J. Symbolic Computation (1994) **17**, 297–310

Complexity, Algorithms, Programs, Systems: The Shifting Focus

JURG NIEVERGELT

Department of Computer Science, ETH Zurich

(Received 18 February 1994)

We investigate the changing relationship between the small research community of theoretical computer scientists and the much larger community of computer users, in particular, the technology transfer problem of how to exploit theoretical insights that can lead to better products. Our recommendation can be summarized in four points:

1 The computing community is impressed by usable tools and by little else. Although a powerful theorem or an elegant algorithm may be a useful tool for a fellow theoretician, by and large, the only tools directly usable by the general computing community are *systems*. No systems, no impact!
2 System development means programming-in-the-large, but the algorithms research community so far has learned only how to program in-the-small.
3 Algorithm researchers must enshrine their algorithms not merely in individual elegant programs, but collectively in useful application packages aimed at some identifiable user group.
4 Since the development of software systems easily turns into a full-time activity that requires different skills from those of algorithms research, we must strive to develop techniques that lets a small group of algorithm researchers develop simply structured, open-ended systems whose kernel can be implemented with an effort of the order of 1 man-year. Low-complexity systems is the goal!

Contents

0747–7171/94/040297 + 14 $08.00/0

1. The broad spectrum of algorithm research: From algorithm complexity to system complexity

Workshops on algorithms have often been dominated by complexity issues with little regard for practicality, but this ALCOM Workshop on "Algorithms: Implementation, Libraries, and Use" may be indicative of a new trend. In the past two days we have not only heard about programs and systems, we have actually seen them in operation. Impressive graphical animations have been used to illustrate the experimental nature of practical algorithms. We saw program libraries that run on systems with integrated user interaction and algorithm animation, features that I believe belong to any system or program library, no matter what its application domain. So on the progression indicated in the title: Complexity, algorithms, programs, systems, this subgroup of the algorithm research community is now focusing on stages 3 and 4, and that is a big step forward from just a couple of years ago. In order to understand where the field is going it is instructive to retrace important past developments.

Algorithmics is concerned with a broad spectrum of issues ranging from the most theoretical to the highly practical: Models of computation; the analysis of problems that can be stated in rigorous mathematical terms; the design and analysis of algorithms for solving a given problem using the primitive operations made available by such a model of computation; and, last but not least, correct, robust and efficient implementation and experimental testing.

Among the contributions of algorithmics to computer science and to computing technology, abstract results about the inherent complexity of problems and algorithms are among the most fundamental. Most of these are a mere quarter century old, but we are convinced of the timeless importance of such insights as the large class of optimal $n \log n$ algorithms, of the problem classes P and NP, and of PRAM complexity. The algorithms community knows that such insights have implications for the practice of computing as drastic as the conservation laws of physics have for engineering but most computer users and programmers do not, and even the computer science community at large attaches greater importance to annual technology improvements and marketing successes than to timeless theoretical results.

If I am correct in this somewhat pessimistic assessment of the impact algorithmics has had so far on the computing "culture", where is the bottleneck that holds up the flow of ideas from theory to products? Have we failed to impress on our students the importance of the theoretical foundations of computer science? Is the user community still dominated by old-timers happy with COBOL and Fortran? Can one be a competent computer scientist without any knowledge of algorithms and complexity theory? Do other subdisciplines of computer science that also have a theory/practice dichotomy face the same problem of users ignoring the theory that supports practice?

Or perhaps everything is as it should be? If not, what can we do to educate the computing community towards a proper appreciation and use of theory?

These are some of the questions we shall explore. The line of reasoning can be followed more easily if we jump ahead to some concisely stated conclusions, even at the risk of oversimplification:

1 The computing community is impressed by usable tools and by little else. Although a powerful theorem or an elegant algorithm may be a useful tool for fellow theo-

reticians, by and large, the only tools directly usable by the general computing community are *systems*. No systems, no impact!

2 System development means programming-in-the-large, but so far the algorithm research community has learned only how to program in-the-small.

3 Algorithm researchers must enshrine their algorithms not merely in individual elegant programs, but collectively in useful application packages aimed at some identifiable user group.

4 Since the development of software systems easily turns into a full-time activity that requires different skills from those of algorithms research, we must strive to develop techniques that lets a small group of algorithm researchers develop simply structured, open-ended systems whose kernel can be implemented with an effort of the order of 1 man-year. Low-complexity systems is the goal!

This is the message and the mission of my talk, to be supported with illustrative examples.

