The Book Review Column[1]

by William Gasarch
Department of Computer Science
University of Maryland at College Park
College Park, MD, 20742
email: gasarch@cs.umd.edu

Slight change in information given: in the past I have provided price and ISBN numbers of books. The notion of price is ill-defined: the website www.bestbookbuys.com gives a range of prices from place to puchase books on the web. (amazon.com is usually *not* the cheapest, and its usually not even close.). Providing ISBN numbers is also not needed in this electronic age where such information is available at the touch of a button. If the price is notable– either rather high or rather low, this might be mentioned in the review.

Welcome to the Book Reviews Column. We hope to bring you at least two reviews of books every month. In this column four books are reviewed.

1. **Data Structures and Algorithms in Java (2nd ed)** by Michael T Goodrich and Roberto Tamassia. Review by Hassan Masum. This is a textbook for a sophmore course on data structures and algorithms using Java as its base language. This is not just a standard data structures book translated into Java— this should be viewed as a book on object-oriented data structures.

2. **Selected Papers on Analysis of Algorithms** by Donald E. Knuth. Review by Timothy H. McNicholl. This is a collection of papers by Donald Knuth on the Analysis of Algorithms. The papers take the reader through a large variety of mathematical techniques in a motivated way.

3. **How to Solve It: Modern Heuristics** by Zbigniew Michalewicz and David B Fogel. Review by Hassan Masum. This book presents an overview of several problem solving techniques including search-methods, evolutionary computation, and fuzzy systems.

4. **Proof, Language, and Interaction: Essays in Honour of Robin Milner** Edited by Plotkin, Stirling and Tofte. Review by Riccardo Pucella. This is a collection of essay about the work of Robin Milner. The essays are on semantic foundations, programming logic, programming languages, concurrency, and mobility.

I am looking for reviewers for the following books

If you want a FREE copy of one of these books in exchange for a review, then email me at gasarchcs.umd.edu If you want more information about any of these books, again, feel free to email me. Reviewing a book is a great way to learn a field. I have personally reviewed books and then went on to use what I learned in my research.

1. *Introduction to Distributed Algorithms* by Gerald Tel.

2. *Modern Computer Algebra* by von zur Gathen and Gerhard.

3. *Computational and Geometric Aspects of Modern Algebra* Edited by Atkinson, Gilbert, Howie, Linton, and Robertson. (Proceedings of a workshop)

4. *The Clausal Theory of Types* by Wolfram.

5. *Introduction to Process Algebra* by Fokkink.

6. *Learning Automata: Theory and Applications* by Najim and Poznyak.

7. *Algorithmic Geometry* by Boissonnat and Yvinec.

8. *Algorithms Sequential and Parallel: A unified approach* by Miller and Boxer

9. *Parallel Processing and Parallel Algorithms: Theory and Computation* by Roosta.

The following are DIMACS workshop books which are collections of articles on the topic in the title: Discrete Mathematical Problems with Medical Applications, Discrete Mathematical Chemistry, Randomization Methods in Algorithm Design, Multichannel Optical Networks: Theory and Practice, Networks in Distributed Computing, Advances in Switching Networks, External Memory Algorithms, Mobile Networks and Computing, and Robust Communication Networks: Interconnection and Survivability.

Review of **Data Structures and Algorithms in Java (2nd ed)**[2]
Michael T Goodrich and Roberto Tamassia
Publisher: John Wiley and Sons, 2000
Hardcover, $79.55
ISBN: 0-471-38367-8

Review by:
Hassan Masum, Carleton University, Ottawa, Canada, hmasum@ccs.carleton.ca

1 Overview

Data Structures is a first book on algorithms and data structures, using an object- oriented approach. The target audience for the book is a second-year CS class introducing fundamental data structures and their associated algorithms. This second edition of the book has been corrected and revised, and is a substantial improvement over the first edition. A companion web site contains useful ancillary tools, including an excellent set of slides.

Despite several minor errors and some questionable stylistic choices, I found this version of the book to be well-written. The problem sets are large, interesting, and thought-stimulating. In several places connections are drawn between the algorithm being discussed and important contemporary real-world problems (e.g. search engines, DNA sequence comparison, garbage collection), which students usually appreciate.

2 Book Features and Contents

There is a good selection of problems at the end of each chapter. They are usefully divided into three sections: Reinforcement, Creativity, and Projects. As the names indicate, the first set should be relatively straightforward, while the second set includes many trickier problems. The third set contains problems suitable for longer assignments or group projects.

