The Reliable Algorithmic Software Challenge RASC

dedicated to Thomas Ottmann on the occasion of his 60th birthday

Kurt Mehlhorn*

September 8, 2002

1 Introduction

When I was asked to contribute to a volume dedicated to Thomas Ottmann's sixtieth birthday, I immediately agreed. I have known Thomas for more than 25 years, I like him, and I admire his work and his abilities as a cyclist. Of course, when it came to start writing, I started to have second thoughts. What should I write about? I could have taken one of my recent papers. But that seemed inappropriate; none of them is single authored. It had to be more personal.

A sixtieth birthday is an occasion to look back, but it is also an occasion to look forward, and this is what I plan to do. I describe what I consider a major challenge in algorithmics, and then outline some venues of attack.

2 The Challenge

Algorithms are 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 theoretical computer science [OW96].

In comparison, algorithmic software is in its infancy. For many fundamental algorithmic tasks no reliable implementations are available due to a lack of understanding of the principles underlying reliable algorithmic software, see Section 3 for some examples. *The challenge is*

- *to work out the principles underlying reliable algorithmic software and*
- *to create a comprehensive collection of reliable algorithmic software components.*

*Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany,www.mpi-sb.mpg.de/~mehlhorn

Problem				CPLEX solution		Exact Verification		
Name	C	R	NZ	RelObjErr	TC	V	Res	TV
degen3	1504	1818	26230	6.91e-16	8.08	0	opt	8.79
etamacro	401	688	2489	1.50e-16	0.13	10	feas	1.11
fffff800	525	854	6235	0.00e+00	0.09	0	opt	4.41
pilot.we	737	2789	9218	2.93e-11	3.8	0	opt	1654.64
scsd6	148	1350	5666	0.00e+00	0.1	13	feas	0.52
scsd8	398	2750	11334	7.54e-16	0.48	0	opt	1.52

Table 1: Behavior of CPLEX for problems in the Netlib library. The first four columns give the name of the instance, and the number of constraints, variables, and non-zeroes in the constraint matrix. The columns labeled RelObjErr, T, and Res give information about the solution computed by CPLEX: the column labeled TC shows the time (in seconds) for solving the LP, and the column labeled Res shows whether the basis (= a symbolic representation of a solution) computed by CPLEX is optimal or not. An entry "feas" indicates that the computed basis is feasible but *not* optimal. RelObjErr is the relative error in the objective value at the basis returned by CPLEX. For example, for problem pilot.we CPLEX found the optimal basis and returned an objective value with relative error 2.93e-11. The meaning of the remaining columns is explained later in the text.

3 State of the Art

I give examples of basic algorithmic problems for which no truly reliable software is available.

Linear Programming is arguably one of the most useful algorithmic paradigms. It allows one to formulate optimization problems over real-valued variables that have to obey a set of linear inequalities.

$$\text{maximize} \quad c^T x \quad \text{subject to} \quad Ax \leq b$$

where x is a vector of n real variables, A is an $m \times n$ matrix with $m > n$, b is an m-vector, and c is an n-vector. A large number of problems can be formulated as linear programs, e.g., shortest paths, network flow, matchings, convex hull, ..., and hence a linear programming solver is an extremely useful algorithmic tool. *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.*[1] For example, the current version of CPLEX does not find the optimal solution for problems etamacro and scsd6 in the Netlib library, a popular collection of benchmark problems; see Table 1.

Computer aided design systems manipulate 3-dimensional solids under boolean operations (and other operations); see Figure 1 for an example. *No existing system is guaranteed to compute the correct result, not even for solids bounded by plane faces*; see Table 2.

There are systems that solve large-scale problems on graphs and networks. These systems are trustworthy because their authors are. *They come with no formal guarantee.*

[1] There are solvers that solve small problems to optimality.

