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

In this column we review 10 books on ALGORITHMS.

1. **How to Think about Algorihtms** by Jeff Edmonds. Review by Kryiakos N. Sgarbas. This algorithmic textbook uses a meta-algorithmic approach to reveal that all algorithms of the same category work more or less the same way. The emphasis is more on how they work and how to think about them than on formal details of analysis.

2. **A Programmer's Companion to Algorithm Analysis** by Ernst Leiss. Review by Dean Kelley. This is *not* a textbook on algorithms. Rather, it focuses on the process of converting an algorithm to efficient, correctly behaving and stable code. In doing so, it necessarily examines the considerable gulf between the abstract view of things when an algorithm is being designed and analyzed and the hard reality of the environment in which it's implementation executes.

3. Joint review of **Algorithms** by Johnonbaugh and Schaefer and **Algorithms** by Dasgupta, Papadimitriou, and U. Vazirani. Joint review by Dean Kelley. This is a joint review of two Algorithms textbooks. In many cases this course follows a mid-level data structures course which covers up through trees. There have been several well written new books in recent years aimed at this course (including the two reviewed here). Both of the reviewed books have at their core material which has become standard for this course. Despite the commonality, the books are quite different from each other and from Edmonds book.

4. **Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms.** by Juraj Hromkovic. Review by Marious Mavronicolas. Randomized algorithms has matured to the point where they *have* paradigms that can be written about in a textbook on the subject. This is that book!

5. **Theoretical Aspects of Local Search** by Michiels, Aarts, and Korst. Review by Jakub Mareček. Yes, there really are Theoretical aspects to local search, a field in which one usually hears about heuristics. This book will tell you all about them and help close the gap between theory and practice.

6. **The Traveling Salesman Problem: A Computational Study** by Applegate, Bixby, Chvátal, and Cook. Review by W. Springer. This book describes methods to really solve large TSP problems. This will give the reader a tour of much math and computer science of interest.

7. **Visibility Algorithms in the Plane** by Ghosh. Review by Alice Dean. One of the most famous theorems in computational geometry is the *Art Gallery Theorem*, and this theorem also serves as an example of the focus of the book under review. Posed by Victor Klee in 1973,

it asks how many stationary guards are required to see all points in an art gallery represented by a simple, n-sided polygon. The answer, given first by Chvátal [3] and later by Fisk [5], using an elegant graph-theoretic proof, is that $\lfloor n/3 \rfloor$ guards are always sufficient and may be necessary. Questions such as this one, of visibility within a polygon, are the subject of this book.

8. **A Course on the Web Graph** by Anthony Bonato. Review by Elisa Schaeffer. Imagine the following graph: the vertices are web pages and two vertices u and v are connected by a (directed) edge (u, v) if there is a hyperlink from u to v. The computational challenge is due to the amount of vertices. This book describes modeling and analyzing the Web graph and summarizes many of the most important results published in the past decade of Web graph studies.

9. **Higher Arithmetic** by Edwards. Review by Brittany Terese Fasy and David L. Millman. This is a text in Number Theory with an algorithmic approach to the topic.

We are looking for reviewers of the following books

Books I want Reviewed

If you want a FREE copy of one of these books in exchange for a review, then email me at gasarchcs.umd.edu

Reviews need to be in LaTeX, LaTeX2e, or Plaintext.

Books on Algorithms and Data Structures

1. *Algorithms and Data Structures: The Basic Toolbox* by Mehlorn and Sanders.
2. *The Algorithms Design Manual* by Skiena.
3. *Algorithms on Strings* by Crochemore, Hancart, and Lecroq.
4. *Algorithms for Statistical Signal Processing* by Proakis, Rader, Ling, Nikias, Moonen, Proudler.
5. *Nonlinear Integer Programming* by Li and Sun.
6. *Binary Quadratic Forms: An Algorithmic Approach* by Buchmann and Vollmer.
7. *Time Dependent Scheduling* by Gawiejnowicz.
8. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching* by Adjeroh, Bell, Mukherjee.
9. *Parallel Algorithms* by Casanova, Legrand, and Robert.
10. *Mathematics for the Analysis of Algorithms* by Greene and Knuth.

Books on Cryptography, Coding Theory

1. *Introduction to Modern Cryptography* by Katz and Lindell.
2. *Concurrent Zero-Knowledge* by Alon Rosen.
3. *Introduction to cryptography: Principles and Applications* by Delfs and Knebl.

4. *Primality Testing and Integer Factorization in Public-Key Cryptography* by Yan

5. *Secure Key Establishment* by Choo.

6. *Codes: An Introduction to Information Communication and Cryptography*

7. *Algebraic Function Fields and Codes* by Stichtenoth.

8. *Coding for Data and Computer Communications* by David Salomon.

9. *Block Error-Correcting Codes: A Computational Primer* by Xambo-Descamps.

Books on Theory of Computation

1. *A Conscise Introduction to Languages and Machines* by Parkes

2. *The Calculus of Computation: Decision Procedures with Applications to Verification* by Bradley and Manna.

3. *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine* by Perzold.

4. *Computability of the Julia Sets* by Braverman and Yampolsky.

Combinatorics

1. C̀ombinatorics and Graph Theory by Harris, Hirst, and Mossinghoff.

2. *Analytic Combinatorics* by Flajolet and Sedgewick.

3. *Combinatorics the Rota Way* by Kung, Rota, and Yan.

4. *A Course in Enumeration* by Aigner.

5. *Random Graphs* by Bollobas.

Misc Books

1. *Difference Equations: From Rabbits to Chaos* by Cull, Flahive, and Robson.

2. *Mathematical Tools for Data Mining* by Simovici and Djeraba.

3. *The Modern Algebra of Information Retrieval* by Dominich.

4. *A Concise introduction to Data Compression* by Salomon.

5. *Proofs and Other Dilemmas: Mathematics and Philosophy* Edited by Gold and Simons.

Review of[2]
How to Think about Algorithms
by Jeff Edmonds
Cambridge University Press, 2008
xiv+450 pages, ISBN: 9780521849319 (Hardback, £55.00, $99.00),
9780521614108 (Paperback, £19.99, $36.99)

Review by
Kyriakos N. Sgarbas (sgarbas@upatras.gr)
Electrical & Computer Engineering Department, University of Patras, Greece

1 Overview

This is a book on algorithms. Or rather a book on applied abstract thinking about algorithms. The algorithms are grouped into three broad categories (iterative, recursive, and algorithms for optimization problems) and each category is presented in a separate part of the book. The book does not focus into the formal aspects of each algorithm. Instead, it aims to explain how one thinks in order to devise such an algorithm, why the algorithm works, and how it relates to other algorithms of the same category. It uses a meta-algorithmic approach to reveal that all algorithms of the same category work more or less the same way. Of course, pseudocode, asymptotic notations, proofs of correctness, are all there, but they are used to explain rather than formally define each algorithm. The language used is simple and approachable and the author prefers to use words to explain how things work instead of overloading the text with mathematical formulas. The layout of the book is quite original (for an algorithm book); in some places it even contains pictures (obviously of the author's family) remotely relevant to some notion mentioned in the page, but giving a nice artistic touch nonetheless.

