Algorithm Engineering[*]

Camil Demetrescu[§] Irene Finocchi[¶] Giuseppe F. Italiano[||]

Abstract

Algorithm Engineering is concerned with the design, analysis, implementation, tuning, debugging and experimental evaluation of computer programs for solving algorithmic problems. It provides methodologies and tools for developing and engineering efficient algorithmic codes and aims at integrating and reinforcing traditional theoretical approaches for the design and analysis of algorithms and data structures.

1 Introduction

In recent years, many areas of theoretical computer science have shown growing interest in solving problems arising in real-world applications, experiencing a remarkable shift to more application-motivated research. However, for many decades, researchers have been mostly using mathematical methods for analyzing and predicting the behavior of algorithms: asymptotic analysis in the Random Access Model has been the main tool in the design of efficient algorithms, yielding substantial benefits in comparing and characterizing their behavior and leading to major algorithmic advances. The new demand for algorithms that are of practical utility has now raised the need to refine and reinforce the traditional theoretical approach with experimental studies, fitting the general models and techniques used by theoreticians to actual existing machines, and bringing algorithmic research back to its roots. This implies to consider often overlooked, yet practically important issues such as hidden constant factors, effects of the memory hierarchy, implications of communication complexity, numerical precision, and use of heuristics.

The whole process of designing, analyzing, implementing, tuning, debugging and experimentally evaluating algorithms is usually referred to as *Algorithm Engineering*. Algorithm Engineering views algorithmics also as an engineering discipline rather than a purely mathematical discipline. Implementing algorithms and engineering algorithmic codes is a key step for the transfer of algorithmic technology, which often requires a high-level of expertise, to different and broader communities, and for its effective deployment in industry and real applications. Moreover, experiments often raise new conjectures and theoretical questions,

[*]Work partially supported by the IST Programmes of the EU under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33555 (COSIN), and by the Italian Ministry of University and Scientific Research (Project "ALINWEB: Algorithmics for Internet and the Web").

[§]Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. Email: `demetres@dis.uniroma1.it`, URL: `http://www.dis.uniroma1.it/~demetres/`.

[¶]Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via di Tor Vergata 110, 00133 Roma, Italy. Email: `finocchi@disp.uniroma2.it`, URL: `http://www.dsi.uniroma1.it/~finocchi/`.

[||]Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via di Tor Vergata 110, 00133 Roma, Italy. Email: `italiano@disp.uniroma2.it`, URL: `http://www.info.uniroma2.it/~italiano/`.

opening unexplored research directions that may lead to further theoretical improvements and eventually to more practical algorithms. Theoretical breakthroughs that answer fundamental algorithmic questions and solve long-standing open problems (such as deterministic primality testing [3]) are a crucial step towards the solution of a problem. However, they often lead to algorithms that are far from being amenable to efficient implementations, raising the question of whether more practical solutions exist. We take as an example the case of linear programming, i.e., the problem of optimizing a linear function subject to a collection of linear inequalities. The worst-case exponential running time of the simplex algorithm proposed by Dantzig [25] in 1947, together with the experimental observation that it usually runs in low-order polynomial time on real-world instances, stimulated much theoretical research aimed at discovering if linear programming was polynomially solvable. The first answer was the ellipsoid method of Khachiyan [37], which, however, was impractical. The quest for a practical algorithm finally led to the interior-point approach of Karmarkar [36]. Introducing a competitor of the simplex method, in turn, inspired a great fluorishing of theoretical and experimental work, leading to major improvements in performances for both approaches, that have been also implemented in a widely used library. We believe that Algorithm Engineering should take into account the *whole* process, from the early design stage to the realization of efficient implementations.

A major goal of Algorithm Engineering is to define standard methodologies and realistic computational models for the analysis of algorithms. For instance, there is more and more interest in defining models for the memory hierarchy and for Web algorithmics. Studying data distribution schemes to achieve locality of reference at each level of the memory hierarchy is indeed fundamental in order to minimize I/O accesses and cache misses and to get efficient codes. This topic recently has been the subject of much research: we review some of the most promising advances in Section 2. Issues related to Web algorithmics are also very interesting, as the Internet is nowadays a primary motivation for several problems: security infrastructure, Web caching, Internet searching and information retrieval are just a few of the hot topics. Devising realistic models for the Internet and Web graphs is thus essential for testing the algorithmic solutions proposed in this settings.

Another aspect of Algorithm Engineering, usually referred to as *Experimental Algorithmics*, is related to performing empirical studies for comparing actual relative performance of algorithms so as to study their amenability for use in specific applications. This may lead to the discovery of algorithm separators, i.e., families of problem instances for which the performances of solving algorithms are clearly different, and to identifying and collecting problem instances from the real world. Other important results of empirical investigations include assessing heuristics for hard problems, characterizing the asymptotic behavior of complex algorithms, discovering the speed-up achieved by parallel algorithms and studying the effects of the memory hierarchy and of communication on real machines, thus helping in predicting performance and finding bottlenecks in real systems. The main issues and common pitfalls in Experimental Algorithmics are reviewed in Section 3.