2. From ideas to users: Technology transfer in various subdisciplines of computer science

How well have various computer science research communities marketed their theoretical insights as products for users? A few examples suffice to make the point that many innovations survive as products if (and only if?) they they are developed to the stage where they can be marketed as systems or are integrated as components into some widely used system. Readers will undoubtedly come up with many instructive examples that fit this rule, and some that don't – I would be interested in hearing about both kinds.

2.1. Programming languages

This research community has been very effective for decades in quickly turning theoretical ideas into successful products, as countless features of today's programming languages testify. Just a few examples may serve to remind us that what we take for granted today was a research problem in the recent past.

Automatic formula translation, a non-trivial process in the 50s, has been serving programmers well ever since it was incorporated into FORTRAN 1954. Needless to say, the vast majority of application programmers is unaware of how it is done, and would not bother to study the algorithm.

The formal definition of syntax, using Backus-Naur Form or similar notations (e.g. syntax diagrams), was developed for Algol 60 and has been an unavoidable tool ever since. On the other hand, formalisms developed by logicians or theoretical linguists that were not tied to some widely used programming language remain the well-kept secret of specialists.

Recursive procedures, and the techniques needed to process them, were first understood during the 50s and have been part of most high level programming languages since Algol 60. Recursive functions of symbolic expressions gave rise to LISP and functional programming languages.

A "calculus" of data type definitions emerged during the 70s and was immediately taken up by Pascal. We now take it for granted that every programmer defines his own data types on top of the base types provided by his programming language.

Logic programming became widely known once this idea inspired the development of languages such as Prolog and many rule-based systems.

Object-orientation can be traced at least to the (special-purpose) language Simula in the 60s. The idea took off like brushfire as it was being enshrined in popular extensions to existing languages, such as the step from C to C++, or from Pascal to Object Pascal.

Besides such technology transfer or marketing successes, the field of programming languages has of course also generated its fair share of stillborn creations. For example, the desire for "universal languages" that are "all things to all people", or wishful thinking about "programming in English" or in DWIM – "do what I mean, not what I say". The filter of time works in a funny way: success stories as well as failures are forgotten, but with a difference – successes make it into products that are taken for granted, failures become irrelevant.

Let us sample some other disciplines even more sporadically, hoping that some historian of computer science will trace the development and spread of important innovations more systematically.

2.2. Computer systems (architecture and operating systems)

An impressive multitude of architectural ideas spawned a zoo of diverse machines (as described two decades ago in Gordon Bell and Allen Newell: Computer structures: Readings and examples, McGraw-Hill 1971). The fact that fewer innovations survive than die out in the marketplace is part of life – they all had to be tried for the process of selecting the fittest to run its course. In the field of operating systems there has perhaps been less experimentation, but the success stories are impressive: Unix, the operating system used on the largest variety of machine types, originated with research tinkerers.

2.3. Data base systems

Data base systems provide an architectural structure at the level of the application designer and data base administrator. This field exhibits a track record comparable to that of computer systems. The seminal concept of relational data bases prospered within two decades to dominate commercial data base software.

2.4. User interfaces

Innovative concepts of the seventies, such as WYSIWIG (what you see is what you get), the desktop metaphor, direct manipulation, and the mouse, may have languished in the research labs for too long, but after the Macintosh's market break through, they spread like fire.

2.5. Algorithmics

The track record of algorithmics shows success stories as well. A modest but early technology transfer occurred when sorting was a fruitful research topic that created dozens of efficient sorting algorithms: Sort generators select an appropriate algorithm based on a specification of the files to be sorted and the system configuration, without requiring the user to know anything about $n \log n$. The technology transfer of greatest impact came from the field of numerics, in the form of program libraries for linear algebra

or differential equations without which modern scientific and engineering computation would be impossible. A more recent success story is symbolic computation, where systems such as Mathematica and Maple have found widespread use.

It would be interesting to study a larger and more representative sample of research ideas turned to products. My conclusion from this attempt at understanding the reason for the varying degree of successful spread of computer science research ideas into practice is:

Potentially useful ideas travel all the way from concept to end users if and when their creators package these ideas in the form of ready-to-use systems!

3. Algorithms programmed in-the-small

In the early days of automatic computing, in the 50s and early 60s, numerics researchers usually considered it to be their job to transform theoretical insights, such as a new equation solver, into portable programs freely available: Communications of the ACM and various GAMM journals regularly published complete, ready-to-run Fortran and Algol programs. This tradition came to an end in the late sixties when a split emerged: A few groups of specialists went into full-time professional development of numeric program libraries, whereas the majority of algorithms researchers, by now mainly focused on discrete combinatorial problems, was satisfied with theoretical results.