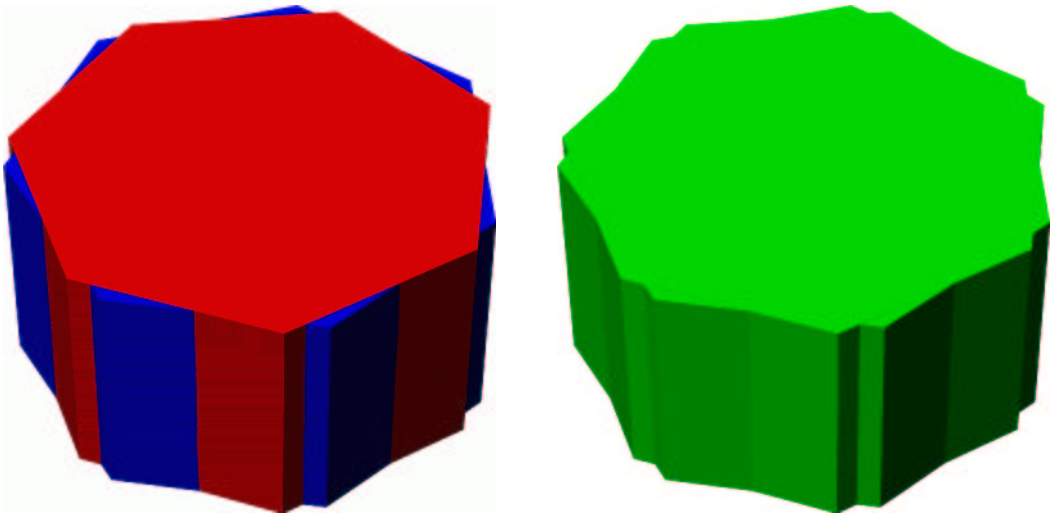

Figure 1: The figure on the left shows a red and a blue solid and the figure on the right shows their union.

System	n	α	time	output
ACIS	1000	1.0e-4	5 min	correct
ACIS	1000	1.0e-6	30 sec	incorrect answer
Rhino3D	200	1.0e-2	15sec	correct
Rhino3D	400	1.0e-2	–	CRASH

Table 2: Runs for the example of Figure 1. We used cylinders whose basis is a regular n-gon and that were rotated by α degrees relative each other. ACIS and Rhino3D are popular commercial CAD-engines.

The situation is even worse for parallel and distributed algorithms.

Many algorithmic libraries exist, e.g., Maple and Mathematica for symbolic computation, STL for data structures, LEDA for data structures, graph and network algorithms, and computational geometry, CGAL for computational geometry, ACIS for computer aided design, LAPACK for linear algebra problems, MATLAB for numerical computation and visualization, CPLEX and Xpress for optimization, and ILOG solvers for constraint solving. None of the implementations in any of these systems comes with a formal guarantee of correctness. Moreover, for problems such as linear programming and boolean operations on solids the existing implementations are known to be incorrect.

Given that algorithms (and frequently in the form of implementations from these libraries) form the core of any software system, we are facing a major challenge: Develop a theory for reliable algorithmic software and construct reliable algorithmic components based on it.

4 Approaches

I discuss some approaches for addressing the challenge. I hope to demonstrate that viable roads of attack exist. I do not claim and, in fact, do not believe that the approaches outlined are sufficient for solving the challenge.

4.1 Program Verification

Formal program verification is the obvious approach. However, there are several obstacles to applying it. I mention just two: (1) the non-trivial mathematics underlying the algorithms must be formalized, and (2) verification must be applicable to languages in which algorithmicists want to formulate their algorithms. In particular, we would need a formal semantics for languages like C++.In view of these obstacles, the direct applicability of program verification is doubtful, but see Section 4.5 below.

4.2 The Exact Computation Paradigm

Algorithms are frequently designed for ideal machines that are assumed to be able to calculate with real numbers in the sense of mathematics. However, real machines offer only crude approximations, namely fixed precision integers and floating point numbers. Arbitrary-precision integers and floating point systems go beyond, but are still approximate.

The *exact computation paradigm* goes a step further. It aims to exploit the fact that computations with algebraic numbers can be performed exactly. There are symbolic [Yap99, VG99] and numerical methods [BFM+01] to discover, for example, that

$$3\sqrt{\sqrt[3]{5}-\sqrt[3]{4}}-\sqrt[3]{2}-\sqrt[3]{20}+\sqrt[3]{25}=0$$

In principle, the computations required in computational geometry, computer aided design, and for a large class of optimization problems stay within the algebraic numbers and hence can be solved by the exact computation paradigm. The caveat is that the cost of exact algebraic computation is enormous, so enormous, in fact, that the approach was considered to be doomed to failure until recently.