The surge of investigations in Algorithm Engineering is also producing, as a side effect, several tools whose target is to offer a general-purpose workbench for the experimental validation and fine-tuning of algorithms and data structures. In particular, software libraries of efficient algorithms and data structures, collections and generators of test sets, program checkers, and software systems for supporting the implementation and debugging process are relevant examples of such an effort. We will discuss some relevant aspects concerning tools for coding and debugging in Section 4 and the benefits of robust, efficient, and well-documented software libraries in Section 5.

Last, but not least, Algorithm Engineering encourages fruitful cooperation not only between theoreticians and practitioners, but also, and more importantly, between computer scientists and researchers from other fields. Indeed, experiments have played so far a crucial role in many scientific disciplines: in physics, biology, and other natural sciences, for instance, researchers have been extensively running experiments to learn certain aspects of nature and to discover unpredictable features of its internal organization. Approaches and results from different fields may be therefore very useful for algorithm design and optimization.

2 Abstractions and Models in Algorithmics

The mainstream model of computation used in algorithm design is the Random Access Model [4], which assumes a sequential processor and constant access time to memory locations. The main analytical tool is asymptotic analysis of worst-case behavior, w.r.t. either running time or quality of the solution. These abstractions have provided a strong mathematical framework for the analysis of algorithms, that led to major algorithmic advances over the last 30 years. However, predicting the *actual* behavior of an algorithmic code may be a difficult task, since these general models and techniques do not directly fit to existing machines and real problems. In the following we discuss some aspects of these difficulties.

At the edge of Asymptopia. Many authors call "Asymptopia" the range of problem instances for which an algorithm exhibits clear asymptotic behavior. Unfortunately, for certain algorithms, Asymptopia may include only huge problem instances, far beyond the needs of any reasonable practical application. This means that, due to the high constants hidden in the analysis, theoretical bounds may fail to describe the behavior of algorithms on many instances of practical interest. As a typical example, the experimental study in [48] compares implementations of Prim's minimum spanning tree algorithm based on simple binary heaps and on the more sophisticated Fibonacci heaps by Fredman and Tarjan [32]: the latter implementation, although asymptotically better, improves in practice upon the former only for huge dense graphs.

The situation may be even worse: constants hidden in the asymptotic time bounds may be so large as to prevent any practical implementation from running to completion. The Robertson and Seymour cubic-time algorithm [50] for testing if a graph is a minor of another graph provides an extreme example: as a matter of fact, no practical implementation seems able to face the daunting 10^{150} constant factor embedded in the algorithm, and no substantial improvement has been proposed yet.

Beyond worst-case analysis. Algorithms are usually analyzed either in the worst or in the average case. Many algorithms typically behave much better than in the worst case, so considering just worst-case bounds may lead to underestimate their practical utility. On the other side, many algorithms perform unusually well on random inputs in the average-case, but random instances often little resemble instances actually encountered in practice. Since the input distribution is unknown in most applications, a realistic average case analysis is typically unfeasible. Quite recently, an alternative approach, dubbed *smoothed analysis*, has been proposed in [53]: the performance of an algorithm is measured under slight random perturbations of arbitrary inputs. It seems that the smoothed approach can explain the behavior of some algorithms where other analyses fail.

An interesting example in this setting is provided by the simplex algorithm for solving linear programs. In the late 1970's, the simplex algorithm was shown to converge in expected polynomial time on various distributions of random inputs with quite unrealistic characteristics. The asymptotic worst-case time bound was known to be exponential, yet experiments showed that the running time was bounded by a low-degree polynomial also in many real-world instances. This raised the quest for a more precise analysis. It has been actually proved in [53] that the simplex algorithm has polynomial smoothed complexity.

Of a similar flavor is research on *parameterized complexity*, proposed by Rod Downey and Mike Fellows [30]: this is a novel approach to complexity theory which offers a means of analysing algorithms in terms of their tractability and which is much important in dealing with NP-hard problems. The key idea of the theory is to isolate some aspect of the input, the parameter, and to confine the combinatorial explosion of computational difficulty to an additional function of the parameter, with other costs remaining polynomial. Fixed-parameter tractable problems have algorithms running in $O(f(k)n^d)$ time, where n is the instance size, k is the parameter, f is some exponential (or worse) function of k, and d is a constant independent of k. This kind of algorithms are of practical importance for small values of the parameter k and are often at the base of very effective heuristics.

Computational models. Providing simplified computational models which preserve the essential features of the real technology is essential in Algorithm Engineering. To exemplify this idea, we discuss issues related to memory performances. Modern computers are indeed characterized by an increasingly steep memory hierarchy, consisting of several levels of internal memories (registers, caches, main memory) and of external devices (e.g., disks or CD-ROMs). Due to physical principles and economical reasons, faster memories are small and expensive, while slower memories can be larger and inexpensive. Designing algorithms in the classical RAM model, which does not take into account memory latency issues, benefits of cache hits and overheads due to cache misses, is therefore becoming progressively more dangerous. Efficient algorithmic codes are forced to cope with bounded (and not unlimited) memory resources and should attempt at taking advantage of the hierarchy as much as possible. This poses the challenging problem of designing algorithms that maintain locality of reference in data access patterns, i.e., able to cluster memory accesses both in time and in space.