In retrospect the lure of theory is understandable. Around 1960 it was by no means obvious that an algorithm could be studied as a mathematical object to the extent that we now take for granted. The most rigorous mathematical results about algorithms had to do with rates of convergence and error propagation. Results of both types only involve analyzing execution "once-around-the-innermost-loop", so they depend more on analyzing an arithmetic expression than a full-blown program. The possibility of asymptotic performance analysis and lower bound techniques were a revelation, followed by natural curiosity about how far these mathematical techniques would carry.

They carried far, a full three decades worth of remarkable progress – but they also carried too far. The split between the computing community and the theoretical community had the unfortunate consequence that theory distanced itself increasingly from the practice of computing and its applications. Examples of these excesses are "optimal algorithms" (in some technical sense) so complicated, uncertain, and untested that no one would consider them for actual use. And those theoretical results of potential value to practical computing had to travel such a long distance from discovery to actual use that most never arrived at the destination but remained buried in the research literature.

Although most algorithm researchers stop when they have proven asymptotic optimality, a minority has persisted in pushing the realization of an algorithm all the way to the *program optimization stage of mapping mathematical operations to machine primitives*. Among the prominent exceptions who deemed research incomplete unless their algorithms had been turned into programs controlled down to the last bit, Don Knuth stands out: his encyclopedic "Art of computer programming" (note the revealing title) presents many algorithms as programs written in the assemly language MIX.

To this minority of algorithm researchers and programmers we owe beautiful examples that show the intellectual elegance to aim at when programming in-the-small, i.e. writing programs that range from a few lines to a few pages of code. A good book on algorithms and data structures should be a treasure trove of such "programming pearls" (title of Bentley's books). Two examples illustrate this point.

1) **Warshall's** well-known **transitive closure** algorithm for a graph of n nodes given by its adjacency matrix A is best explained as computing a sequence of matrices $B_0 \ldots B_n$:

B_0 := adjacency matrix A;
for $k := 1$ to n do
  $B_k[i,j] = B_{k-1}[i,j]$ or $(B_{k-1}[i,k]$ and $B_{k-1}[k,j])$.
connectivity matrix $C := B_n$

It is programmed elegantly and efficiently to work in-place, thereby saving memory and copy operations:

```
procedure warshall(var a : array [1..n, 1..n] of boolean);
var i, j, k : integer;
begin
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                a[i,j] := a[i,j] or (a[i,k] and a[k,j])
end;
```

Because the assignment mixes values of the old and the new matrix, we get a program for a *different* algorithm, which must again be proven correct with new arguments. This example makes the point that an algorithm must be looked at from different perspectives depending on whether you express it for the benefit of people, i.e. in a form most readily understood, or as a program aimed at efficient machine consumption. The form most suitable when using conventional mathematical notation is not necessarily the best possible program!

2) I often use the second example for surprise effect in class. The **logarithmic bit sum** program claims to compute an integer equal to the number of 1s in a bitstring packed into a machine word (Nievergelt and Hinrichs, 1993). When some CDC 6000 series computers of the 70s suddenly had an operation "population count" much faster than earlier versions, Control Data must have rediscovered the logarithmic bit sum algorithm:

```
function logbitsum(w : w16) : integer;
const mask[0] =' 0101010101010101';
      mask[1] =' 0011001100110011';
      mask[2] =' 0000111100001111';
      mask[3] =' 0000000011111111';
var   i, d : integer; weven, wodd: w16;
begin
      d := 2;
      for i := 0 to 3 do begin
          weven := w ∩ mask[i];
          w := w/d; {shift w right 2^i bits}
          d := d^2;
          wodd := w ∩ mask[i];
          w := weven + wodd;
      end;
```

```
    return(w)
end;
```

This code is mysterious, to say the least, and purists might reject it out of hand because of the type conflict in using variable w both as an integer and as a bit-vector.

It comes as a surprise that the bit-acrobatics above is the logical outcome of the divide&conquer paradigm: bitsum(w) = bitsum(left half of w) + bitsum(right half of w). However, the useful algorithm design principle d&c alone would be counterproductive for this problem if used sequentially. Only when combined with program optimization based on the fact that Boolean operations on a word are executed bit-parallel on most computers does it achieve surprising elegance and efficiency. Figure 1 illustrates this lemma.