Significant progress has been made in the last years. The geometry in LEDA [LED] and CGAL [CGA] follows the exact computation paradigm, but these systems deal only with linear objects. CORE [KLPY99] and LEDA [BFM+01] offer (reasonably) efficient computation with algebraic numbers that can be represented as expressions involving radicals, and ESOLID [KCF+02] takes a step towards exact boundary evaluation of curved solids. Exact boolean operations on 2-dimensional curved objects of low degree are now feasible; see Figure 2. However, the problem is much harder in 3D.

One of the earliest paper on the exact computation paradigm was co-authored by Thomas Ottmann; see [OTU87].

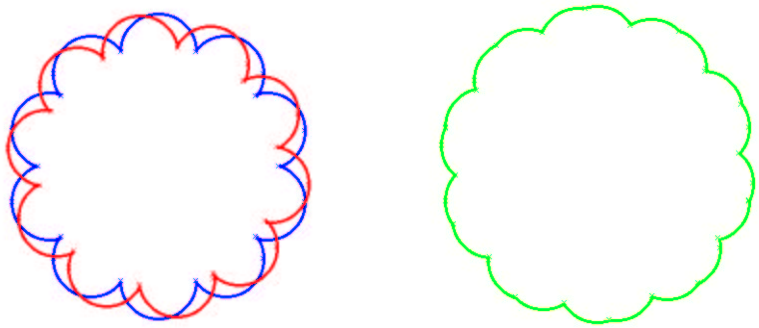

Figure 2: The figure on the left shows a red and a blue polygon with curved boundaries and the figure on the right shows their union. The computation takes about 1 minute for input polygons with 1000 vertices, see [BEH+02] for details.

The recent progress makes it plausible (this might be wishful thinking on my side) to build a CAD-system that is exact and efficient at the same time. *I consider the construction of such a system the litmus test for the exact computation paradigm.*

4.3 Program Result Checking

Instead of ensuring that a program computes the correct answer for *any* input (= verification), one may also try to verify that it computed the correct answer for a *given* input. The latter approach is called program result checking [BK89, WB97, BW96].

In its pure form, a checker for a program computing a function f takes an instance x and an output y, and returns true if $y = f(x)$. Of course, this is usually a simpler algorithmic task than computing $f(x)$.

We give an example: The multiplication problem is to compute the product $y = x_1 \cdot x_2$ of two given numbers x_1 and x_2. A (probabilistic) checker for the multiplication problem gets x_1, x_2, and y, chooses a random prime p and verifies that $(x_1 \bmod p) \cdot (x_1 \bmod p) \bmod p = y \bmod p$. If the equality holds, it returns true, otherwise it returns false. By repeating the test with independently chosen primes, the error probability can be made arbitrarily small. Some readers were taught this check (with $p = 9$) by their math teachers.

4.4 Certifying Algorithms

A looser version of program result checking is provided by certifying algorithms [SM90, MNS+96, KMMS02]. Such algorithms return additional output (frequently called a *witness*) that simplifies the verification of the result. More precisely, a certifying program for a function f returns on input x a value y, the alleged value $f(x)$, and additional information I that makes it easy to check that $y = f(x)$. By "easy to verify" we mean two things. Firstly, there must be a simple program

C (a checking program) that given x, y, and I checks[2] whether indeed $y = f(x)$. The program C should be so simple that its correctness is "obvious". Secondly, 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. This guarantees that the checking program C can be used without severe penalty in running time.

We give some examples.

Consider a program that takes an $m \times n$ matrix A and an m vector b and is supposed to determine whether the linear system $A \cdot x = b$ has a solution. As stated, the program is supposed to return a boolean value indicating whether the system is solvable or not. This program is not certifying. In order to make it certifying, we extend the interface. On input A and b the program returns either

- "the system is solvable" and a vector x such that $A \cdot x = b$ or
- "the system is unsolvable" and a vector c such that $c^T A = 0$ and $c^T \cdot b \neq 0$. Observe that such a vector c certifies that the system is unsolvable. Assume otherwise, say $Ax_0 = b$ for some x_0. Multiplying this equation with c from the left, gives $c^T Ax_0 = c^T b$ and hence $0 \neq 0$. Gaussian elimination is easily modified to compute the vector c in the case of an unsolvable system.

The certifying program is easy to check. If it answers "the system is solvable", we check that $A \cdot x = b$ and if it answers "the system is unsolvable" we check that $c^T A = 0$ and $c^T \cdot b \neq 0$. Thus the check amounts to a matrix-vector and a vector-vector product which are fast and also easy to program.