Many extensions of the RAM model have been proposed at this aim. Since I/O accesses represent the major bottleneck when dealing with massive data sets, early works focused on issues related to external memories. Many efficient algorithms and data structures have been designed in the classical *I/O model* [2, 8, 57], which abstracts a two-level hierarchy consisting of a main memory and a disk from/to which data are transferred in blocks of B contiguous items. The model was soon extended to deal with parallel systems (multiple processors, parallel memories) [58, 59] and with multi-level hierarchies [1, 59]. Unfortunately, analyzing algorithms in multi-level models is not easy, due to the many parameters involved in the analysis: among others, block size B and memory size M at each level of the hierarchy must be considered. This encouraged research for simpler alternatives.

The *cache-oblivious model* recently introduced by Frigo et al. [33] appears to be very appealing in this setting, as it allows it to reason about two levels of the hierarchy, but to prove results for an unknown number of levels. The key idea is not using parameters B and M

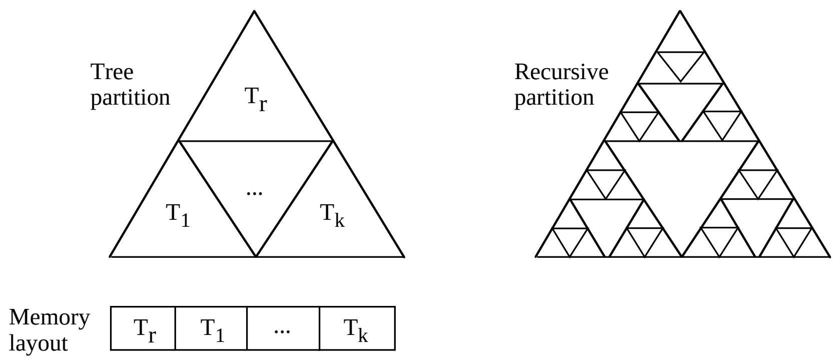

Figure 1: Van Emde-Boas layout.

explicitly: we exemplify it on B-trees, the classical dynamic external-memory dictionary [11]. In a B-tree on n elements, the degree of each internal node is proportional to the block size B: in this way, the depth of the tree is reduced from $O(\log_2 n)$ to $O(\log_B n)$, thus achieving optimality of I/O accesses in search operations. As noticed by Prokop [49], the same effect can be achieved even if B is unknown by using the Van Emde-Boas layout, which distributes the nodes of a balanced binary tree in memory as shown in Figure 1. The tree is partitioned in $\Theta(\sqrt{n})$ subtrees, which can appear in any order, provided their internal layout is obtained by recursively applying the same partitioning scheme. Bender, Demaine and Farach-Colton show how to make the Van Emde-Boas layout dynamic, i.e., how to support insertions and deletions of elements, thus devising the first cache-oblivious version of B-trees [13]. Many other cache-oblivious algorithms and data structures have been designed since then [14, 12, 18]. Such algorithms appear to be portable and flexible. A few works, including [14, 18, 40], study also experimentally the practicality of cache-oblivious data structures w.r.t. hand-tuned cache-aware versions, though we believe that much work still remains to be done on this topic.

3 Experimental Algorithmics

Algorithms must be implemented and tested in order to have a practical impact. Experiments can help measure practical indicators, such as implementation constant factors, real-life bottlenecks, locality of references, cache effects and communication complexity, that may be extremely difficult to predict theoretically. They may also help discover easy and hard instances for a problem. For example, it has been observed that some hard computational problems, such as data clustering, seem to be “hard only when they are not interesting”: when the solution is robust, like in many real-world instances, it is often not too difficult to discover it. A careful tuning of the code, as well as the addition of ad-hoc heuristics and local hacks, may dramatically improve the performances of some algorithms, although the theoretical asymptotic behavior may be not affected. Many clear examples of this fact are addressed in the literature: among them, the implementation issues of the push-relabel algorithm for the maximum flow problem by Goldberg and Tarjan [22] stand out.

Unfortunately, as in any empirical science, it may be sometimes difficult to draw general conclusions about algorithms from experiments. Common pitfalls, often experienced by researchers in their studies, seem to be:

- Dependence of empirical results upon the experimental setup:
  - Architecture of the running machine: memory hierarchy, CPU instructions pipelining, CISC vs RISC architectures, CPU and data bus speed are technical issues that may substantially affect the execution performance.
  - Operating system: CPU scheduling, communication management, I/O buffering and memory management are also important factors.
  - Encoding language: features such as built-in types, data and control flow syntactic structures and language paradigm should be taken into account when choosing the encoding language. Among others, C++, C and Fortran are most commonly used in this context. However, we point out that powerful C++ features such as method invocations, overloading of functions and operators, overriding of virtual functions, dynamic casting and templates may introduce high hidden computation costs in the generated machine code even using professional compilers.
  - Compiler's optimization level: memory alignment, register allocation, instruction scheduling, repeated common subexpression elimination are the most common optimization issues.
  - Measuring of performance indicators: time measuring may be a critical point in many situations including profiling of fast routines. Important factors are the granularity of the time measuring function (typically 1 μsec to 10 $msec$, depending upon the platform), and whether we are measuring the real elapsed time, the time used by the user's process, or the time spent by the operating system to do I/O, communication or memory management.
  - Programming skills of developers: the same algorithm implemented by different programmers may lead to different conclusions on its practical performances. Moreover, even different successive refined implementations coded by the same programmer may greatly differ from each other.
  - Problem instances used in the experiments: the range of parameters defining the test sets used in the experiments and the structure of the problem instances themselves may lead to formulate specific conclusions on the performance of algorithms without ensuring generality. Another typical pitfall in this context consists of testing codes on data sets representing classes that are not broad enough. This may lead to inaccurate performance prediction. An extreme example is given by the Netgen problem instances for the minimum cost flow problem [38] that were used to select the best code for a multicommodity flow application [41]. That code was later proved to behave much slower than several other codes on real-life instances by the same authors of [41]. In general, it has been observed that some algorithms behave quite differently if applied on real-life instances and on randomly generated test sets. Linear programming provides a well known example.
- Difficulty in separating the behavior of algorithms: it is sometimes hard to identify problem instances on which the performance of two codes is clearly distinguishable. In general, good algorithm separators are problem families on which differences grow with the problem size [34].

- Unreproducibility of experimental results: possibly wrong, inaccurate or misleading conclusions presented in experimental studies may be extremely difficult to detect if the results are not exactly and independently reproducible by other researchers.

- Modularity and reusability of the code: modularity and reusability of the code seem to conflict with any size and speed optimization issues. Usually, special implementations are difficult to reuse and to modify because of hidden or implicit interconnections between different parts of the code, often due to sophisticated programming techniques, tricks and hacks which they are based on, but yield to the best performances in practice. In general, using the C++ language seems to be a good choice if the goal is to come up with a modular and reusable code because it allows defining clean, elegant interfaces towards (and between) algorithms and data structures, while C is especially well suited for fast, compact and highly optimized code.

- Limits of the implementations: many implementations have strict requirements on the size of the data they deal with, e.g., work only with small numbers or problem instances up to a certain maximum size. It is important to notice that ignoring size limits may lead to substantially wrong empirical conclusions, especially in the case where the used implementations, for performance reasons, do not explicitly perform accurate data size checking.

- Numerical robustness: implementations of computational geometry algorithms may typically suffer from numerical errors due to the finite-precision arithmetic of real computing machines.

Although it seems there is no sound and generally accepted solution to these issues, some researchers have proposed accurate and comprehensive guidelines on different aspects of the empirical evaluation of algorithms maturated from their own experience in the field (see, for example [7, 34, 35, 47]). The interested reader may find in [43] an annotated bibliography of experimental algorithmics sources addressing methodology, tools and techniques.

3.1 Test Sets

The outcome of an experimental study may strongly depend on instance type and size: using wrong or not general test sets may lead to wrong conclusions. Performing experiments on a good collection of test problems may also help in establishing the correctness of a code: in particular, collecting problem instances on which a code has exhibited buggy behavior may be useful for testing further implementations for the same problem. In the case of optimization algorithms, there usually exist classes of examples for which an algorithm does well or poorly: examining such "pathological" examples may help characterize why the algorithm does poorly and improve its performances on a larger class.

Standardizing common benchmarks for algorithm evaluation is a fundamental task in Experimental Algorithmics. Much effort has been put in collecting, designing and generating good problem instances both for specific problems and for general purpose applications. A first important question is whether to use real or synthetic test-sets. It is often the case that algorithms behave quite differently if applied on real-life instances and on randomly generated test sets: both types of instances should be therefore considered. It may be also useful testing the algorithm on synthetic instances with solution structure known in advance:

this is especially true in the case of hard problems, where exact optima are difficult, or even impossible, to find efficiently. A common technique for generating instances with a given solution structure consists of starting from the desired solution and adding data to hide it [34]. We wish to remark that finding real data may not be easy, as they may be owned by private companies. The Stanford GraphBase, the NIST collection and the DIMACS test sets are successful examples of such an effort.

The Stanford GraphBase [39] is a collection of datasets and computer programs that generate and examine a wide variety of graphs and networks. It consists of small building blocks of code and data and is less than 1.2 megabytes altogether. Data files include, for instance, numerical data representing the input/output structure of US economy, highway distances between North American cities, digitized version of famous paintings, "digested" versions of classic works of literature. Several instance generators included in the package are designed to convert these data files into a large variety of interesting test sets that can be used to explore combinatorial algorithms. Other generators produce graphs with a regular structure or random instances.