Chapter-by-chapter contents:

1) Java Programming
2) Object-Oriented Design
3) Analysis Tools
4) Stacks, Queues, and Deques
5) Vectors, Lists, and Sequences
6) Trees (includes traversals)
7) Priority Queues
8) Dictionaries (includes hash tables and skip lists)
9) Search Trees (binary, AVL, multi-way, (2,4), and red-black trees)
10) Sorting, Sets, and Selection
11) Text Processing (pattern matching, tries, compression)
12) Graphs (traversals, directed and weighted graphs, shortest paths and MSTs)
Appendix A: Useful Mathematical Facts
Bibliography, Index

The first two sections are a handy overview of Java and object-oriented practices. While reasonably well-written, there are several places where the overview could be improved, such as in the descriptions of passing by value vs passing by reference and the merits of casting with interfaces. (This material will hopefully be review for most students. From personal teaching and tutoring experience, if they are not already reasonably proficient in programming, they may have serious difficulties in completing assignments involving practical implementations of the more advanced data structures. A good Java book, such as Core Java, would be a valuable complement to the text for many students.)

The remaining sections cover a fairly standard selection of data structures, along with their complementary algorithms. Since the book takes an object-oriented approach, the focus is on defining the properties of each data type and its interface methods, and then explaining the algorithms that can be done with that data type; this is a natural and intuitive pedagogical method. (The book's content and explanations are tied fairly closely to the Java programming language, so those who are not interested in learning or teaching with Java should look elsewhere.)

While not part of the book per se, the companion Web site (linked to from the home pages of both of the authors) contains handy pedagogical tools. Instructor-only resources are password-protected. The most valuable for instructors and students may be the slide sets that the authors have prepared, which cover much of the material in the book in a concise and clear way. There are separate sets for students and teachers, with the latter being more detailed and suitable for classroom use; they are clear and well-designed. Other resources on the site include a hints server, source code and applets, a problem database, and relevant web links.

3 Opinion

The Web site adds quite a bit of value to the book, particularly for instructors. Hopefully it will continue to be maintained, and used by the authors to solicit feedback and errata. Useful additions to the Web site would be an Errata section, and a broader selection in the Papers and Web Resources sections.

I noticed several minor errors in presentation, primarily spelling mistakes and errors in vertical spacing. While not serious, it does not inspire confidence in the reader to see such obvious mistakes

make it through the proofreading process...one naturally wonders what other mistakes may be lurking in the algorithms or code.

The book takes a slightly abstract and general approach, defining abstract data types and then discussing particular specializations of them. While this approach is elegant, it may be problematic as an introductory approach for less theoretically-minded students. In the same vein, the book might benefit from using a more developmental approach to initial algorithm explanation (e.g. explaining a problem and then building up the solution in "stream of consciousness" step-by- step mode). (Of course, neither of these points necessarily apply to the reader who is mathematically sophisticated or who is reviewing previously-learned material, although for such readers other books such as Cormen et al's *Introduction to Algorithms* would be more suitable.)

Data Structures has a lot of good points, not least of which is a successful presentation of traditional data structures in an object-oriented framework; however, there are still some blemishes. If the authors put effort into incorporating feedback and increasing ease of understanding, the third edition could be a classic. Nevertheless, this second edition is quite usable as a classroom text or even as a reference for basic topics.

A review of **Selected Papers on Analysis of Algorithms** by Donald E. Knuth[3]
Reviewed by Timothy H. McNicholl[4]
$22.29 from www.ecampus.com
Trade Paperback, February 2000, 540 pages
ISBN number 1575862123
Cambridge University Press

This book contains an excellent collection of Donald Knuth's papers on the analysis of algorithms. In addition, there are a few papers on number theory. The chronological range of the essays begins with Knuth's first published paper on algorithms in 1963 ('Analysis of Length of Strings in a Merge Sort') and continues until the present day ('Linear Probing and Graphs', 1998). The range of topics, although mostly confined to the analysis of algorithms, is vast. There are papers on game algorithms, algorithms related to common mathematical problems (greatest common divisors, factoring, ...), and algorithms related to computer science problems. There are even a few philosophical and etymological essays. It is this wide range of topics that makes this a hard book to review. To focus on any one aspect of this book would give short-shrift to the others. I therefore decided to focus this review on **why I believe every reader of SIGACT news should buy this book**. I divided my reasons up into categories as follows.