2 Summary of Contents

Just after the Table of Contents the book starts with a 2.5-page Preface where its goals are explained and a 2-page Introduction with some non-formal definitions concerning computational problems, algorithms, abstract data types, correctness, running time and meta-algorithms. After that it is organized in 5 parts and 28 chapters (each with its own set of exercises), as follows:

Part I "Iterative Algorithms and Loop Invariants" discusses problems that can be solved using iterative algorithms and analyzes such algorithms using loop invariants (i.e. assertions that must hold true every time the computation returns to the top of the loop). Part I consists of Chapters 1 to 7:

Chapter 1 *"Iterative Algorithms: Measures of Progress and Loop Invariants"* (24 pages) on a first reading it seems like a set of advices on how to write algorithms that work. But seen as a set these advices propose a different way of abstract thinking, shifting from the one-instruction-per-step model to the one-instance-of-the-solution-per-step model, by defining appropriate loop invariants

[2]

and measures of progress and applying them promptly. Examples used: insertion/selection/bubble sort and binary search.

Chapter 2 *"Examples Using More-of-the-Input Loop Invariants"* (14 pages) presents some more complex problems solved using more-of-the-input loop invariants (i.e. loop invariants where the measure of progress is related to the amount of the input considered) and a comparison with problems more easily solved with more-of-the-output invariants (i.e. loop invariants where the measure of progress is related to the amount of the output constructed). Examples used: Plane coloring, parsing with DFA, arithmetic operations, Euler cycle, etc.

Chapter 3 *"Abstract Data Types"* (17 pages) briefly introduces lists, stacks, queues, priority queues, sets, graphs, and trees, their implementation and basic operations on them.

Chapter 4 *"Narrowing the Search Space: Binary Search"* (11 pages) as its title suggests, presents loop invariants using the principle of narrowing the search space of a problem after ensuring that the target element always remains in the reduced search space. It uses two very nice examples: Magic Sevens (a card trick) and VLSI Chip Testing (not really a hardware problem, just a problem of element comparison). These examples are intentionally flawed in their definition, so the analysis does not stop to their solution but extends to some variants as well.

Chapter 5 *"Iterative Sorting Algorithms"* (8 pages) discusses bucket sort, counting sort, radix sort and radix counting sort and explains how they relate to each other.

Chapter 6 *"Euclid's GCD Algorithm"* (6 pages) is a short chapter on the well-known algorithm presented separately obviously due to its 'strange' loop invariant. Two other problems using similar-type invariants are mentioned in the chapter exercises.

Chapter 7 *"The Loop Invariant for Lower Bounds"* (10 pages) discusses how to prove the lower bound of an iterative algorithm. The author points out that the process of calculation of a lower bound can be seen as an algorithm itself, therefore the same model based on loop invariants can be used to evaluate it. Examples used: sorting, binary search, parity, and multiplexer.

Part II "Recursion" discusses problems that can be solved using recursive algorithms and explains some methods for devising them. Part II consists of Chapters 8-12:

Chapter 8 *"Abstractions, Techniques, and Theory"* (17 pages) provides the basic thinking tools for confronting problems by recursion. Stack frames, strong induction and a framework for a general-purpose blueprint for recursive algorithms are discussed. The problem of the Towers of Hanoi is used as example and analyzed thoroughly.

Chapter 9 *"Some Simple Examples of Recursive Algorithms"* (16 pages) provides some more examples of recursive algorithms explained with the model of Chapter 8. Examples used: merge sort, quick sort, calculation of integer powers, matrix multiplication, etc.

Chapter 10 *"Recursion on Trees"* (23 pages) presents and analyzes recursive algorithms for operations on trees: counting the number of nodes in binary trees, perform tree traversals, return the maximum of data fields of tree-nodes, calculate the height of a tree, count the number of its leaves, making a copy of the whole tree. A thorough analysis of the heap sort algorithm continues and the chapter concludes with some algorithms for tree-representations of algebraic expressions: expression evaluation, symbolic differentiation, expression simplification.

Chapter 11 *"Recursive Images"* (6 pages) is a small chapter explaining how to use recursive algorithms to produce fractal-like images and random mazes.

Chapter 12 *"Parsing with Context-Free Grammars"* (9 pages): Context-Free Grammars (CFGs) are used in compiler development and computational linguistics to express the syntactic rules of an artificial or natural language. This chapter discusses how to develop algorithms for parsing

expressions (strings) according to a set of CFG-rules and produce structured representations of the input expressions.

Part III "Optimization Problems" presents methodologies for building algorithms (iterative and recursive) to confront several classes of optimization problems. Part III consists of Chapters 13-21:

Chapter 13 *"Definition of Optimization Problems"* (2 pages) presents the definition of what an optimization problem is and gives short examples (only the definition) of different degrees of complexity: longest common subsequence, course scheduling and airplane construction.

Chapter 14 *"Graph Search Algorithms"* (25 pages) presents and analyzes several algorithms on graphs: a generic search algorithm, a breadth-first search algorithm for shortest paths, Dijkstra's shortest-weighted-path algorithm, iterative and recursive depth-first search algorithms, and topological sorting.

Chapter 15 *"Network Flows and Linear Programming"* (27 pages) starts with the definition of the Network Flow and Min Cut optimization problems and presents gradual solutions introducing several variations of hill-climbing algorithms (simple, primal-dual, steepest-ascent). The chapter concludes with a discussion on linear programming.

Chapter 16 *"Greedy Algorithms"* (26 pages) uses the Making Change problem (i.e. finding the minimum number of coins that sum to a certain amount) to explain the basics of greedy algorithms and then applies them in some more examples on job/event scheduling, interval covering, and producing the minimum-spanning-tree of a graph.

Chapter 17 *"Recursive Backtracking"* (16 pages) explores the idea of finding an optimal solution for one instance of the problem using a recurrence relation that combines optimal solutions for some smaller instances of the same problem that are computed recursively. Examples used: searching a maze, searching a classification tree, n-Queens problem, SAT problem (Davis-Putnam algorithm).

Chapter 18 *"Dynamic Programming Algorithms"* (28 pages): Dynamic Programming resembles Recursive Backtracking except that the optimal solutions for the smaller instances of the problem are not computed recursively but iteratively and are stored in a table. The chapter uses the problem of finding the shortest weighted path within a directed leveled graph to explain the basics on developing a dynamic programming algorithm and then elaborates in several optimization aspects.