The US National Institute of Standards and Technology (NIST) collects heterogeneous types of real data sets. In particular, Matrix Market [17] is a visual repository of test data for use in comparative studies of algorithms, with special emphasis on numerical linear algebra. It features nearly 500 sparse matrices from a variety of application domains, such as air traffic control, astrophysics, biochemistry, circuit physics, computer system simulation, finite elements, demography, economics, fluid flow, nuclear reactor design, oceanography, and petroleum engineering. Each matrix in the collection features a web page which provides statistics on the matrix properties and different visualizations of its structure (e.g., 2-dimensional structure plots or 3-dimensional city plots).

The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) encourages experimentation with algorithms by organizing, since 1993, challenges in which the best codes for specific problems are compared. Previous challenges include, among others, network flow and matching, cliques, coloring and satisfiability, traveling salesman problem, priority queues, dictionaries, and multi-dimensional point sets, parallel algorithms for combinatorial problems [29]. Many test sets and instance generators have been collected for the problems considered so far. All of them conform to specific formats and are available over the Web from the DIMACS website [29]. DIMACS test sets are commonly used as standard benchmarks for comparing the behavior of different algorithms for the same problem.

4 Tools for Coding and Debugging

Software tools such as editors for test sets, program checkers, and systems for development, debugging, visualization and analysis have proved themselves to yield consistent and valuable support in all phases of the algorithm implementation process. In this section we discuss two such efforts: *algorithm visualization systems*, which help developers gain insight about algorithms using a form of interactive graphics, and *program checkers*, which can be used to show the correctness of the implementation of an algorithm and to gain confidence in its output.

4.1 Algorithm Visualization Systems

Many software systems in the algorithmic area have been designed with the goal of providing specialized environments for algorithm visualization. Such environments exploit interactive graphics to enhance the development, presentation, and understanding of computer programs [55]. Thanks to the capability of conveying a large amount of information in a compact form that is easily perceivable by a human observer, visualization systems can help developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes. Some examples of this kind of usage are described in [26].

Systems for algorithm animation have matured significantly since the rise of modern computer graphic interfaces and dozens of algorithm animation systems have been developed in the last two decades [19, 20, 54, 51, 27, 24, 56, 42, 21, 10]. For a comprehensive survey we refer the interested reader to [55, 28] and to the references therein. In the following we limit to discuss the features of algorithm visualization systems that appear to be most appealing for their deployment in algorithm engineering.

- From the viewpoint of the algorithm developer, it is desirable to rely on systems that offer visualizations at a *high level of abstraction*. Namely, one would be more interested in visualizing the behavior of a complex data structure, such as a graph, than in obtaining a particular value of a given pointer.

- *Fast prototyping* of visualizations is another fundamental issue: algorithm designers should be allowed to create visualization from the source code at hand with little effort and without heavy modifications. At this aim, *reusability* of visualization code could be of substantial help in speeding up the time required to produce a running animation.

- One of the most important aspects of algorithm engineering is the development of *libraries*. It is thus quite natural to try to interface visualization tools to algorithmic software libraries: libraries should offer default visualizations of algorithms and data structures that can be refined and customized by developers for specific purposes.

- Software visualization tools should be able to animate *not just "toy programs"*, but significantly complex algorithmic codes, and to test their behavior on large data sets. Unfortunately, even those systems well suited for large information spaces often lack advanced navigation techniques and methods to alleviate the screen bottleneck. Finding a solution to this kind of limitations is nowadays a challenge.

- Advanced debuggers take little advantage of sophisticated graphical displays, even in commercial software development environments. Nevertheless, software visualization tools may be very beneficial in addressing problems such as finding memory leaks, understanding anomalous program behavior, and studying performance. In particular, environments that provide interpreted execution may more easily integrate advanced facilities in support to *debugging and performance monitoring*, and many recent systems attempt at exploring this research direction.

- Debugging *concurrent programs* is more complicated than understanding the behavior of sequential codes: not only concurrent computations may produce vast quantities of data, but also the presence of multiple threads that communicate, compete for resources, and periodically synchronize may result in unexpected interactions and non-deterministic executions. Tools for the visualization of concurrent computations should therefore support a non-invasive approach to visualization, since the execution may be non-deterministic and an invasive visualization code may change the outcome of a computation. Declarative specification appears well suited at this aim.

- There is a general consensus that algorithm visualization systems can strongly benefit from the potentialities offered by the *World Wide Web*. Indeed, the use of the Web for easy communication, education, and distance learning can be naturally considered a valid support for improving the cooperation between students and instructors, and between algorithm engineers.

4.2 Program Checkers

Showing the correctness of an algorithm and of its implementation is fundamental to gain total confidence in its output. While theoretical proofs are the standard approach to the former task, devising methodologies for assuring software reliability still remains a difficult problem, for which general powerful solutions are unlikely to be found.

A typical debugging approach relies on *testing suites*: a variety of input instances for which the correct solution is known is selected, the program is run on the test set and its output is compared to the expected one, possibly discovering anomalies and bugs. A careful selection of the test suite (e.g., degenerate or worst-case instances) may lead to discover interesting facts about the code, but there is no consesus on what a good test set is and how meaningful test sets can be generated. More importantly, testing allows it to prove the correctness of a code only on specific instances.