The second example id primal-dual algorithms for problems that can be formulated as linear programs. Consider the problem of finding a maximum cardinality matching M in a bipartite graph $G = (V, E)$. A matching M is a set of edges no two of which share an endpoint. A node cover C is a set of nodes such that every edge of G is incident to some node in C. Since edges in a matching do not share endpoints, $|M| \leq |C|$ for any matching M and any node cover C. It can be shown [CCPS98] that for every maximum-cardinality matching M there is a node cover C of the same cardinality. A certifying algorithm for the matching problem in bipartite graphs returns a matching M and a node cover C with $|M| = |C|$.

The third example is planarity testing. The task is to decide whether a graph G is planar. A witness of planarity is a planar embedding and a witness of non-planarity is a Kuratowski subgraph, see [MN99, Chapter 8] for details. It was known since the early 70s that planarity of a graph can be tested in linear time [LEC67, HT74]. Linear time algorithms to compute witnesses for planarity [CNAO85] and non-planarity [MN99, Section 8.7] were found much later.

For any algorithmic problem, we may ask the question whether a certifying algorithm exists for it. Short witnesses that can be checked in polynomial time can only exist for problems in NP $\cap$ co-NP. This holds for decision problems and optimization problems.

[2] Of course, if $y \neq f(x)$ then there should be no I such that the triple (x, y, I) passes checking.

4.5 Verification of Checkers

We postulated in the preceding section that the task of verifying a triple (x, y, I) should be so simple that the correctness of the program implementing the checker is "obvious". In fact, formal verification of checkers is probably a feasible task for program verification. Observe, that the verification problem is simplified in many ways: (1) the mathematics required for verifying the checker is usually much simpler than that underlying the algorithm for finding solutions and witnesses, (2) the checkers are simple programs, and (3) algorithmicists may be willing to code the checkers in languages that ease verification, e.g., in a functional language.

For a correct program, verification of the checker is as good as verification of the program itself.

4.6 Cooperation of Verification and Checking

Consider the following example[3]: a sorting routine working on a set S

(a) must not change S and

(b) must produce a sorted output.

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). A combination of verification and checking provides a simple solution for both parts.

4.7 A Posteriori Analysis

Despite the approaches outlined in the preceding sections, there will be many situations where we have to be content with inexact algorithms. It is likely always to be true that exact and verified methods are significantly less efficient than inexact methods.

In this realm, the question of *a posteriori* analysis arises. Given an instance of the problem and a solution, we may want to analyze the quality of the solution. As an example, consider the problem of finding the roots of a univariate polynomial $f(x)$ of degree n. Given approximate solutions $x_1, \ldots, x_n$, compute the quantities [Smi70]

$$\sigma_i = \frac{f(x_i)}{\prod_{j \neq i}(x_i - x_j)} \quad \text{for } 1 \leq i \leq n \; .$$

Let Γ_i be the disk in the complex plane centered at x_i with radius $n|\sigma_i|$. Then the union of the disks contains all roots of f. Moreover, a connected component consisting of k disks contains exactly k roots of f. The disks Γ_i give a posteriori analysis of the quality of root estimates and the σ_i are easily computed with controlled error using multi-precision floating point arithmetic.

Are there analogous examples in the realm of combinatorial computing. Clearly, in the area of approximation algorithms, one frequently computes *a priori* bounds. The nice feature of the example above is that the approximate solution plays essential part in estimating its quality.

[3]The author does not recall where he learned about this example.

4.8 Test and Repair

Sometimes we can even do better than just a posteriori analysis. We might be able to take the approximate solution returned by an inexact algorithm as the starting point for an exact algorithm.

Consider the linear programming problem

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

where x is a vector of n real variables, A is an $m \times n$ matrix, $m < n$, b is an m-vector, and c is an n-vector. For simplicity, we assume A to have rank m. It is well known that one can restrict consideration to basic solutions. A basic solution is defined by an $m \times m$ non-singular sub-matrix B of A and is equal to (x_B, x_N) where the x_B are the variables corresponding to the columns in B, $x_B = B^{-1}b$ and $x_N = 0$. A basic solution is primal feasible if $x_B \geq 0$, and is dual feasible if $c_B^T - c_N^T A_B^{-1} A_N \leq 0$. It is optimal if it is primal and dual feasible.