Historical essays (the evolution of ...) The evolution of current terminology and fundamental unifying ideas can be found in these essays. For example, the currently accepted use of Ω, O, Θ is laid down in 'Big Omicron and Big Omega and Big Theta'. Interestingly, the earliest use of Ω turns out to be a paper by Hardy and Littlewood [HL]. Their definition of $\Omega(f(n))$ is not the same as Knuth's, but is similar (it requires the key inequality to hold for infinitely many n instead of all sufficiently large n). Knuth makes the case that his definition is more appropriate for computer science, and that since Hardy and Littlewood's definition is not widely used, there is no danger in his re-definition of Ω conflicting with current usage. It should also be noted that with Knuth's definition we get the identity $\Theta = O \cap \Omega$.

[3]©Tim McNicholl 2001
[4]Department of Mathematics, University of Dallas, Irving, TX 75062, tmcnicho@acad.udallas.edu

The essays 'A terminological proposal' and 'Postscript about NP-hard problems' trace the evolution of the term **NP-hard**. In the first paper, a number of terms for this class are proposed such as 'Herculean', 'arduous', 'formidable'. A poll is taken among SIGACT readers to determine which is most acceptable. The term **NP-hard** is not on this list, but is a write-in. Its consonance with other terminology and descriptive power win out over democratic considerations.

Other essays trace the emergence of the key ideas in algorithm analysis such as average-case complexity, asymptotic values, and generating functions. For example, the author's first paper in the subject, 'Length of Strings in a Merge sort', published in 1963, examines the average length of strings that occur in a kind of merge sort (which doesn't look at all like the merge sort we teach our students today). Many of the early papers examine algorithms that are stated in a very machine-like context. With the evolution of Algol, it becomes possible to state them in a purer fashion so that their mathematical essence shines through. 'Mathematical Analysis of Algorithms', published in 1972, clearly articulates the basic kinds of questions asked such as asymptotic behavior of average case complexity and lower bounds on algorithmic solutions of a certain kind to a given problem. The difficulties with the latter kind of problem are also lucidly described. In fact, this essay would make excellent reading for a beginning student in algorithm analysis. Some of the history of random tree algorithms can be found in 'A trivial algorithm whose analysis isn't'.

Philosophical essays Certainly, the most provocative essay in this book for readers of SIGACT news is 'The Dangers of Computer Science Theory.' The question is posed , 'Has theoretical computer science contributed anything of value to programmers?' The essay then goes on to catalog many hilarious misapplications of theory to practice from a not-so-random number generator to a class of sorting algorithms for which bubble sort is optimal. Of course, this only shows that there have been a significant number of blunders, not that there haven't been any successes. *E.g.* as a programmer I am much better off for knowing that quicksort is usually better than selection sort unless I expect the input lists to be 'nearly sorted'. The preceding essay in the book, 'Mathematical Analysis of Algorithms', also provides a partial answer. Namely, when engaging in the analysis of algorithms we sharpen those mental faculties we will use in everyday programming tasks. What his examples **do** certainly demonstrate is the danger of taking theory as gospel before empirical testing to see how well our models reflect the real world.

Methodological essays Knuth does an amazing job of relating **how** he approaches problems rather than merely recording a highly polished solution which to the reader seems to come out of the blue. An essay that illustrates this is 'An Analysis of Alpha-Beta Pruning.' Here, a very subtle mistake involving conditional probability is intentionally left in one of the derivations, whose conclusion then turns out to be wrong. The discovery of the mistake through empirical testing of the results, and its correction are then described. The paper is thus more instructive than had the final correct version appeared by itself.

It's just plain fun to read. I will briefly discuss three essays I found most delightful for the problems they explored and the range of mathematics they employed.

(1) 'Linear probing and graphs'.

A surprising connection is found between a pure computer science problem and a problem of graph theory. The situation unfolds as follows. Consider a hashing table with m table slots and n keys and where the hashing is done by linear probing. Let d be the **total displacement** of the resulting table. That is, the sum over all keys k of the displacement between k's initial probe

and where it winds up in the table. Let $F_{m,n}(x) = \Sigma x^d$ where the sum ranges over all m^n hash sequences. Thus, the coefficient of x^d in this sum is the number of hash sequences that yield displacement d. Let $F_n(x) = F_{n+1,n}(x)$. Now, let $C_{m,n}$ be the number of connected graphs on n vertices and m edges. It is shown that

$$F_n(1+w) = \sum_{i=0}^{\infty} w^i C_{n+i,n+1}.$$

Unfortunately, this beautiful and surprising result is demonstrated through primarily algebraic means. It would be nice to find a proof that was more combinatorial in nature.

(2) 'The subtractive algorithm for greatest common divisors'.