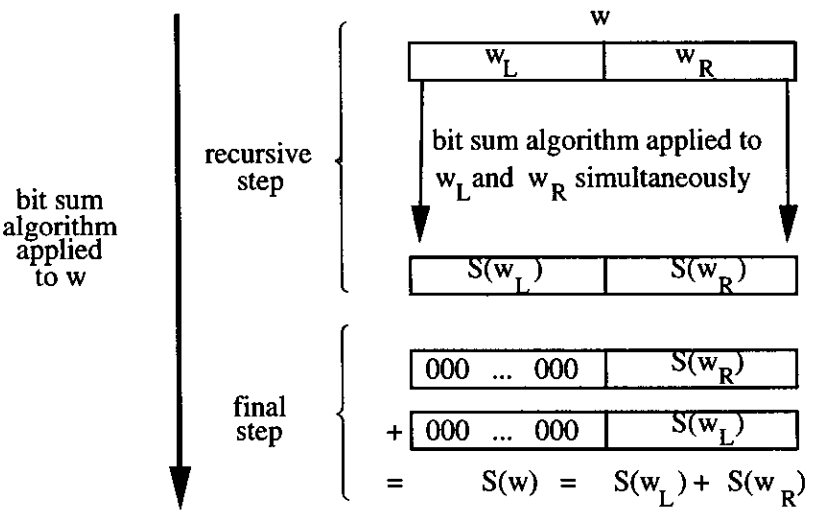

Figure 1. Bitsum Algorithm

I recall these examples to make the point that the community of algorithm researchers, or at least some of them, have learned very well indeed how to program in-the-small. Unfortunately, their feats of programming artistry were accessible and useful only to a small group of peers. Even today the computing community at large hardly knows that there is a rich collection of elegant algorithms that can be implemented very efficiently in just a few pages of code. In contrast, this same computing community has heard all about object-oriented programming languages and data bases, and knows more about the latest release of Excel, or Word, or Windows than I do. That is because, with a few exceptions, we failed to package our elegant and efficient algorithms into end-user applications but left them to be discovered in text books or in the research literature.

4. What is a system?

Developing a system means programming-in-the-large, a much tougher skill to learn than programing-in-the-small. It took the software community decades to learn that

useful systems need not be large. Our profession started out in the 50s and 60s developing systems with the motto "the bigger the better" emblazoned on our flag. The resulting crises led to the software engineering crusades of the 70s, one of whose positive lessons, for those with their eyes open, was the counter-motto "the smaller and simpler the better". My colleague Niklaus Wirth certainly owes much of his success in software engineering to the fact that his programming languages, while incorporating some concepts that were new at the time, always remained Spartan. By focusing on only a few key concepts, mainly the then-novel issue of data types, Pascal could be implemented by portable single-pass compilers and ended up having a greater impact than PL/1, the "all-purpose language" de signed to beat all other languages on their own turf. Similarly, Unix started out as a small, simple operating system. It could only afford to explode after it had gained wide acceptance and system administrators were hooked to it "for better or for worse, until death do us part".

Accepting the fact that a "system" cannot be a small program, and in particular has more code than an individual algorithm, size is neither desirable nor a distinguishing feature of a "system". Some of the characteristics of a system are:

A system must help its user perform a **variety of related tasks**, not just one particular data transformation. An exact a priori input-output specification (as required for correctness proofs, say) is not practical; rather, the choice of functions to be offered by a system is a task to be tackled with experience, intuition, and trial-and-error.

A system is **more than a mere collection of independent programs** whose union happens to cover all the functionality we seek. At the very least we want data-compatibility among all the programs in a system, so a common data model is a key requirement.

A successful system will be used by a **variety of users** with different goals, knowledge, and skills. A novice wants to browse thru it, whereas an expert expects to find powerful tools.

A system is **never finished – it adapts or dies!** Hence it must be designed for constant modification, including the potential for growth, and must be maintained. (Some argue that because software, unlike hardware, does not degrade on its own, it needs no maintenance. It does, because it must be adapted to ever-changing expectations and environments).

The lifetime of a system, from conception to obsolescence, is of the order of a decade. Hence those responsible for the system must **nurture human expertise** that survives that long.

Any system needs an **interactive component**. Today, this suggests that a powerful graphical user interface might as well be designed into it from the beginning. This holds even for a mere library of independent programs to be called by client programs. Any system is enhanced significantly by the presence of an interactive data visualization or animation system so the end-user can explore and "get a feel for" the results produced.

System development is a more demanding task than programming-in-the-small in at least two important respects. Technically because, unlike for a collection of in dependent programs, a system designer must solve many compatibility problems in such a way as to not impair achievable performance – many intellectually elegant designs have failed as

systems for this reason of built-in inefficiency, e.g. Algol 68. Secondly, a system creator's task, unlike a theoretician's task, is not finished when the technical labor is complete: the system has to be introduced to an appropriate user group who knows what to do with it. This marketing task calls for different skills and suits the character of few technically inclined people.