Chapter 19 *"Examples of Dynamic Programs"* (29 pages) provides some additional examples of problems solved using dynamic programming algorithms, i.e. the longest-common-subsequence problem (with several variations), the weighted job/event scheduling problem, matrix multiplication, CFG-parsing, etc.

Chapter 20 *"Reductions and NP-Completeness"* (22 pages) begins with the definition of problem reduction and then uses it to classify problems according to this relation. It discusses NP-completeness and satisfiability and explains with great detail how to prove that a problem is NP-complete. Examples used: 3-coloring, bipartite matching.

Chapter 21 *"Randomized Algorithms"* (8 pages) points out that sometimes it is a good idea to use randomness (e.g. to avoid a worst case) and presents two models for analysis of randomized algorithms: Las Vegas (guaranteed to give the correct answer but the running time is random), and Monte Carlo (guaranteed to stop within a certain time limit but may not give the right answer). These models, along with the classic (deterministic) worst case analysis are used in some comparative examples.

Part IV "Appendix": These are not simple appendices, they are normal chapters just like all

the previous ones. Some of them have exercises too! Part IV consists of Chapters 22-28:

Chapter 22 *"Existential and Universal Quantifiers"* (9 pages) provides the basic definitions for relations, quantifiers, free and bound variables and discusses some proof strategies involving universal and existential quantifiers.

Chapter 23 *"Time Complexity"* (8 pages) provides definitions of time and space complexity of algorithms and examples on their estimation.

Chapter 24 *"Logarithms and Exponentials"* (3 pages) summarizes the definitions, the properties and some frequent uses of logarithms and exponentials.

Chapter 25 *"Asymptotic Growth"* (11 pages): Formal definitions, estimations and examples of Big Oh, Omega, Theta, etc.

Chapter 26 *"Adding-Made-Easy Approximations"* (10 pages) contains formulas and methods to calculate or approximate sums of series.

Chapter 27 *"Recurrence Relations"* (10 pages) contains formulas and methods for solving recurrence relations.

Chapter 28 *"A Formal Proof of Correctness"* (2 pages) sketches the required steps in order to formally proof the correctness of an algorithm. It does not include any examples but one can compare the process with some of the formal proofs of correctness provided in previous chapters.

Part V "Exercise Solutions" (25 pages) contains solutions of many of the exercises.

The book concludes with a few lines Conclusion and a 10-page Index. It does not have any list of references.

3 Opinion

Since page one, the reader can realize that this is not an ordinary book about algorithms. Its layout, its structure, and its language, all indicate that it is something uncommon, probably unique. Reading it is an experience much different that what one gets from reading a common algorithm textbook. Instead of presenting and analyzing algorithms formally down to every single mathematical detail, instead of stressing the left part of your brain with lengthy proofs and dive deep into algorithm science, this book resembles more than a practical guide on algorithm craftsmanship.

Approachable and informal, although it uses pseudocode to express algorithms and meta-algorithms, it does not formally define the syntax of the pseudo language used.

Its structure is uncommon as well. There are chapters with 2 pages each and chapters with more than 20 pages each. Most of them have exercises, usually in the last section of the chapter, but some have exercises also inside the sections (in appendices too).

Furthermore you will see mathematical formulas with no explanation of what the variables stand for (e.g. in the beginning of Chapter 7). You either already know from previous experience, or you may try to deduce from the context, or -finally- you might think to skip into an appropriate appendix (sometimes the text prompts you to do so, sometimes not).

Actually, the text has many self references to other chapters but not any external references. Regardless whether it presents some well-known algorithm (like Dijkstra's algorithm), or provides additional information (e.g. in Chapter 20 where it mentions that the relation of any optimization problem to CIR-SAT was proved by Steve Cook in 1971), or discussing the current state of the art (like in Chapter 7 on proving lower bounds) there is not a single external reference, neither in any footnote nor in any list of collected bibliography at the end of the book. Strange thing indeed.

The careful reader will also observe a few typos, but they are not serious enough to hinder the reading or alter the semantics of the text.

Reading the text sometimes feels like reading the slides of a class lecture, along with the actual words of what the teacher said, with all the redundant information, the not-so-relevant comments and the jokes the speaker usually adds to make his speech more interesting (e.g. you'll be surprised on the ingredients of hot dogs when you read the linear programming section). Sometimes it uses metaphors to explain something, and keeps the metaphor along several chapters, thus resulting into a unique pictorial terminology framework (e.g. friends, little birds, and magic denote algorithmic components).

Of course, whether you like reading about algorithms under these terms or not, I believe is a matter of personal taste. Personally I loved reading this book, but I suppose one might just as easily hate it for exactly the same reasons. But surprisingly enough, this model works well when explaining difficult subjects. It tends to communicate effectively the whole picture without much details, while the examples fill in the gaps. For instance, Chapter 20 on Reductions and NP-completeness is probably the most well-explained introduction text I have read on the subject.

In conclusion, I believe this book could be considered a must-read for every teacher of algorithms. Even if he reads things he already knows, he will be able to view them from different angles and in the process get some very useful ideas on how to explain algorithms in class. The book would also be invaluable to researchers who wish to gain a deeper understanding on how algorithms work, and to undergraduate students who wish to develop their algorithmic thought. However, I would not recommend someone to try to learn algorithms starting from this book alone. I believe one has to know a great deal on the subject already, in order to fully appreciate the book and benefit from it. For this is not really an algorithm textbook; it's more like the right-brain counterpart of an ordinary algorithm textbook. And as such, it has the potential to be considered a classic.

Review of[3]
A Programmer's Companion to Algorithm Analysis
by Ernst Leiss
Chapman & Hall/CRC, 2007
55 pages, Softcover

Review by
Dean Kelley (dean.kelley@mnsu.edu)
Minnesota State University, Mankato

1 Introduction

In a perfect world software would... well, *work*. It would be stable and produce correct results in a timely manner. Ernst Leiss has written an interesting book which addresses some of the issues which inhibit the development of perfect-world software.

This book is directed toward programmers and software developers and is *not* a textbook on algorithms. Rather, it focuses on the process of converting an algorithm to efficient, correctly behaving and stable code. In doing so, it necessarily examines the considerable gulf between the

[3]

abstract view of things when an algorithm is being designed and analyzed and the hard reality of the environment in which its implementation executes.

2 What's in There – Summary of the Book

The book consists of two main parts plus substantial appendix material. Part 1 describes "the idealized universe that algorithm designers inhabit" with an eye toward ultimately identifying areas where the translation of an algorithm into a program can yield less than satisfactory (or even anticipated) results. Part 2 outlines how the "idealized universe" can be adapted to the real world of the programmer.

Part 1 begins with two chapters that describe various aspects of algorithmic complexity and the assumptions that underlie them. A third chapter gives flesh to these concepts with examples and careful, worked-out analysis.