Algorithms for finding the greatest common divisor are some of the oldest in existence. They demonstrate that our discipline has origins that pre-date the modern computer. In this paper, an amazingly simple algorithm due to the ancient Greeks and older than Euclid's algorithm is examined. The algorithm works as follows. Given a pair of distinct numbers, subtract the smaller from the larger. Replace the larger with the difference. Repeat until the two numbers are the same. This common value is the greatest common divisor of the two original numbers. Let $S(n)$ be the average number of steps performed by this algorithm on pairs of the form (m, n) where $1 \leq m \leq n$. (It is assumed that all such values of m are equally likely.) It is shown that

$$S(n) = 6\pi^{-2}(\log(n))^2 + O(\log(n)(\log\log(n))^2).$$

Average-case results are known for the Euclidean algorithm. Amongst these, the one that is most comparable to the above result is that the average amount of steps performed by Euclid's algorithm on two numbers that are both $\leq n$ is is $O((\log n)^2)$ [Kn2]. Hence the subtraction algorithm is (surprisingly) of the same order of magnitude.

(3) 'The toilet paper problem'.

This somewhat tongue-in-cheek paper examines the following problem. Assume that a bathroom stall has two toilet paper dispensers, either one of which can be accessed at any time. It is likely that at any time, one roll will have more toilet paper than the other. Assume that the probability that a person will choose from the smaller roll is q, and let $p = 1 - q$ be the probability that a person will choose from the larger roll. Assume also that each person uses the same amount of toilet paper. Let n be the length (in number of sheets) of a roll of toilet paper, and let $M_n(q)$ be the average number of sheets left on the larger roll when the smaller roll empties. For fixed q, the asymptotic behavior of $M_n(q)$ is analyzed. From such a whimsical problem, much mathematics arises! The Catalan number make an appearance since they are related to the number of ways to get from one toilet-paper state to another. Let $C(z)$ be the generating function for the Catalan numbers. That is,

$$C(z)\Sigma_{n1}^{\infty}\binom{2n-2}{n-1}\frac{1}{n}z^n.$$

Let $M(z)$ be the generating function for $M_n(q)$. It is shown that

$$M(z)\frac{z}{(1-z)^2}\left(\frac{q - C(pqz)}{q}\right).$$

Asymptotic values for $M_n(p)$ are then derived. If $p \neq q$ then if r is any number greater than $4pq$ we have:

$$M_n(p) = \begin{cases} p/(p-q) + O(r^n) & \text{if } q < p \\ ((q-p)/q)n + p/(q-p) + O(r^n) & \text{if } q > p. \end{cases}$$

If $p = q$, then

$$M_n(p) = 2\sqrt{\frac{n}{\pi}} - \frac{1}{4}\sqrt{\frac{1}{\pi n}} + O(n^{-3/2}).$$

I close this review by proposing a problem based on the toilet paper problem. Instead of assuming that everyone uses the same amount of toilet paper, assume that different people use different amounts of toilet paper in a range $[a, b]$ subject to a continuous probability distribution f (it might be objected that we are exchanging one unreasonable assumption for another since no one uses $\frac{1}{2}$ a sheet or π sheets of paper) and that this probability is independent of whether a person chooses from the little or big roll. Proceed to investigate $M_n(q)$. A solution will certainly fill an important gap.

1. [HL] G. H. Hardy and J. E. Littlewood, Some problems of Diophantine approximation, **Acta Mathematica 37**(1914), 155 - 238.

2. [Kn2] Donald E. Knuth, **Seminumerical Algorithms**, Volume 2 of **The Art of Computer Programming** (Reading, Massachusetts Addison-Wesley, 1969).

Review of **How to Solve It: Modern Heuristics**[5]
Zbigniew Michalewicz and David B Fogel

Publisher: Springer, 2000
Hardcover, $49.95
ISBN: 3-540-66061-5

Review by:
Hassan Masum
Carleton University, Ottawa, Canada
hmasum@ccs.carleton.ca

1 Overview

How to Solve It is a friendly gem of a book which introduces the basic principles of traditional optimization, evolutionary optimization, neural nets, and fuzzy methods. In the spirit of Polya's classic of the same name, the authors emphasize the "how and why" of the problem-solving process, constantly prodding the reader to stop and solve subproblems, or come up with new heuristics, or indeed question whether or not the problem has been posed correctly in the first place.

This book is clear, concise, and fun to read. It is not a handbook of heuristics, but rather an accessible high-level overview of the pros, cons, and applicability of the major categories of heuristic methods, with particular emphasis on optimization methods utilizing evolutionary computation. The presentation is lucid, and the authors do a good job of picking out key properties of algorithms and problem domains. The only prerequisites are basic mathematics and some problem-solving talent.