5. Sampling the zoo of systems: Experiences and lessons

An underpopulated niche of computer science has attracted my attention from the beginning of my career: The bridge that leads from theory to system development. This niche is underpopulated mainly because it calls for skills and styles of thinking that appear mutually exclusive:

> **Theory** calls for a heavy dose of mathematical thinking and rigorous logic.
> **System development** at the design stage calls for conceptual thinking-in-the-large that can or must be vague and uncertain; and at the other extreme, when implementing key routines, calls for a hacker's love of detail, even when there is no rhyme nor reason to the particular details one has to live with.

With the goal of working on projects that bridge the gap between theory and systems, I have had the good fortune of having been involved in system development for three decades in a variety of capacities, starting in 1962-64 as one of three graduate students designated to implement the mathematical library of a then-supercomputer, Illiac 2. What little system integrity this rudimentary library ended up having was to be found mainly in the design of the "calling sequence" and in recommended common ways of using the 8 registers of the machine and of allocating vectors and matrices. The three major tools any Illiac 2 programmer depended on, the loader, the assembler, and the program library, formed a rudimentary but effective tiny "system".

A decade and several learning experiences later, from 1970 to 77, I led a group of up to two dozen faculty and research assistants to develop ACSES, an automated computer science education system, on PLATO, the University of Illinois' pioneering CAI (computer-assisted instruction) system (Nievergelt, 1975 and Nievergelt, 1980). I learned to grapple with big systems, concluded that every system should be interactive, and saw the design of user interfaces as a major challenge.

ACSES had a big local impact: For over a decade, more than 1000 students per semester used it in a variety of introductory CS courses which had been changed from the original 3 lecture hours per week to a single lecture hour per week, augmented by course-work and exercises accomplished on PLATO. This early integration of a computer system into the teaching operation was only possible thanks to the committed support of the head of the CS department and the dean of the engineering college.

Although the massive instructional use of ACSES went well, keeping a big system like ACSES running on the yet bigger system Plato was a nerve-racking organizational and maintenance chore. Having made my experience with dangerously large systems, during 1976-84 at ETH Zurich I focused my group's work on the development of a series of experimental interactive systems for education, XS-0, 1, and 2. These implementations were conducted as integrated group projects, with 4-6 people working closely together according to detailed design specifications developed jointly. The experience was in stark contrast compared with the previous project on PLATO: managing the development

was fun, with plenty of unconstrained opportunity to explore research ideas (Nievergelt and Weydert, 1980). Without an institutional commitment and thus a captive audience, however, the educational impact of these systems was minimal. The closed, "take-it-or-leave-it" turn-key-nature of these systems discouraged others from tinkering with them, and so we lost potential customers knowledgeable enough to be able to tailor the system to their own wishes.

Looking for ways to simplify the management of software projects, I am now using another model of system development: the "1-man-system". The idea is to conceive a system kernel that is as small and Spartan as you can possibly make it. It need not be able to do much, but it is executable and demonstrable – in other words, it does something you can show to interested people, to potential "customers". By itself, it does not provide the end functions the user wants – it merely provides a stepping stone from which the user can get to his goal more easily than if he started from the conventional platforms available today, typically a programming environment consisting of an operating system, programming languages, and various editors. A "cook-it-yourself restaurant" may be an apt analogy: the restaurant provides the meat, fire, and sauces; the customer picks the ingredients and uses the grill to adapt the steak to his own specification. He achieves his goal more quickly, and perhaps more reliably, than if he started from the raw materials available on the ranch. Many customers can use the same infrastructure simultaneously to achieve individually tailored results.

Because the kernel's functionality is limited, its design and implementation can be assigned to one person. In the academic world this is a PhD student, of course, who devotes his primary activity to this one project for a few years. Any number of others can build applications on top of the kernel, small or large. The kernel enhanced by one these application modules becomes an end-user system of respectable power. Let me mention two such kernel systems developed in recent years, and applications thereof.

5.1. The Smart Game Board (Anders Kierulf et al)

The Smart Game Board (Kierulf, 1990 and Kierulf, Chen, and Nievergelt, 1990) is a computerized board and a programmer's workbench for developing two-person board games. The architecture clearly separates the game-independent kernel and the game-specific modules needed to implement a new game on top of the Smart Game Board.