A different approach, named *program verification*, attempts at solving the debugging problem by formally proving that the program is correct. However, proving mathematical claims about the behavior of even simple programs appears to be very difficult.

An alternative method is offered by *program checking*, which is easier to do than verification, but more rigorous than testing. A program checker is an algorithm for checking the output of a computation: given a program and an instance on which the program is run, the checker certifies whether the output of the program on that instance is correct. For instance, a checker for a code that implements a planarity testing algorithm should exhibit a plane embedding if the input graph is declared to be planar, or a Kuratowski subgraph if the input graph is declared to be non-planar [44].

Designing good checkers may be not easy. Randomization and techniques derived from the theory of error-detecting codes proved themselves to be valuable tools at this aim [16, 60]. We take as an example the case of sorting. The input of the checker is a pair of arrays: the first, say $\bar{x}$, represents the input of the sorting program, the second, say $\bar{y}$, its output. The checker must verify not only that the elements in $\bar{y}$ are in increasing order, but also that they in fact are a permutation of the elements in $\bar{x}$. This latter check can be easily accomplished if each element of $\bar{y}$ has attached a pointer to its original position in $\bar{x}$. Even if $\bar{y}$ cannot be augmented with such pointers, the problem can still be solved by employing a randomized method [16, 61]: a hash function h is chosen at random from a suitably defined set of possibilities, and $\sum_i h(x_i)$ is compared against $\sum_i h(y_i)$. If $\bar{y}$ is a permutation of $\bar{x}$, the two values must be equal; otherwise, it can be proved that they differ with probability at least $\frac{1}{2}$. If the checker accepts only pairs $\bar{x}$-$\bar{y}$ for which t such tests pass, the probability of error will be $\leq (\frac{1}{2})^t$, which can be made arbitrarily small.

To conclude, we wish to remark that checking allows one to test on *all* inputs, and not only on inputs for which the correct output is known. Checkers are usually simpler and faster than the programs they check, and thus, presumably, less likely to have bugs. Moreover, checkers are typically reusable in the following sense: since they depend only on the specification of a computational task, the introduction of a new unreliable algorithm for solving an old task may not require any change to the associated checker.

5 Putting Algorithms to Work: Software Libraries

New algorithmic results often rely on previous ones, and devising them only at theoretical level may lead to a major problem: researchers who would eventually come up with a practical implementation of their results may be required to code several layers of earlier unimplemented complex algorithms and data structures, and this task may be extremely difficult. The need for robust and efficient implementations of algorithms and data structures that can be used by non-experts is one of the main motivations for Algorithm Engineering. Devising fast, well documented, reliable, and tested algorithmic codes is a key aspect in the transfer of theoretical results into the setting of applications, but it is fraught with many of the pitfalls described in Section 3. Without claim of completeness, we survey some examples of such an effort.

LEDA. LEDA, the Library of Efficient Data Types and Algorithms, is a library of efficient data structures and algorithms used in combinatorial computing [45]. It provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. The authors started the LEDA project in 1989 as an attempt to bridge the gap between algorithm research, teaching and implementation. The library is written in C++ and features efficient implementations of most of the algorithms and data types described in classical text books such as [4, 5] and [23]. Additional LEDA Extension Packages have been developed by different researchers, and include dynamic graph algorithms, abstract Voronoi diagrams, D-dimensional geometry, graph iterators, parametric search, SD-trees and PQ-trees. Besides, LEDA includes classes for building graphic user interfaces and for I/O, error handling and memory management. LEDA is currently being developed for commercial purposes. Further information can be found over the Internet at the URL: `http://www.algorithmic-solutions.com/enleda.htm`. The design of the LEDA library is heavily based upon features of the C++ language and the library itself is intended to be a flexible and general purpose tool: for this reason, programs based on it tend to be less efficient than especially tuned implementations. However, LEDA is often used as a practical framework for empirical studies in the field of Experimental Algorithmics.

Stony Brook Algorithm Repository. The Stony Brook Algorithm Repository is a comprehensive collection of algorithm implementations for over seventy of the most fundamental problems in combinatorial algorithms. The repository is accessible via WWW at the URL: `http://www.cs.sunysb.edu/~algorith/`. Problems are classified according to different categories: data structures, numerical problems, combinatorial problems, polynomial-time problems on graphs, hard problems on graphs, computational geometry, set and string problems.

The repository features implementations coded in in different programming languages, including C, C++, Fortran, Lisp, Mathematica and Pascal. Also available are some input data files including airplanes routes and schedules, a list of over 3000 names from several nations, and a subgraph of the Erdos-number author connectivity graph. According to a study on WWW hits to the Stony Brook Algorithm Repository site recorded over a period of ten weeks [52], most popular problems were shortest paths, traveling-salesman, minimum spanning trees as well as triagulations and graph data structures. On the opposite, least popular problems, among others, were determinants, satisfiability and planar graph drawing.