Part 2 begins by exploring a collection of ways in which a program's reality can fail to match the expectations that algorithm analysis might have led to. The next four chapters look at specific real-world factors that impact what happens when an algorithm becomes code, and what can be done to reduce their impact. Another chapter focuses on constant factors with attention to crossover points. The last chapter in this part of the book examines the impacts of undecidability, infeasibility and NP-completeness.

The four appendices are mostly in support of the material in Part 2 with two of them directly illuminating and expanding the material on undecidability and NP-completeness in the last chapter of that part of the book.

2.1 Chapter 1: A Taxonomy of Algorithmic Complexity

This chapter presents the traditional measures of complexity: space and time complexity, average-case, best-case, worst-case complexity, parallel complexity in various flavors and bit versus word complexity.

Additionally, and in anticipation of material in Part 2, I/O complexity is introduced. This somewhat nontraditional complexity measure essentially measures the amount of data transferred between types of memory. A consequence of the need to move data between memory types is that memory access in an algorithm does not necessarily have uniform cost.

The intent in this chapter is to present a conceptual framework for evaluating the performance of algorithms, emphasizing that there are many dimensions to complexity and that all reflect performance, though from different points of view.

2.2 Chapter 2: Fundamental Assumptions Underlying Algorithmic Complexity

In designing and analyzing algorithms we often (usually) assume that things are inherently friendly. For example, in most cases all arithmetic operations require times which are within a constant multiple of each other and so it's convenient (and safe) to ignore that constant. This is an assumption that generally remains valid when software is designed based on an algorithm which has been analyzed with that assumption.

In frequent analyses, assignment is assumed to have the same kind of uniformity as arithmetic operations – that all assignments cost essentially the same. Similarly with retrieval. We are

effectively assuming that we have enough memory for the data. This assumption doesn't always remain valid when the algorithm is transitioned to code. The resulting impact on performance can be detrimental.

Much of Chapter 2 emphasizes those assumptions which are made in analyzing algorithmic complexity and which are ultimately unfriendly to a programmer implementing the algorithm. This is what the author refers to as "the idealized universe that algorithm designers inhabit" and the differences between it and the world inhabited by software designers are explored in Part 2 of the book.

2.3 Chapter 3: Examples of Complexity Analysis

This chapter applies techniques of complexity analysis to some standard algorithms. The book is intended for software developers who presumably have *some* exposure to design and analysis, but may not necessarily be comfortable enough to be able to see the implications of the details of analysis. In view of that, Chapter 3 provides considerable hands-on exposure to the analysis of a fairly large collection of standard algorithms, some of which will be used in the exposition(s) of Part 2.

Algorithms analyzed include Strassen's algorithm, optimizing a sequence of matrix multiplications, comparison-based sorting algorithms and radixsort, binary search and quickselect, algorithms for binary search tree operations, hashing, graph representation and graph searching algorithms, and Dijkstra's algorithm.

In the analysis of these algorithms, the author directs the reader's attention toward all of the complexity measures in Chapter 1's taxonomy. Because of the quantity of material covered, the chapter is long (the longest in the book). It may be skipped or lightly read by readers who are not uncomfortable or inexperienced with algorithm complexity analysis. In particular, if the book is being used as a companion book for an algorithms course, most of this material is likely presented more pedagogically elsewhere in the course.

2.4 Chapter 4: Sources of Disappointments

Chapter 4 begins Part 2 of the book which focuses on how to deal with lower-level details that can adversely affect both the performance and correctness of an algorithm's translation into code. In Chapter 4 the author discusses general categories of "disappointments" that can occur leaving treatment of the underlying causes to subsequent chapters.

Beginning with a correct algorithm, one reasonably expects that the code derived from a faithful translation of it should produce correct results. Unfortunately, this may not be the case. For example, number representation can give rise to incorrect numerical results as well as error situations which must be properly handled to insure correct results. Things like this are often not part of an algorithm's design and analysis but are of considerable concern for implementation.

Significant differences in the performance suggested by the analysis and the performance observed in faithfully rendered code may be more familiar than outright incorrect results. The choice of parameter-passing mechanism(s), the presence (or absence) of sufficient memory for the algorithm's data and code, and the way in which data structures are represented can all contribute to devastate an algorithm's actual execution-time performance.

How the execution of code is supported by the operating system and the implementation language's runtime system can also affect execution time performance. For example, the recovery of

unused dynamically allocated memory (garbage collection) may be automatically handled by the runtime support system or the programmer may be required to explicitly release the memory. In the case of automatic garbage collection, it is simply not predictable when the garbage collection is carried out. In the case of the programmer explicitly releasing the memory, failure to do so consistently may result in additional activity by the operating system's virtual memory management. In either situation, performance will likely be adversely effected and/or unpredictable.

2.5 Chapter 5: Implications of Nonuniform Memory for Software

Many of the assumptions underlying complexity analysis relate to uniformity. This chapter considers the often unrealistic assumption that the cost of memory access is uniform and what can be done to reduce the effects of nonuniformity.

The increasing access time as one moves down the memory hierarchy from processor registers to cache memory to main memory to external memory coupled with the decreasing capacities as one moves the opposite direction strongly argue that great care should be taken in how data should be organized and used in software. The convenience of virtual memory management drastically reduces the amount of tedious, error-prone, and often restrictive aspects of memory management that a programmer needs to be concerned with, but it also trades away considerable control over the movement of data around the memory hierarchy.

Understanding the system software, the structure of the memory hierarchy of the target computer, the memory needs of the program itself, and how all three interact can help to understand *why* performance may deteriorate. Of these things, a programmer can only exert some control over the last two, predominately by the way in which operations within the program are performed.

Focusing on matrix multiplication, the main section of Chapter 5 shows how to manually apply and analyze program transformations in order to reduce the effects of expensive data movement on execution-time performance. The author makes a compelling argument that more and better support for execution-time optimization via I/O analysis, program transformations and the implementation of nonstandard memory mapping functions should be provided by compilers. The absence of such support yields two unpalatable alternatives: accept inefficiency or accept unreadable code.

2.6 Chapter 6: Implications of Compiler and Systems Issues for Software

Fundamental issues of programming languages and the environments that support the execution of code written in them can have significant effects on performance and correctness. Chapter 6 considers several of these things and how they can contribute to performance degradation and/or unpredictability:

Memory issues aren't necessarily restricted to data and code. The implementation of recursion creates a memory need by the execution environment that can dynamically change. Typically it's not possible (or at least not easy) to predict the exact space needs of the recursion stack where the activations of functions, including the recursive ones, are managed.

Dynamic data structures are appealing because they can never be full - unless the execution environment runs out of available memory. Reusing chunks of memory that are no longer in use helps to delay the time at which memory is exhausted (in many cases, reuse may

avoid total depletion). Locating a chunk of memory to fulfill a program's request can require determining which chunks are no longer in use, making decisions about which available chunk should be allocated and even spending time defragmenting memory.