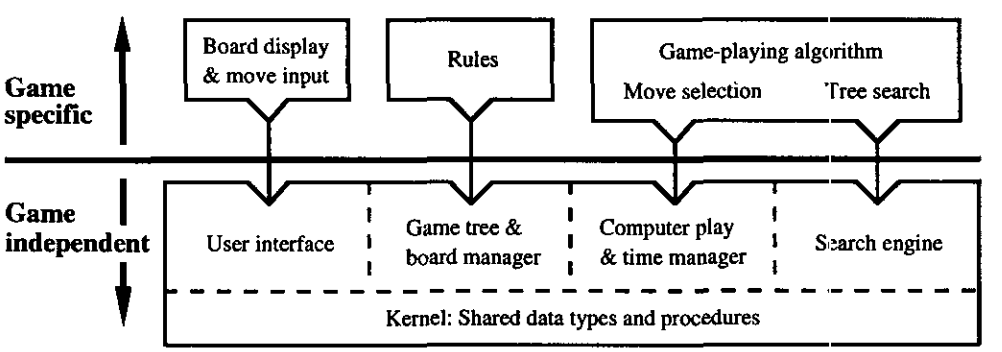

Figure 2. The Smart Game Board

So far chess, Go (Kierulf and Nievergelt, 1989 and Chen, Kierulf, Müller, and Nievergelt, 1990), Othello, Go-Moku, and Nine-Men's Morris (Gasser, 1991) have been implemented, with hundreds of game fans regularly using this software for many functions they normally perform using a wooden board and paper: Playing, analyzing, teaching, annotating, organizing and storing game collections.

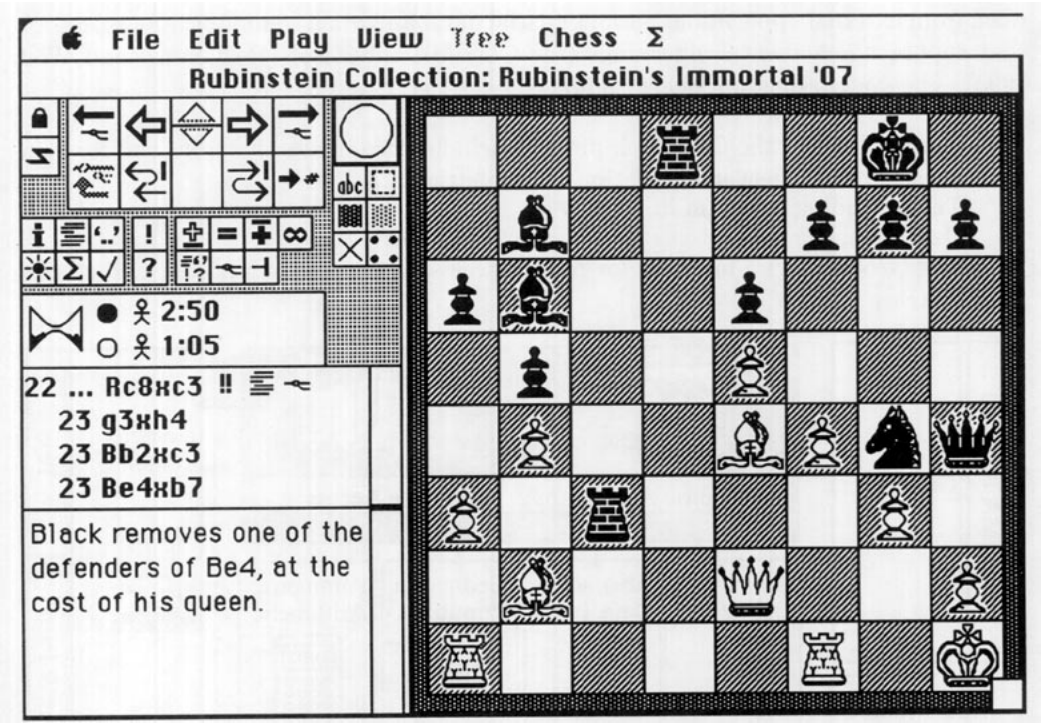

Figure 3. The Smart Game Board: Chess

Most of the Smart Game Board's user interface is embodied in a control panel common to all games (shown at the top left in the figure with the chess board). Game-independent operations are defined as motions on a game tree. Game-specific operations (e.g. setting up chess positions, marking points on the Go board for an notations) and status information (e.g. castling status in chess) are provided in a menu specific for each game.

The Smart Game Board is a sophisticated tool that requires a few weeks of learning effort but then repays this investment with interest. Once the student understands the interfaces, he can plug small game-specific modules into the much larger game-independent part to create as a term-project some game-playing program he could not have implemented without this toolkit.