Libraries for specialized domains. Many other libraries for specialized domains have been also implemented. TPIE [9] is a templated C++ library that abstracts the Parallel Disk Model and implements basic I/O primitives such as scanning, sorting, permuting, and I/O efficient data structures such as B-trees, R-trees, K-D-B-trees. CNOP [46] is a package for resource constrained network optimization problems that implements, among others, state of the art algorithms for the constrained shortest path and minimum spanning tree problems. The AGD library [6] offers a broad range of existing algorithms for two-dimensional graph drawing, including drawing planar graphs on the grid with straight lines, orthogonal layout algorithms, hierarchical approaches. The CGAL library [31] supports numerically robust implementations of many computational geometry algorithms, providing primitives for manipulating basic and advanced geometric objects and data structures, including polygons, triangulations, polyhedrons, planar maps, range and segment trees, and kd-trees. It also contains generators for geometric objects. CPLEX is a widely used software package for solving integer, linear and quadratic programming problems. It was originally developed by Bixby [15] and later commercialized. The software includes several versions of the simplex and barriers algorithms and has undergone several major improvements in more than ten years. This fact, coupled with the improvements in computer hardware, resulted in a software which is today more than 6 orders of magnitude faster than in its first version: there are problems that can be solved in seconds today on desktop computers that would have taken more than 2 years to solve using the best linear programming codes and best hardware available in the late 1980s.

6 Concluding Remarks

Algorithm Engineering refers to the process of designing, analyzing, implementing, tuning, debugging, and experimentally evaluating algorithms. It refines and reinforces the traditional theoretical approach with experimental studies, fitting the general models and techniques used by theoreticians to actual existing machines and leading to robust and efficient implementations of algorithms and data structures that can be used by non-experts. Trends in algorithm engineering include the following. First of all, programs should be considered as first class citizens: going from efficient algorithms to programs is not a trivial task and the development of algorithmic libraries may be helpful for this. Furthermore, in may applications seconds matter, and thus experimentation can add new insights to theoretical analyses. Moreover, current computing machines are far away from RAMs: if algorithms are to have a practical utility, issues such as effects of the memory hierarchy (cache, external memory), implications of communication complexity, numerical precision must be considered. Machine architecture, compiler optimization, operating system, programming language are just a few of the technical issues that may substantially affect the execution performance. Algorithmic software libraries, tools for algorithm visualization, program checkers, generators of test sets for experimenting with algorithms may be of great help throughout this process.

Since it may be sometimes difficult to draw general conclusions about algorithms from experimental observations, developing a science of algorithm testing appears to be fundamental. Important questions that have been only partially answered include the following. What is an adequate set of test instances and how does performance depend on instance type and size? How can we provide evidence of the correctness of an implementation, i.e., guarantee that the algorithm implemented is the one intended and that the results it provides are correct? Which are proper performance indicators? In particular, in case of bicriteria problems, how can one consistently explore tradeoffs between running time and quality of the solutions? And, finally, what are the proper questions that should be asked and the proper methodologies for experimental works?

References

[1] A. Aggarwal, A.K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science (FOCS 87)*, pages 204–216, 1987.

[2] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. Manuscript, August 2002.

[4] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[5] R.K. Ahuia, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[6] D. Alberts, C. Gutwenger, P. Mutzel, and S. Näher. AGD-library: a library of algorithms for graph drawing. In *Proc. 1st Int. Workshop on Algorithm Engineering (WAE 97)*, pages 112–123, 1997.

[7] R. Anderson. The role of experiment in the theory of algorithms. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/~dsj/methday.html`.

[8] L. Arge. External memory data structures. In *Proc. European Symposium on Algorithm (ESA 01)*, LNCS 2161, pages 1–29, 2001.

[9] L. Arge, O. Procopiuc, and J.S. Vitter. Implementing I/O efficient data structures using TPIE. In *Proc. 10th Annual European Symposium on Algorithm (ESA 02)*, LNCS 2461, pages 88–100, 2002.

[10] R.S. Baker, M. Boilen, M.T. Goodrich, R. Tamassia, and B. Stibel. Testers and visualizers for teaching data structures. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 31, 1999.

[11] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[12] M. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proc. 29st Int. Colloquium on Automata, Languages and Programming (ICALP 02)*, LNCS 2380, pages 195–207, 2002.

[13] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS 00)*, pages 399–409, 2000.

[14] M. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 02)*, pages 29–38, 2002.

[15] R.E. Bixby. Implementing the simplex method: The initial basis. *ORSA Journal on Computing*, 4:267–284, 1992.

[16] M. Blum and S. Kannan. Designing programs that check their work. In *Proc. 21st Annual ACM Symp. on Theory of Computing (STOC 89)*, pages 86–97, 1989.

[17] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. Dongarra. Matrix Market: a web resource for test matrix collections. In R. Boisvert, editor, *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman and Hall, London, 1997. Matrix Market is available at the URL: `http://math.nist.gov/MatrixMarket/`.

[18] G.S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via bynary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 02)*, pages 39–48, 2002.

[19] M.H. Brown. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988.

[20] M.H. Brown. Zeus: a System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 7-th IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[21] G. Cattaneo, G.F. Italiano, and U. Ferraro-Petrillo. CATAI: Concurrent Algorithms and Data Types Animation over the Internet. *Journal of Visual Languages and Computing*, 13(4):391–419, 2002. System Home Page: `http://isis.dia.unisa.it/catai/`.