Parameter passing provides two opportunities for things to go awry. The two usual choices, call-by-value and call-by-reference, can both affect performance and correctness. In call-by-value, essentially a copy of the parameter is made and handed off to the called function. In call-by-reference a copy of the address of the parameter is given to the called function. Copying a parameter can be time consuming. Passing an address can lead to safety issues.

Somewhat less tangible, but equally important, issues discussed in this chapter include programming language properties such as initialization of memory, packing data and overspecification of execution order and aspects of optimizing compilers.

2.7 Chapter 7: Implicit Assumptions

An algorithm is often designed with some basic, implicit assumptions. For example, it is not unreasonable to assume that $n \geq 1$ for an algorithm that multiplies two $n \times n$ matrices. That n satisfies $n \geq 1$ is an implicit assumption in the algorithm's design. A programmer must treat that assumption as a *fundamental requirement* that is met by the n that the code deals with. Good, bullet-proof programming requires that the code handles the *exceptional situation* that arises when it is asked to execute with $n \leq 0$.

Chapter 7 outlines issues that implicit assumptions and exceptional situations can provoke. Determining exceptional situations and reacting to them can damage execution-time performance despite the validity of the algorithm's complexity analysis.

2.8 Chapter 8: Implications of the Finiteness of Representation of Numbers

Because numerical values are represented in a finite number of words each of which is of finite length, only a finite number of values can be represented. Consequently, issues of overflow, underflow and rounding come into play regardless of whether a program is doing heavy-duty numerical computation or simply compounding interest. Additionally, since the representation of values is binary, some numbers with perfectly good finite representations cannot be represented finitely (for example 0.3 has binary representation $0.01\overline{0011}$). Chapter 8 examines how these things impact equality testing, mathematical identities and convergence leading to potentially unreliable behavior or incorrect results.

2.9 Chapter 9: Asymptotic Complexities and the Selection of Algorithms

This chapter and Chapter 10 deal somewhat less with disappointments arising from the translation of algorithms into code and somewhat more with how to react to disappointing theoretical results.

Focusing on time complexity and using examples from sorting, selection and matrix multiplication as motivation this short chapter develops the concept of hybrid algorithms based on careful analysis of constant factors and crossover points.

2.10 Chapter 10: Infeasibility and Undecidability: Implications for Software Development

Impossibility can be an annoying problem and this chapter explores various aspects of the impossible: undecidability, prohibitively large complexity for decidable problems, and heuristics and approximation algorithms. Two of the 4 appendices are devoted to providing the background for understanding NP-completeness and undecidability but this chapter contains an extremely concise, 1-paragraph discussion of why NP-completeness is of importance to the programmer.

3 Opinion

It is unfortunate that it has taken so long for this book to appear. Had it appeared earlier there would likely be more and better coverage of the issues it addresses in software development curricula. In my opinion, this book should be on the bookshelf of anyone aspiring to become a good programmer.

There is very little in the book that software developers and programmers could not benefit from by being aware of. Very likely, good programmers eventually have to learn most of the material from the first five chapters of Part 2 the hard way: on the job at crunch time. Their ordeal could be reduced by exposure to it as part of their undergraduate study.

As the title suggests, this book would be a good auxiliary book in courses on algorithms, especially those intended for students intent on becoming software developers. Similarly with an advanced programming-type course. The book also would work well for self-study.

The book is somewhat dense. Physically, the text fully occupies the pages which made my eyes complain. There are a lot of concepts packed into the 10 chapters and they are tied together nicely making for compelling reading, in my opinion.

Thank you to Ernst Leiss and to Chapman & Hall/CRC for bringing this book to us.

Joint review of[4]
Algorithms
by Richard Johnsonbaugh and Marcus Schaefer
Pearson/Prentice-Hall, 004
52 pages, hardcover
and
Algorithms
by Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani
McGraw-Hill, 008
20 pages, softcover

Reviewed by
Dean Kelley (dean.kelley@mnsu.edu)
Minnesota State University, Mankato

[4]

1 Introduction

This is a joint review of two textbooks for a mid-level, or possibly upper-level, undergraduate algorithms course. In many cases this course follows a mid-level data structures course which covers up through trees. There have been several well written new books in recent years aimed at this course (including the two reviewed here). This likely reflects the continuing maturation of the discipline, an improved understanding of how to teach algorithms, and a clearer understanding of how and where the study of algorithms as an individual topic fits into an undergraduate computer science curriculum.

Both of the reviewed books have at their core material which has become standard for this course. Despite that commonality, the books are quite different.

2 Johnsonbaugh and Schaefer

Focusing on design techniques, Johnsonbaugh and Schaefer have written a textbook which should be accessible to students with essentially a second-year undergraduate "data structures and algorithms" background. The book has sufficient depth that it could contribute to a beginning graduate course as well.

Providing a fairly standard path through sorting and searching, divide-and-conquer, dynamic programming, greedy algorithms, P/NP, and parallel and distributed algorithms, the book contains hundreds of examples and roughly as many solutions to end-of-section exercises (spanning about 80 pages at the end of the book).

Chapters end with chapter problem sets which considerably expand the material of the chapter and/or explore interesting real-world applications. For example, the problems at the end of the chapter on dynamic programming explore optimizing static search trees, minimizing gap penalties in typesetting and the membership problem for context-free grammars. As another example, the problems at the end of the chapter on parallel and distributed algorithms explore alternative proofs of correctness for sorting networks and an alternative approach for leader election (choosing a processor to be "in charge").

Throughout the book the quality of presentation is extremely good. This reflects the considerable teaching experience and expertise of the authors and the number of iterations the material has been through in their courses at DePaul. An instructor's manual is available from the publisher and one author's website (Johnsonbaugh) contains a useful set of additional resources.

3 Dasgupta, Papadimitriou, and Vazirani

Dasgupta, Papadimitriou and Vazirani have written a book which takes a different approach to the algorithms course than Johnsonbaugh and Schaefer. Eschewing a formal and traditional presentation, the authors focus on distilling the core of a problem and/or the fundamental idea that makes an algorithm work. Definitions, theorems and proofs are present, of course, but less visibly so and are less formally presented than in the other text reviewed here.

The result is a book which finds a rigorous, but nicely direct path through standard subjects such as divide-and-conquer, graph algorithms, greedy algorithms, dynamic programming, and NP-completeness. You won't necessarily find every topic that you might want in all of these subjects,

but the book doesn't claim to be encyclopedic and the authors' presentation doesn't suffer as a result of their choice of specific topics.