For medium-scale linear programs, we succeeded in checking (exactly !!!) in reasonable time whether a given basis is primal or dual feasible [DFK+02]. This suggests the following approach. Use an inexact LP solver to determine an "optimal" basis B. Check the basis for primal and/or dual feasibility. If so, declare it optimal. If not and the basis is X-feasible ($X \in \{\text{primal,dual}\}$), use an exact X-simplex algorithm starting at B to find the true optimum. If B is neither primal nor dual feasible, I do not really know how to proceed; one can first use the primal simplex method to find the optimum of the subproblem defined by the satisfied constraints and then use the dual simplex method to add the remaining constraints, but this is not really satisfactory. The hope is that the inexact method find a basis B that is close enough to the optimum, so that even a slow exact algorithm can find it. Table 1 supports this hope. The last column shows the time required to check whether the basis returned by CPLEX is optimal, and if not, to obtain the optimal basis from it. Column V shows the number of constraints violated by the basis returned by CPLEX.

The general question for optimization problems is the following. 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 given and the optimal solution.

5 Conclusion

Reliability (trustworthiness) is a desirable feature of humans and also programs. I love to use TEXand one of the reasons is that it never crashes. Many users of LEDA tell me that the reliability of its programs is important to them and that they do not really care about the speed.

The strive for reliability poses many hard and relevant scientific questions. Part of the answers can be found within combinatorial algorithmics, but for many of them we have to reach out and make contact with other areas of computer science: numerical analysis, computer algebra, and even semantics and program verification.

References

[BEH+02] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. to appear in ESA 2002, www.mpi-sb.mpg.de/~mehlhorn/ftp/ConicPolygons.ps, 2002.

[BFM+01] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *ESA 2001*, Lecture Notes in Computer Science, pages 254–265, 2001. http://www.mpi-sb.mpg.de/~mehlhorn/ftp/ImprovedSepBounds.ps.gz.

[BK89] M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the 21th Annual ACM Symposium on Theory of Computing (STOC'89)*, pages 86–97, 1989.

[BW96] M. Blum and H. Wasserman. Reflections on the pentium division bug. *IEEE Transaction on Computing*, 45(4):385–393, 1996.

[CCPS98] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Inc, 1998.

[CGA] CGAL (Computational Geometry Algorithms Library). www.cgal.org.

[CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences*, 30(1):54–76, 1985.

[DFK+02] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, and D. Weber. Certifying and repairing solutions to large LPs, How good are LP-solvers? to appear in SODA 2003, www.mpi-sb.mpg.de/ mehlhorn/ftp/LPExactShort.ps, 2002.

[HT74] J.E. Hopcroft and R.E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 1974.

[KCF+02] J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha. ESOLID: A system for exact boundary evaluation. In *7th ACM Symposium on Solid Modelling and Applications*, pages 23–34, 2002.

[KLPY99] V. Karamcheti, C. Li, I. Pechtchanski, and Chee Yap. A core library for robust numeric and geometric computation. In *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pages 351–359, Miami, Florida, 1999.

[KMMS02] D. Kratsch, R. McConnell, K. Mehlhorn, and J.P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. SODA 2003 to appear, www.mpi-sb.mpg.de/ mehlhorn/ftp/intervalgraph.ps, 2002.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs, International Symposium, Rome*, pages 215–232, 1967.

[LED] LEDA (Library of Efficient Data Types and Algorithms). www.algorithmic-solutions.com.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.

[MNS+96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking geometric programs or verification of geometric structures. In *Proceedings of the 12th Annual Symposium on Computational Geometry (SCG'96)*, pages 159–165, 1996.

[OTU87] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In Derick Wood, editor, *Proceedings of the 3rd Annual Symposium on Computational Geometry (SCG '87)*, pages 119–125, Waterloo, ON, Canada, June 1987. ACM Press.

[OW96] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 1996.

[SM90] G.F. Sullivan and G.M. Masson. Using certification trails to achieve software fault tolerance. In Brian Randell, editor, *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS '90)*, pages 423–433. IEEE, 1990.

[Smi70] Brian T. Smith. Error bounds for zeros of a polynomial based upon Gerschgorin's theorems. *Journal of the ACM*, 17(4):661–674, October 1970.

[VG99] Joachim Von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999. Chapters 1 and 21 cover cryptography and public key cryptography.

[WB97] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

[Yap99] C.K. Yap. *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 1999.