[22] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill, 2001.

[24] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2), 2000. System home page: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

[25] G.B. Dantzig. *Linear programming and extensions.* Princeton University Press, Princeton, New Jersey, 1963.

[26] C. Demetrescu, I. Finocchi, G.F. Italiano, and S. Naeher. Visualization in algorithm engineering: Tools and techniques. In *Dagstuhl Seminar on Experimental Algorithmics 00371.* Springer Verlag, 2001. To appear.

[27] C. Demetrescu, I. Finocchi, and G. Liotta. Visualizing Algorithms over the Web with the Publication-driven Approach. In *Proc. of the 4-th Workshop on Algorithm Engineering (WAE'00)*, LNCS 1982, pages 147–158, 2000.

[28] S. Diehl. *Software Visualization.* LNCS 2269. Springer Verlag, 2001.

[29] DIMACS Implementation Challenges web page: `http://dimacs.rutgers.edu/Challenges/`.

[30] R.G. Downey and M.R. Fellows. *Parameterized Complexity.* Monographs in computer science. Springer-Verlag, 1999.

[31] A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, and S. Schnherr. On the design of CGAL, the computational geometry algorithms library. *Software - Practice and Experience*, 30:1167–1202, 2000.

[32] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of ACM*, 34:596–615, 1987.

[33] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Symp. on Foundations of Computer Science (FOCS 99)*, pages 285–297, 1999.

[34] A.V. Goldberg. Selecting problems for algorithm evaluation. In *Proc. 3-rd Workshop on Algorithm Engineering (WAE 99), LNCS 1668*, pages 1–11, 1999.

[35] D. Johnson. A theoretician's guide to the experimental analysis of algorithms. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/~dsj/methday.html`.

[36] N.K. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[37] L.G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademiia Nauk SSSR*, 244:1093–1096, 1979. Translated into English in *Soviet Mathematics Doklady*, 20:191–194.

[38] D. Klingman, A. Napier, and J. Stutz. Netgen: A program for generating large scale capacitated assignment, transportation, and minimum cost network flow problems. *Management Science*, 20:814–821, 1974.

[39] Donald E. Knuth. Stanford GraphBase: A platform for combinatorial algorithms. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 93)*, pages 41–43, 1993.

[40] R.E. Ladner, R. Fortna, and B.H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In R. Fleischer, B. Moret, and E. Meineche-Schmidt, editors, *Experimental Algorithmics*, LNCS 2547, pages 78–92. Springer Verlag, 2002.

[41] T. Leong, P. Shor, and C. Stein. Implementation of a combinatorial multicommodity flow algorithm. In D.S. Johnson and C.C. McGeoch, eds.*, Network Flows and Matching: First DIMACS Implementation Challenge*, pages 387–406, 1993.

[42] A. Malony and D. Reed. Visualizing Parallel Computer System Performance. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 59–90. ACM Press, 1999.

[43] C. McGeoch. A bibliography of algorithm experimentation. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/~dsj/methday.html`.

[44] K. Mehlhorn and S. Naher. From algorithms to working programs: On the use of program checking in LEDA. In *Proc. Int. Conf. on Mathematical Foundations of Computer Science (MFCS 98)*, 1998.

[45] K. Mehlhorn and S. Naher. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambrige University Press, 1999.

[46] K. Mehlhorn and M. Ziegelmann. CNOP: a package for constrained network optimization. In *Proc. 3rd Int. Workshop on Algorithm Engineering and Experiments (ALENEX 01)*, LNCS 2153, pages 17–30, 2001.

[47] B.M.E. Moret. Towards a discipline of experimental algorithmics. In *Proceedings of the 5th DIMACS Challenge Workshop*, 1996. Available over the Internet at the URL: `http://www.cs.amherst.edu/~dsj/methday.html`.

[48] B.M.E. Moret and H.D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. *Computational Support for Discrete Mathematics,* N. Dean and G. Shannon eds.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 15:99–117, 1994.

[49] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.

[50] N. Robertson and P. Seymour. Graph minors: a survey. In J. Anderson, editor, *Surveys in Combinatorics*, pages 153–171. Cambridge University Press, 1985.

[51] G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

[52] S. Skiena. Who is interested in algorithms and why? Lessons from the Stony Brook algorithms repository. In *Proc. 2nd Workshop on Algorithm Engineering (WAE 98)*, pages 204–212, 1998.

[53] D.A. Spielman and S.H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC 01)*, pages 296–305, 2001.

[54] J.T. Stasko. Animating Algorithms with X-TANGO. *SIGACT News*, 23(2):67–71, 1992.

[55] J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience.* MIT Press, Cambridge, MA, 1997.

[56] A. Tal and D. Dobkin. Visualization of Geometric Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):194–204, 1995.

[57] J.S. Vitter. External memory algorithms and data structures. In J. Abello and J.S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, 1999.

[58] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[59] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, 1994.

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

[61] M. Wegman and J. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.