Nice collections of chapter problems provide opportunities to formalize parts of the presentation and explore additional topics. The text contains plenty of "asides" (digressions, illuminations, addenda, perspectives, etc.) presented as boxes on some of the pages. These little side trips are fun to read, enhance the presentation and can often lead to opportunitites to include additional material in the course. It has been my experience that even a disinterested, academically self-destructive student can find something of sufficient interest in these excursions to grab their attention.

In addition to the relatively standard subjects listed above, the book ends with a chapter on quantum computation, specifically applied to factoring numbers. This chapter bookends nicely with the first chapter's motivation for studying algorithms (which is based around factoring and primality testing). The chapter's presentation is focused and accessible. Beginning with a quote from Richard Feynman, "I think I can safely say that no one understands quantum physics," the chapter proceeds to demonstrate that quantum *computing* just might be understandable.

4 Opinion

Either of these books would support a mid-level or upper-level undergraduate course in algorithms very well. Both are quite accessible to reasonably engaged (and prepared) undergraduate students. Both would do a good job of feeding the intellectual curiousity of enthusiastic students, though in somewhat different manners.

Johnsonbaugh and Schaefer's book has a density of topics which, when coupled with its thoroughness (and resulting size), may require some care and guidance if a student is turned loose on it independently. On the other hand, the quantity of topics and the depth and care taken in their coverage make it an excellent textbook for a regular course. Dasgupta, Papadimitriou, and Vazirani's book might take an unguided student farther faster, but students may need help filling in details. On the other hand, its tightly focused progression seems to yield an excellent big-picture perspective when used in a course.

There is considerable difference in price of these two books. The current (March, 2008) new price of Johnsonbaugh and Schaefer's book on amazon.com is $96.20 and the price of Dasgupta, Papadimitriou and Vazirani's is $33.75. (The used book prices of the books are roughly equal) A portion of this difference likely arises from hardcover/paperback production cost differences and from the considerable difference in page counts. As one might surmise from my above review, I like *both* of these books and am not able to form an opinion about cost versus quality. However a smaller, cheaper, well-done textbook from a major publisher is a welcome surprise.

Review[5] of
Design and Analysis of Randomized Algorithms:
Introduction to Design Paradigms
Published by Springer. $56.00
Author of Book: Juraj Hromkovič
Review by Marios Mavronicolas

[5]

1 Introduction

Everybody in the Theory of Computing community (especially those who work on Algorithm Design) is well acquainted with the concept of Randomization. It is not an exaggeration to say that Randomization is currently one of the major approaches to Algorithm Design. This is evidenced by the fact that modern leading textbooks on Algorithm Design and Analysis often include a chapter on Randomization (next to chapters on Dynamic Programming, Approximation and Linear Programming, to mention a few of the other standard techniques for the design of Computer Algorithms). Besides its (partial) coverage in general textbooks on Algorithm Design, there are already two textbooks devoted to Randomization and emphasizing Randomized Algorithms, Random Processes and Probabilistic Analysis of Algorithms; the textbook by Hromkovič is the third in this row. In my view (and see my comments below on this), this book still gives a fresh and interesting point of view to Randomization and Randomized Algorithms.

2 Content

The book is organized along a remarkable logical structure. The first two chapters are introductory, while each of the remaining five chapters focuses on a particular technique for designing Randomized Algorithms, together with analyzing a few representative Randomized Algorithms that were designed using the particular technique.

The first chapter starts on with a philosophical discussion on Randomness and Randomization. Although the less philosophically inclined reader may skip Section 2.1 to proceed faster to the most substantial body of the chapter, I whole heartedly recommend Section 2.1 to all lovers of philosophical principles who embarn on learning the beneftis and the mathematical principles of Randomized Algorithms. Section 1.2 is a very crucial section in the book since it provides the first concrete example of a Randomized Algorithms while demonstrating the efficiency benefits of using Randomization in Algorithm Design. The chapter concludes with a self-portrait of the book (Section 1.3) and immense advice to the student (Section 1.4) and the teacher (Section 1.5) using the book.

Section 2 is a background chapter. It is devoted to three main axes: the mathematical foundations of Randomization, which are the fundamentals of Probability Theory (Section 2.2), the theoretical foundations of Randomized Algorithms (Sections 2.3, 2.4 and 2.5), and an outline of the main techniques for the design of Randomized Algorithms (Section 2.6). (Actually, the third axis may be seen as a prologue to the rest of the book.) In more detail, Section 2.3 squeezes the most elementary facts from Probability Theory that are a must for the designer and analyst of Randomized Algorithms. Readers with a strong (or even average) background in Probability Theory may choose to skim through this section very fast. Section 2.3 defines the main complexity-theoretic measures for the evaluation of Randomized Algorithms under two distinct theoretical models. The provided examples are best chosen to demonstrate the analysis of the presented measures. Further, Section 2.4 classifies Randomized Algorithms as Las Vegas and Monte Carlo (with various levels of error). A corresponding classification of Randomized Algorithms for optimization problems is pursued in Section 2.4. Each of the divisions of Section 2.6 is a fast prelude to a subsequent chapter; some readers may prefer to proceed directly to the subsequent chapters. This chapter is significantly longer than any other chapter.

The technique of Fingerprinting is the subject of Chapter 4. Chapter 4 present several inter-

esting applications of the technique; in fact, some of the applications were indeed the original cases out of which the technique emerged; of those, verification of matrix multiplication (Section 4.4) merits a special mention.

The method of failing the adversary is one of the most basic techniques for the design of Randomized Algorithms. As the authors point out in Chapter 3, this method could also be called the method of avoiding the worst-case inputs. In essence, the method attempts to create a suitable set of deterministic algorithms such that the randomized algorithm is a distribution on this set achieving that a randomly drawn deterministic algorithm (according to the distribution) will compute the correct result with high probability. The particular examples of Hashing (Section 3.2) and Universal Hashing (Section 3.3) illustrate well the technique, although their technical level is a bit above the average (over all sections of the book). Section 3.4 is an advanced section on applying the technique of foiling the adversary to online algorithms.

Chapter 5 turns to the very successful technique of Random Sampling and its relative technique of Success Amplification. (The author argues that these two techniques can be presented together; however, one could see them to be completely separate). The examples employed for both techniques in this chapter are both excellent - especially (in my opinion) the example of generating quadratic nonresidues (Section 5.4).

Another yet successful technique for the design of Randomized Algorithms is that of Abundance of Witnesses, presented in Chapter 6. Some mathematical background (included in the book's Appendix) is necessary for following this chapter. The Randomized Algorithms for Primality Testing (Section 6.3) and Prime Generation (Section 6.4) are probably excellent examples that demonstrate the potential and the mathematical wealth of the technique.

Finally, Chapter 7 describes the technique of Randomized Rounding, which is based on relaxation to linear programming and subsequent rounding of the solution. The provided examples (especially the one on MAX-SAT) are all excellent. Section 7.4 is perhaps the only section in the book that demonstrates how two of the presented techniques can be combined together; I strongly recommend it since it promotes the synthetic abilities of the reader.