[5]

2 Evolutionary Computation in a Nutshell

The book spends more time on evolutionary computation than on any of the other methods; the authors (who are both well-known in the field) claim that this is due to the wide applicability of evolutionary methods. It's therefore worthwhile examining this claim, starting with a brief overview of the field.

Evolutionary computation (EC), while tracing its roots back to isolated experiments in the 1960's on adaptive heuristics, rose to prominence as an important subfield in the 1990's. Although quite a few variations exist – such as genetic algorithms, genetic programming, and evolution strategies – they can all be considered as specializations of the general EC theme. A good overview is given in the two-volume set *Evolutionary Computation 1 and 2*, and the best online starting point is the *Hitchhiker's Guide to Evolutionary Computation*.

The essential idea behind EC is to specify an algorithmic description of the evolutionary process, explore the properties of this algorithm via experimentation, and then tweak and solve problems for which the particular algorithm is suited. A key advantage of EC is that some of these simple algorithmic descriptions have been shown to perform well across a wide range of problem domains; they thus provide a "swiss army knife" approach, which may not be the best method for a given problem but is often simple and "good enough".

In its simplest algorithmic formulation, evolution is deemed to contain these methods:
Individual // a computational entity, often specified as a vector or matrix
Population // a bunch of Individuals
Fitness(Individual) // returns the fitness of an Individual, usually as a bounded float
Generate(Population) // generates some new Individuals to add to the Population, through heuristics and / or recombination of previous highly fit Individuals (usually replacing some previous Population members which were less fit)
Vary(Population) // randomly change some Population members (e.g. mutation)
Select(Population) // using the Fitness() method, evaluate each individual in the Population; more fit ones will have a greater chance of staying around during the Generate() method

So, after the initial Population has been formed, there is a repeated Generate-Vary-Select cycle; those Individuals which do well in the Select() method will tend to stay around, both by being retained and by being chosen as templates from which new Individuals are formed.

To take a simple example, consider representing a CNF-SAT problem in this formulation (where one is looking for a satisfying assignment of values to boolean variables for a formula in conjunctive normal form) . The Population would consist of a (usually constant) number of Individuals, each of which could encode a potential solution as a fixed-length binary vector, with a 0 or 1 in any position meaning the corresponding variable is true or false respectively.

The Fitness() method takes a little thought; since it should give information on how "good" each Individual is, we can't just use a binary-valued function that returns 0 if the formula is not satisfied and 1 if it is. One way to get around this would be to discretize the interval from 0 to 1 and assign larger values if more clauses are satisfied. Generate() creates a starting pool of solutions in the first cycle (i.e. a random set of boolean strings, unless we have some clever heuristic to let us guess good solutions), and combines Individuals through e.g. crossover (where each string is "cut" in a random position, and the pieces to one side of the cut swapped). Vary() could flip each position of each Individual with a small random probability, and Select() assigns greater chance of survival and replication to Individuals of higher fitness (through methods with self-descriptive names like fitness-proportional selection, rank-based selection, or tournament selection).

Many more details may be involved in encoding real problems, typically including a lot more domain knowledge; as should be apparent, there is a lot of choice in choosing how to represent problems. It might even be the case that there is already an approximation algorithm out there which yields great solutions in low-order polynomial time, in which case evolutionary computation methods are unnecessary and undesirable; the most suitable problems are large and complex ones where our intuition for generating even approximate solutions breaks down.

Clearly, in order to actually solve a problem, the Fitness() method must encode the problem description and environment; this method is usually the computational bottleneck of the whole procedure for complex problems. It's also essential to choose a good representation for Individuals (such that the distribution of Individuals over the problem space satisfies some desirable properties like uniform density), and to pick Vary() and Generate() methods that are matched to the problem (by using heuristics and problem domain knowledge to combine good subsolutions).

In the Genetic Algorithm (GA) approach, each Individual is usually encoded as a vector of parameters. Genetic Programming (GP) extends this idea to let each Individual represent a program, usually encoded as a parse tree in some LISP-like language; these programs are then executed (and their performance on a problem of interest evaluated) by the Fitness() method. For real-life problems, hybrid algorithms which combine GA or GP with some other domain-specific heuristics are quite common; one simple way to combine the GA with many other methods is to use it as a front end controlling parameter values for a domain-specific algorithm.

Is evolutionary computation just the most recent biologically-inspired method to be hyped beyond its intrinsic worth (e.g. cybernetics, neural nets, fuzzy methods, ...)? After a fair amount of thought on this question, my personal opinion is a qualified no. It's true that specific implementations of the idea have been overhyped, but this is fairly universal in our attention- seeking world; it's perhaps even the case that evolutionary computation as currently instantiated in algorithmic form may not scale beyond problems of some (as yet undetermined) complexity.