5.2. XYZ GeoBench (Peter Schorn et al)

Schorn's XYZ GeoBench (Schorn, 1991 and Schorn, 1994) is another well-tested "one-man-system" that has been extended by dozens of other implementors (Nievergelt, Schorn,

De Lorenzi, Ammann, and Brüngger, 1991). Such contributions range from a library program for a single computational geometry algorithm to application systems built on top of the GeoBench such as an office-space allocation package, a prototype workbench for molecular modeling, and a GeoServer. In co-operation with the database group of H.-J. Schek, and as a part of the Esprit project "Algorithms, Models, User and Service Interfaces for Geography," De Lorenzi is extending the GeoBench to a server specialized in performing geometric operations efficiently. The GeoServer can be called over the network from external applications such as spatial databases via a communications protocol that supports incremental algorithms (the COSIMA Interface) (De Lorenzi and Wolf, 1993). The system diagram below shows:

The modules of the GeoBench proper, including the interactive front end for visualization and direct manipulation of geometric data.
The extendible program library.
Data management systems such as the Grid File.
The GeoServer, an interface for remote access to the program library and storage systems.

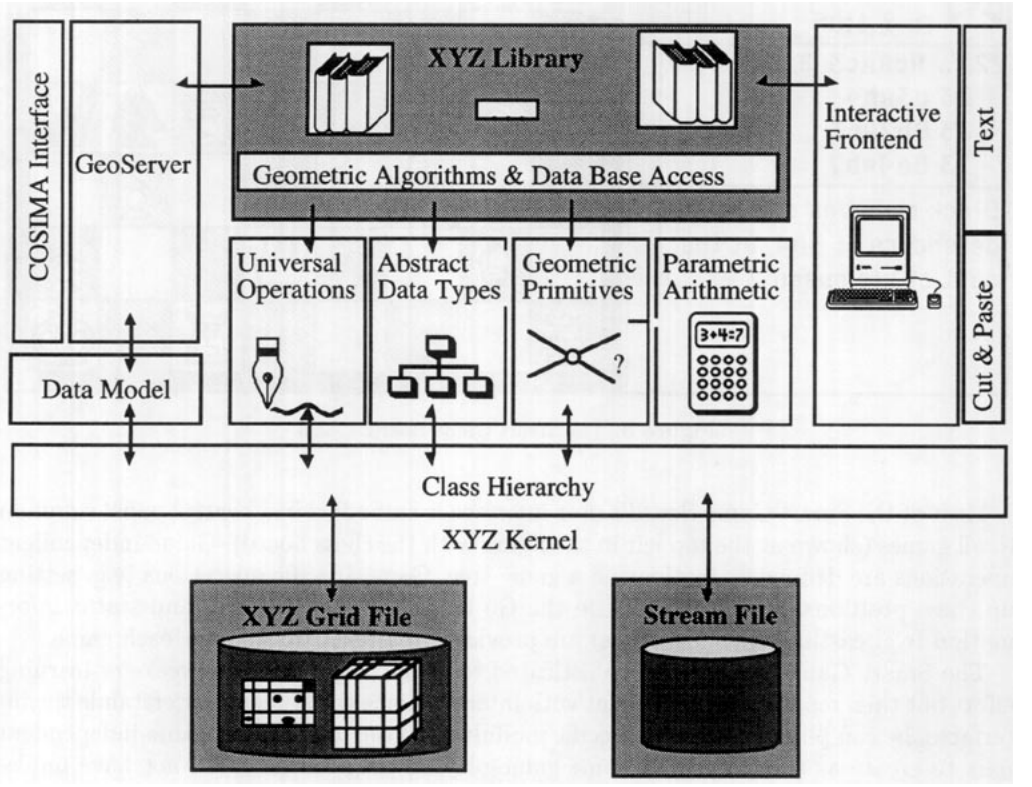

Figure 4. The GeoBench

The experience with this approach of building compact "one-man" kernel systems, to be extended by others who work closer to some application domain, has been more successful than any I had tried before. We continue to use this recipe, and the next such system has already proven itself. Ralph Gasser's SearchBench for retrograde analysis has

recently completed an exhaustive search of the state space of the game variously called Nine Men's Morris or Merrils (Muehle, moulin) with 10^{10} states. The result of months of computation: If played optimally, Merrils is a draw. The SearchBench has also been used to analyze chess endgames and mathematical puzzles.