3 Opinion

This is a very good to excellent textbook on the Design and Analysis of Randomized Algorithms. Its unique (perhaps distinguishing) feature is that it explicitly isolates and promotes the most important design techniques; this is really a great experience for the incoming theorist to specialize on Randomized Algorithms. I would have liked to see some more examples of Randomized Algorithms for hard counting problems via the celebrated Markov Chain technique. Some more exercises and a more detailed discussion on an expanded bibliography would also be very beneficial to the reader. Some discussion on (techniques for) Randomized Distributed Algorithms would also add.

Review[6] of

Theoretical Aspects of Local Search

Author of book: Wil P. A. J. Michiels, Emile H. L. Aarts, and Jan H. M. Korst

Springer in the EATCS Series Monographs in Theoretical Computer Science, 007

ISBN: 78-3-540-35853-4, hardcover, 235 pages, EUR 54.95

[6]

Author of Review: Reviewed by Jakub Mareček

"Theoretical aspects of local search? That must be like two pages long!" said a seasoned theoretical computer scientist, when he learned what a book was under review. Many computer scientists do indeed share a scathing contempt for anything heuristic. Perhaps this is because they do not realise that many real-life optimisation problems do boil down to large instances of hard-to-approximate problems, well beyond the reach of any exact method known today. Perhaps they do not realise that performance of exact solvers for hard problems is largely dependent on the quality of in-built heuristics. Certainly, this leads to the irrelevance of a large body of work in Theoretical Computer Science to the operations research industry, and a certain lack of analytical approach in real-life problem-solving, to say the least. Any attempt to change this situation would be most welcome. Local search, readily admitting theoretical analysis, might be a good starting point.

Design, analysis, and empirical evaluation of algorithms for hard problems generally involve a number of interesting challenges. No matter whether you are designing integer programming solvers, model checkers, or flight scheduling systems, you have to use heuristics, which present-day Theoretical Computer Science by and large ignores. For instance in precoloured bounded/equitable graph colouring, which underlies many timetabling and scheduling problems, the present-best theoretical result says it is $\mathcal{NP}$-Complete for trees and hence a tree-width decomposition doesn't help [4], in addition to being hard to approximate in general [15]. Despite the depth of these negative results, their practical utility might be somewhat hard to see for a person working on solvers for the very problem. The divorce of theoretical study of approximability of rather obscure problems and special cases from the development of solvers for messy real-life problems of operations research, is also reflected in the in the usual undergraduate Computer Science curriculum. If any attention is given to heuristics (see [11] for a rare exampl is often confined to the plain old greedy "iterated improvement" and some elementary stochastic local search heuristics, which may seem too trivial even to an undergraduate. The importance of the choice of neighbourhood and objective function is often stressed, but rarely demonstrated. An abstract discussion of the trade-off between the search across many basins of attraction within the search space (diversification) and the convergence to local optima within each basin (intensification) sometimes follows. (See the gripping personal account of [6].) Only few textbooks proceed to mention heuristic pre-processing and decompositions, or the role of pre-processing within heuristics (at least, for instance, sorting with tie-breaking). Even less attention is usually focused on the concept of relaxation and restriction, such as forbidding features detected in many bad solutions [13], or fixing features common to many [12], instance analysis determining which algorithm to use, or auto-tuning [7], vital to industrial solvers. As non-trivial analyses of expected behaviour remain rare, the treatment is often concluded with a description of the worst case scenario, where the heuristic takes infinitely long to converge or does not converge at all. Not only research, but also the usual treatment in the curriculum tend to remain rather distant to real-world solver design and seem best read with a funeral march playing in the background.

The authors of the book under review are in an excellent position to write a very different account. On one hand, at least two of them are well-known in the algorithm design community, not least for their grim analyses of local search heuristics (see the conclusions of [1]: "simulated annealing algorithm is clearly unsuited for solving combinatorial optimization problems [to optimality]") and vocal contempt for "home-made voodoo optimisation techniques inspired by some sort of physical or biological analogy rather than by familiarity with what has been achieved in mathematics" [3]. On the other hand, however, all three authors are involved in the design of heuristics solving real-life problems at Philips Research. It might then seem reasonable to expect

the book under review to be more upbeat than the usual dead march.

The book under review is clearly aimed at advanced undergraduate students, although the blurb mentions "researchers and graduate students" as the intended audience. At 185 pages bar the appendices, the book is of the right length for a relaxed, largely self-contained term-long course. In chapter two, it introduces five problems used as running examples throughout the book (TSP, vertex colouring, Steiner trees, graph partitioning and make-span scheduling). Chapters 3 and 4 introduce the concept of a neighbourhood and discuss properties of various neighbourhoods for the example problems. Chapter 5 introduces rudiments of approximability and hardness of approximation and presents elementary proofs of performance guarantees for some of the local search methods introduced previously. Chapter 6 recalls the existence of some non-trivial results on the complexity of local search. Chapter 7 previews some general heuristic design schemes ("metaheuristics"), such as simulated annealing. Chapter 8 introduces Markov chains, conditions of their ergodicity and convergence properties of simulated annealing. Finally, three appendices include a list of 12 problems-neighbourhood pairs, which are known to be $\mathcal{PLS}$-complete (see below). All chapters are accompanied by exercises and most have the very comforting feel of an easily understandable lecture transcribed, while maintaining a reasonable level of rigour.

For a narrowly-focused researcher in Complexity, only Chapter 6 may be of immediate interest. It provides a concise overview of $\mathcal{PLS}$-Completeness, a framework for reasoning about complexity of local search algorithms and an alternative measure of complexity of "easier hard" problems, introduced by [10]. A pair of a given problem and neighbourhood structure falls into the class of $\mathcal{PLS}$ (poly-time searchable), if an initial solution can be obtained using poly-time, there is a poly-time evaluation function, and a poly-time test of optimality within a neighbourhood, producing an improving solution together with the negative answer. From this follow the concepts of $\mathcal{PLS}$-Reducibility and $\mathcal{PLS}$-Completeness. Unless $\mathcal{NP}$ = co-$\mathcal{NP}$, problems in $\mathcal{PLS}$ are not $\mathcal{NP}$-Hard. For certain problems, the choice of neighbourhood seems relatively restricted and results suggesting the impossibility of finding an improving solution fast in a commonly used neighbourhood might seem damning. Unlike in approximability, results in $\mathcal{PLS}$-Completeness are, however, tied to a particular neighbourhood, which is similar to results in parametrised complexity being tied to a particular decomposition. Introduction of a new neighbourhood structure or decomposition [8] may bring new breakthroughs. These connections and ramifications are, however, not discussed in the book. At 36 pages, the chapter on $\mathcal{PLS}$-Completeness is more concise than the paper that has introduced the concept [10] or the exposition by [14] "it is based on" (p. 98, sic!), and does not seem to represent good value for money if the rest of the book is of little or no interest to the reader.