However, just based on results that have been achieved to date, evolutionary computation is a demonstrably powerful heuristic method, often matching or exceeding the best known solution through other methods; the series *Genetic Programming I, II, III* by Koza has many examples using GP. And in the long run, my educated guess (bolstered by the apparent success of biological evolution) is that additional layers or methods will be added to the basic "Mendelian genetic" paradigm of EC, that will allow it to scale well beyond levels achieved to date.

As a final note, I would add that the field of EC is particularly ripe for good mathematical and algorithmic analysis. The vast majority of results to date have been empirical, although there has been some analytical work done in places like the *FOGA* conference series, using Markov models and similar analytical tools. Readers of SIGACT could contribute a great deal to understanding the mathematical foundations of these methods, and defining problem classes for which the methods are particularly (un)suitable.

References:

Evolutionary Computation 1 and 2...Edited by Back, Fogel, and Michalewicz; 2000.

FOGA...Foundations of Genetic Algorithms; biennial conference.

Genetic Programming I, II, III...Authored by John Koza; 1992, 1994, and 1999.

Hitchhiker's Guide to Evolutionary Computation...`alife.santafe.edu/joke/encore/www/`

3 Book Features and Contents

The first thing the reader will notice when glancing through the book is the prefacing of each chapter by a problem-solving mini-chapter to loosen up the mind and foreshadow concepts. This

stylistic innovation is highly successful, providing an entertaining and stimulating introduction to various problem-solving methods. As with all good puzzle collections, a plausible route to each solution is explained; problem choice is varied, including several classics (such as the infamous "find the counterfeit coin out of a dozen in three weighings").

As the authors illustrate with a couple of fifth-grade test questions, solving a problem without being given the surrounding context can be surprisingly difficult, and is a skill not emphasized in our educational system. It's therefore important to build up intuition for recognizing and generating contexts, and for knowing when to redefine the problem. One practical context- generating ontological tool is to have nice overall taxonomies of problems, solutions, and knowledge in general; the authors use this in several places by e.g. dividing traditional optimization methods into those utilizing complete versus partial solutions.

The chapter-by-chapter contents of the book are:

1) Why Are Some Problems Difficult to Solve?
2) Basic Concepts
3) Traditional Methods - Part I
4) Traditional Methods - Part II
5) Escaping Local Optima
6) An Evolutionary Approach
7) Designing Evolutionary Algorithms
8) The Traveling Salesman Problem
9) Constraint-Handling Techniques
10) Tuning the Algorithm to the Problem
11) Time-Varying Environments and Noise
12) Neural Networks
13) Fuzzy Systems
14) Hybrid Systems
15) Summary
Appendix A: Probability and Statistics
Appendix B: Problems and Projects
References

As is readily seen, a good chunk of the book is taken up with issues to do with evolutionary optimization. After discussing the background and basic ideas behind evolutionary computation, the reader is given instruction on how to actually use them in practice. Chapter 8, on different representations for solving the TSP, may be useful for those who wish a detailed demonstration of the issues that arise in thinking about potential solution methods.

The extensive bibliography provides many references for further study, and particular references are often singled out at pertinent locations in the main text. The visual and informational layout is easy to read and logical. Two useful appendices review probability and statistics, and suggest problems and projects to try.

4 Opinion

I particularly appreciated the authors' willingness to share their opinions on what does and doesn't work, and what the strengths and weaknesses of each method are; this sort of straight talk is very useful in placing each method in perspective, as opposed to the lamentable tendency in many theory books to let learners generate all their own intuition.

The book is suitable for a senior undergraduate course on heuristic methods. A reader who starts off not knowing anything about heuristic optimization will get a good feel for many methods that are usually taught in isolation, and even the experienced reader may appreciate the relatively unified presentation of ideas. On the downside, the book could use a little beefing up of its content, especially in the sections on traditional methods and hybrid systems; as it stands, the subtitle should really be "An Introduction to Modern Heuristics".

This book is not a comprehensive overview of heuristic algorithms, but rather a selective introduction to key ideas and categories of heuristics. As a friendly big-picture introduction to this important area of modern algorithms, or as a review of the basic ideas behind heuristic methods, *How to Solve It* is worth having on your shelf.

