other examples where a combination of verification and checking does the job
A Posteriori Analysis

- there will always be inexact algorithms.
- a-posteriori analysis: analyze the quality of the solution
- example: roots of a univariate polynomial $f(x)$ of degree $n$
  
  - given approximate solutions $x_1, \ldots, x_n$, compute
  
  $$\sigma_i = \frac{f(x_i)}{\prod_{j \neq i} (x_i - x_j)} \text{ for } 1 \leq i \leq n.$$  

  - $\Gamma_i = \text{disk in the complex plane centered at } x_i \text{ with radius } n|\sigma_i|.$
  - the union of the disks contains all roots of $f$
  - a connected component consisting of $k$ disks contains exactly $k$ roots of $f$.
  - the $\sigma_i$ are easily computed with controlled error using multi-precision floating point arithmetic

- analogous examples in the combinatorial world, e.g., in approximation algs
Test and Repair

• use solution returned by an inexact algorithm as starting point for an exact algorithm

• example: linear programming

\[
\text{maximize } \quad c^T x \quad \text{subject to } \quad Ax = b, \quad x \geq 0
\]

\(A\) is an \(m \times n\) matrix with \(m < n\) and rank \(m\) (for simplicity)

• a basic solution \(x = (x_B, x_N)\) is defined by a \(m \times m\) non-singular sub-matrix \(B\) of \(A\)
  • \(x_B\) are the vars corresponding to cols in \(B\), \(x_N\) remaining vars
  • \(x_N = 0\) and \(x_B = B^{-1}b\)

• a basic solution is \textit{primal feasible} if \(x_B \geq 0\)

• a basic solution is \textit{dual feasible} if \(c_B^T x_B - c_N^T A_B A_N^{-1} A_N \leq 0\).

• it is optimal, if it is primal and dual feasible.

• for medium-size linear programs, we can check (exactly !!!!) for primal or dual feasibility in reasonable time (\textbf{details})
An Exact LP-Solver

- use an inexact LP solver to determine an “optimal” basis $B$
- check the basis for optimality. If optimal, stop.
- if not, use the basis as a starting basis for an exact simplex algorithm
- seems to work reasonably well
- turn this observation into a theorem
- extend to large scale linear programs

- general challenge for optimization problems
  - design (exact) algorithms that start from a given solution $x_0$ towards an optimal solution.
  - the running time should depend on some natural distance measure between the initial and the optimal solution.

- go back to road map slide
An Example of a Distance Measure

- LP is given as a set of inequalities in \( d \) variables, goal is to find the top-vertex.
- Difficulty of a vertex = number of facets whose top vertex is above the given vertex.
- Kalai (92):
  1. Given a vertex \( v \), consider the \( d \) facets incident to it.
  2. If \( v \) is the top vertex of all of them, stop.
  3. Among the facets incident to \( v \) whose top vertex is different from \( v \), choose one at random, say \( F \).
  4. Find the top vertex of \( F \) (by using the same algorithm recursively), call it \( v \), and go to step 1.
- \( T(d,n) \), running time for a problem in dimension \( d \) and starting with a vertex of difficulty \( n \). Then

\[
T(d,n) \leq \exp(O(\sqrt{n \log d}))
\]