6. Conclusion: Three types of results

This workshop has shown impressively that enterprising algorithm researchers have embraced systems development as a marketing outlet for our work. This growing emphasis on product orientation is long overdue. I am not saying that we should try to force every theoretical idea should into a system. What I have tried to say is that in our work we might profitably distinguish three types of projects, and shift the balance away from an excessive concern with only one or two of these, in particular away from unimplemented algorithms that merely yield asymptotic bounds without any indication that they are practical:

1 The purely **theoretical result**, packaged in the time-honored mold "definition-theorem-proof". Forget software.
2 An **algorithm**, implemented as a stand-alone "proof-of-concept-program" for the primary benefit of the author. As we have heard repeatedly at this workshop, test implementation and test runs often yield profound insights that theory misses.
3 **Application systems!** With results of types 1 and 2 we may impress our fellow researchers. If we seek an impact beyond our own small circle, we must seize every reasonable opportunity to turn our work into a product useful to those outside the circle. Such an applications orientation has been a guiding principle for many of the talks at this workshop. The following steps may guide those who choose to follow this path:

   Identify a subject area with potential applications where theory is progressing rapidly and expertise is not yet widely available. Surely every theoretical computer scientist can think of candidates.
   Select a target audience from which you can find partners and customers.
   Design the desired functionality of a systems: what concepts, objects, operations must it embody?
   Design a kernel systems that can be prototyped in about 1 man-year, with an interface for an open-ended collection of application programs.

   This much for the technical part of the job. Next ...

   Find or coerce developers to build applications on top of your kernel system. But that's another story beyond my competence, get Bill Gates to talk about marketing software!

Acknowledgements

This abridged text of an invited talk may serve as an editorial introduction to the special issue dedicated to papers presented at the ALCOM Workshop on "Algorithms: Implementation, Libraries, and Use" held at Dagstuhl, August 16-18, 1993. Thanks to the organizers Kurt Mehlhorn and Stefan Naeher, and to Peter Schorn, Nora Sleumer, and several participants for their work and comments.

References

Chen, K., Kierulf, A., Müller, M., Nievergelt, J. (1990). The design and evolution of Go Explorer. In: Marsland, T. A., Schaeffer, J. (eds.) *Computers, Chess, and Cognition*. Springer.

De Lorenzi, M., Wolf, A. (1993). A protocol for spatial information managers. *Proc. Int. Workshop on Interoperability of Database Applications*. Fribourg.

Gasser, R. (1991). Applying Retrograde Analysis to Nine Men's Morris. In: Levy, D. N. L., Beal, D. F. (eds.): *Heuristic programming in artificial intelligence 2: The 2nd Computer Olympiad*. Ellis Horwood, Chichester, 161–173.

Kierulf, A., and Nievergelt, J. (1989). Swiss Explorer blunders its way into winning the first computer Go Olympiad. In: Levy, D.N.L., Beal, D.F. (eds.): *Heuristic programming in artificial intelligence: The First Computer Olympiad*. Ellis Horwood, Chichester (1989) 51–55.

Kierulf, A. (1990). *Smart Game Board: A Workbench for Game-Playing Programs, with Go and Othello as Case Studies*. Diss. ETH Zurich.

Kierulf, A., Chen, K., Nievergelt, J. (1990). Smart Game Board and Go Explorer: A study in software and knowledge engineering. *Comm. ACM*. **33, 2**, 152–166.

Kierulf, A., Gasser, R., Geiser, P., Müller, M., Nievergelt, J., Wirth, C. (1991). Every interactive system evolves into hyperspace: The case of the Smart Game Board. In: Maurer, H. (ed.): *Proc. Hypertext / Hypermedia 1991*. Springer, 174–180.

Nievergelt, J., Hinrichs, K. (1993). *Algorithms and Data Structures, with Applications to Graphics and Geometry*. Prentice-Hall.

Nievergelt, J. (1975). Interactive systems for education – The new look of CAI. Invited paper, *Proc. IFIP Conf. on Computers in Education*. North Holland, 465–472.

Nievergelt, J. (1980). A pragmatic introduction to courseware design. *IEEE Computer* **13, 9**, 7–21.

Nievergelt, J., Weydert, J. (1980) Sites, modes and trails: Telling the user of an interactive system where he is, what he can do, and how to get places. In: Guedj, R. A., (ed.): *Methodology of Interaction, Proc. IFIP Workshop, Seillac 79*. North Holland, 327–338. (Reprinted in: Baecker, R. M., Buxton, W. (eds.): *Readings in Human-Computer Interaction*. Morgan Kaufmann, 1987).

Nievergelt, J., Schorn, P., De Lorenzi, M., Ammann, C., Brüngger, A. (1991). XYZ: A project in experimental geometric computation. In: Bieri, H., Noltemeier, H. (eds.): *Computational Geometry: Methods, Algorithms and Applications, Proc. CG '91, International Workshop on Computational Geometry, Bern, March 1991*. Springer LNCS, 171–186.

Schorn, P. (1991). *Robust algorithms in a program library for geometric computation*. Diss. ETH Zurich.

Schorn, P. (1994). Evolution of a software system: Interaction, Interfaces, and Applications in the XYZ GeoBench. In this issue.