Review of
Proof, Language, and Interaction: Essays in Honour of Robin Milner[6]
Edited by Plotkin, Stirling and Tofte
Publisher: MIT Press
Review by Riccardo Pucella, Dept of C.S., Cornell University
Price: ecampus.com:$64.75, BooksAMillion.com:$66.70, amazon.com:$74.48
Hardcover
ISBN 0262161885

1 Overview of Milner's work

The field of theoretical computer science has had a long tradition of providing wonderful edited books to honor various prominent members of the community. The idea behind such edited works is clear: celebrate the work of a particular researcher by having co-authors and colleagues write papers on subjects related to the researcher's body of work. When successful, such a project provides for a wonderful excursion through the work of a researcher, often highlighting an underlying coherence to that work.

On that account, the book **Proof, Language, and Interaction: Essays in the Honour of Robin Milner** (edited by Plotkin, Stirling and Tofte) is a success. In no small part this is due to Milner's span of work from theory to practice. It is hard to both briefly describe Milner's contributions and give an idea of the breadth of his effort. Two major tracks of research emerge. First, after some work with John McCarthy's AI group at Stanford, he developed LCF (specifically, Edinburgh LCF), a system for computer-assisted theorem proving based on Dana Scott's ideas on continuous partial functions for denotational semantics. Not only working on the implementation, Milner also worried about the semantic foundations [1]. LCF came with a programming language, Edinburgh ML, which evolved with Milner's help into Standard ML, a higher-order language that introduced many features now standard in advanced programming language, features such as polymorphism and type inference [4]. His second track of research involves concurrency. He invented CCS, the calculus of communicating systems [2]. Work on the semantics of CCS went from the traditional domain-theoretic approach to a new operational view of processes equality based on the notion of bisimulation. Subsequent work by Milner led to the development of other calculi, most notably the π-calculus, which included a powerful notion of mobility [3]. His later work in the area attempted to provide a unifying framework for comparing calculi, and unifying sequential and concurrent computation.

[6]

This range of interests is reflected in the table of contents of the book. The book splits across different subjects: *semantic foundations*, where we discover work aimed at understanding computation, both sequential and concurrent, *programming logic*, where we discover work derived from Milner's work on LCF, and aimed at understanding the role of formal mathematics, *programming languages*, where we discover Standard ML, and other languages based on Milner's idea about concurrent calculi, *concurrency*, where we discover work on Milner's early CCS approach and extensions, and *mobility*, where we discover work related to the π-calculus.

2 Contents

I. **SEMANTIC FOUNDATIONS**

1. **Bistructures, Bidomains, and Linear Logic.**
   Curien, Plotkin and Winskel show how bistructures provide a model of classical linear logic extending the web model; they also show that a certain class of bistructures represent bidomains.

2. **Axioms for Definability and Full Completeness.**
   Abramsky presents axioms on models of PCF from which the key results on definability and full abstraction can be derived; these axioms are derived in part from the successful work on game semantics.

3. **Algebraic Derivation of an Operational Semantics.**
   Hoare, Jifeng and Sampaio show how one can derive an operational semantics from a reasonably complete set of algebraic laws for a programming language.

4. **From Banach to Milner: Metric Semantics for Second Order Communication and Concurrency.**
   De Bakker and van Breugel study the operational and denotational semantics for a simple imperative language with second-order communication, where processes can send statements rather than values to each other.

5. **The Tile Model.**
   Gadducci and Montanari present a framework for specifying rule-based computational systems that combines structure-driven inference rules and an incremental format to build new rules from old ones.

II. **PROGRAMMING LOGIC**

6. **From LCF to HOL: A Short History.**
   Gordon gives the history of HOL, showing how HOL emerged from an early extension of LCF that handled a logic for describing so-called "sequential machines", with a concern towards hardware verification.

7. **A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions.**
   Paulson presents a theorem-proving package (for Isabelle) supporting fixedpoints; it can be used to implement inductive definitions (via least fixedpoints) and coinductive definitions (via greatest fixedpoints).

8. **Constructively Formalizing Automata Theory.**
   Constable, Jackson, Naumov and Uribe show how one can prove fundamental theorems of automata theory (the Myhill-Nerode theorem, for instance) in the NuPRL constructive type theory; this is a first step towards the formalization of computational mathematics.

9. **Constructive Category Theory.**
   Huet and Saïbi present an implementation of category theory in the Calculus of Inductive Constructions, showing that type theory is adequate to faithfully represent categorical reasoning.

10. **Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations.**
    Colette and Jones give a methodology for using rely/guarantee specifications in a tractable way; such specifications are an extension of pre/post conditions for sequential operations to concurrent operations, which have to consider the additional issue of interference between operations arising in the concurrent setting.

11. **Model Checking Algorithms for the μ-Calculus.**
    Berezin, Clarke, Jha and Marrero give an overview of the propositional μ-calculus and general algorithms for evaluating μ-calculus formulas; the μ-calculus expresses properties of transition systems by using least and greatest fixpoint operators, can encode many temporal and program logics, and supports efficient model checking algorithms.

III. PROGRAMMING LANGUAGES

12. **A Type-Theoretic Interpretation of Standard ML.**
    Harper and Stone present the first interpretation of the full Standard ML language in type theory, by translating Standard ML into an explicitly typed λ-calculus; this allows the dynamic semantics to be defined on typed, rather than type-erased, programs.

13. **Unification and Polymorphism in Region Inference.**
    Tofte and Birkedal present rules to perform region inference in a language with region-based memory management, using unification to allow for some amount of region-polymorphic recursion.

14. **The Foundations of Esterel.**
    Berry gives an overview of Esterel, an imperative synchronous language, describing its underlying reactive model, its semantics and implementation in terms of sequential circuits, and issues related to verification.

15. **Pict: A Programming Language Based on the Pi-Calculus.**
    Pierce and Turner describe Pict, a high-level programming language based on the π-calculus, in a way similar to the way ML is based on the λ-calculus; communication is the sole mechanism for computation.

IV. CONCURRENCY

16. **On the Star Height of Unary Regular Behaviours.**
    Hirshfeld and Moller prove a conjecture of Milner, that the star height hierarchy of regular expressions over a unary alphabet is a genuine hierarchy when equivalence is taken to be bisimilarity; the corresponding question for regular languages is negative.

17. **Combining the Typed λ-Calculus with CCS.**
Ferreira, Hennessy and Jeffrey present a concurrent language based on a unification of CCS and the typed call-by-value λ-calculus, with the aim of defining a communicate-by-value concurrent language; the motivation is to analyze the behavior of CCS when the data communicated is taken from a non-trivial data space.

18. **Discrete Time Process Algebra with Silent Step.**
Baeten, Bergstra and Reniers discuss an extension of ACP (a process algebra similar to CCS) with discrete time and silent step τ, of the kind found in CCS; three views of discrete time process algebra with silent step are presented.

19. **A Complete Axiom System for Finite-State Probabilistic Processes.**
Stark and Smolka give a sound and complete equational axiomatization of probabilistic bisimilarity for a probabilistic version of CCS, with binary summation of the form $E_p{+}E'$ (meaning E is chosen with probability p, E' with probability $1-p$).

V. **MOBILITY**

20. **A Calculus of Communicating Systems with Label Passing —- Ten Years After.**
Engberg and Nielsen describe a historical step in the development of the π-calculus from CCS by presenting ECCS, an extension of CCS that introduced a notion of channel passing.

21. **Trios in Concert.**
Parrow exhibits a fragment of the π-calculus, in the form of a subset of its terms, such that any π-calculus term is weakly equivalent to a term in the fragment; this highlights the root of the power of π-calculus.

22. **Concurrent Objects as Mobile Processes.**
Liu and Walker present an extension of the π-calculus suitable for the semantic definition of concurrent object-oriented languages; the calculus includes process abstraction as in the higher-order π-calculus and data other than channel names, but excludes higher-order interaction.

23. **λ-Calculus, Multiplicities, and the π-calculus.**
Boudol and Laneve address the question of the semantics induced by λ-terms when they are encoded in the π-calculus; they show why the π-calculus is more discriminating than the λ-calculus, in the sense that two terms which are equivalent in the λ-calculus may be distinguished in an appropriate π-calculus context.

24. **Lazy Functions and Mobile Processes.**
Sangiorgi examines the encoding of the lazy λ-calculus into the π-calculus; the encoding is used to derive a λ-model from the π-calculus processes; full abstraction of the model is obtained by strengthening the operational equivalence on λ-terms.

3 Opinion

This volume is by no means an introductory book. Most of the articles are at the level of journal papers, and deal with technical issues. Those articles which are more expository or that provide historical perspective require a knowledge of the subject to be fully appreciated. Nevertheless, this volume should form an important part of any collection of work on semantics, programming

languages or concurrency. The articles cover core subjects in those areas, and many present state-of-the-art techniques; for instance, the current trend in game semantics (cf. Abramsky's article), or the use of a type-theoretic internal language for programming language specification (cf. Harper and Stone's article). If anything, abstracts for the various articles would have been very helpful.

References

[1] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[2] R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[3] R. Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, 1999.

[4] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Mass., 1997.