Opinion: Overall, the book provides a concise and easily understandable introduction to the basics of local search, an important concept in the design of heuristics. As it gracefully omits anything and everything a *theoretical* computer scientist need not learn from the arguably bloated jargon of heuristic design as well as applications to complex real-life problems, it is well-suited for a term-long course on heuristic design for theoretically-inclined undergraduates and first-year graduate students. In other circumstances, either a chapter in a good undergraduate algorithm design textbook [11] with some project-work, a broader survey of heuristic and exact optimisation [5], or a more comprehensive study on local search [2, 9] might be worth consideration.

References

[1] E. H. L. Aarts, J. H. M. Korst, and P. Zwietering. Deterministic and randomized local search. In P. Smolensky, M. C. Mozer, and D. E. Rumelhart, editors, *Mathematical perspectives of neural networks*, pages 43–224. Lawrence Erlbaum Associates, Mahwah, NJ, 996.

[2] E. H. L. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. Wiley-Interscience, Chichester, UK, 997.

[3] E. H. L. Aarts and J. K. Lenstra. Preface. In *Local Search in Combinatorial Optimization* [2], pages vii–viii.

[4] H. L. Bodlaender and F. V. Fomin. Equitable colorings of bounded treewidth graphs. *Theoret. Comput. Sci.*, 49(1):22–30, 2005.

[5] E. K. Burke and G. Kendall, editors. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer, New York, NY, 005.

[6] F. Glover. Tabu search – uncharted domains. *Ann. Oper. Res.*, 49:89–98, 2007.

[7] F. Glover and H. J. Greenberg. New approaches for heuristic search: A bilateral linkage with artificial intelligence. *European J. Oper. Res.*, 9(2):119–130, 1989.

[8] P. Hliněný, S. il Oum, D. Seese, and G. Gottlob. Width parameters beyond tree-width and their applications. *Computer J.*, to appear. 0.1093/comjnl/bxm052.

[9] H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann, San Francisco, CA, 004.

[10] D. S. Johnson, C. H. Papadimtriou, and M. Yannakakis. How easy is local search? *J. Comput. Syst. Sci.*, 7(1):79–100, 1988. `http://dx.doi.org/0.1016/0022-0000(88)90046-3`.

[11] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman, Boston, MA, 005.

[12] S. Lin. Computer solutions to the travelling salesman problems. *Bell Syst. Tech. J.*, 4(10):2245–2269, 1965.

[13] K. Steiglitz and P. Weiner. Some improved algorithms for computer solution of the traveling salesman problem. In *Proc. th Annual Allerton Conference on Circuit and Systems Theory*, pages 14–821, Urbana, IL, 1968. `http://note.acm.org/0.1145/800113.803625`.

[14] M. Yannakakis. Computational complexity. In *Local Search in Combinatorial Optimization* [2], pages 19–55.

[15] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory Comput.*, (6):103–128, 2007. `http://www.theoryofcomputing.org/articles/main/v03/a006`.

Review of[7]

The Traveling Salesman Problem: A Computational Study
Authors of Book: Applegate, Bixby, Chvátal, and Cook
Princeton University Press, 06 pages, $46.95
Review by William M Springer II

1 Introduction

Given a set of cities and roads (or other means of travel) between them, find the best route by which to visit every city and return home.

Thus is stated one of the most intensely studied problems in mathematics, the Traveling Salesman Problem (TSP). In various forms, the problem can be found in literature going as far back as the 9th century (expressed as a knight's tour); studies have been done on the ability of children and animals to solve simple TSPs. The authors of "The Traveling Salesman Problem: A Computational Study" have been working on the problem for nearly twenty years, finding solutions to TSPs with as many as eighty-five thousand cities; the book describes the methods and code they developed for solving large cases of the TSP. The book is basically divided into three sections; this review will cover only the first section in depth, leaving the latter two for the interested (and dedicated) reader.

2 Why solve the TSP?

Chapter one gives an introduction to the traveling salesman problem itself, along with a brief history of the various forms in which it has appeared over the last millennium. The mathematical history of the TSP is given, followed by an outline of the rest of the book. The TSP is also explained to be an NP-hard problem, justifying the use of approximation techniques; as it is in fact NP-complete, techniques developed in studying it can also be applied to other NP-complete problems. This is the only chapter of the book which contains no mathematics, and is appropriate for any interested reader.

In chapter two, some applications of the TSP are developed, including both traditional transportation problems such as school bus routes and more general applications ranging from gene-mapping to scheduling space-based telescopes. This chapter also contains very little mathematics and gives the reader an appreciation for the many applications of the TSP.

3 Representing the problem

Chapter three is where the meat of the book really begins. Given n cities, a tour is represented as an incidence vector x of length $n(n-1)/2$, with each component of the vector set to 1 if that edge is part of the tour (and set to zero otherwise). If c is a vector containing the costs to travel between every pair of cities, then the TSP can be restated as a minimization problem; letting S be the set of vectors representing all possible tours, we want to minimize $c^T x$ such that $x \in S$. Notice that the there must be a total of two edges going to every city if we simply minimize $c^T x$ subject

[7]

to this restraint without worrying about the restriction of having a valid circuit (this might, for example, give several unconnected cycles) we have a lower bound on the size of any valid tour. At this point we are not restricted to full edges; there may be "half edges" (displayed in the graph as dotted edges, they have weight 1/2) where three half edges form a cycle; in this case, two half edges incident on the same vertex will have the same weight as one full edge, either of which might be used in the final solution. As such, linear programming methods can be used to both establish a lower bound on solutions and give a starting point which can often be modified to give a "pretty good" solution to the TSP. Once this starting point is found, we can add a series of conditions until eventually the graph becomes a tour. The rest of the book consists largely of methods for either modifying the current graph to produce a tour, or moving from the current tour to one with lower weight.

4 History of TSP Computation

With small TSPs now having been solved computationally, the 1950s saw a flurry of activity around the problem. During these years, branch and bound algorithms became popular. In the previously described cutting-plane method, once a tour is obtained more and more cuts are added to improve the path, and the improvements get smaller and smaller. With the branch and bound method, once the improvements drop below a certain level, the algorithm branches; a vector α and two numbers β' and β'' are chosen, with $\beta' < \beta''$, such that every $x \in S$ satisfies either $\alpha^T x \leq \beta'$ or $\alpha^T x \geq \beta''$. There are now two subproblems, which are solved separately:

$$\text{minimize } c^T x \text{ subject to } x \in S \text{ and } \alpha^T x \leq \beta'$$
$$\text{minimize } c^T x \text{ subject to } x \in S \text{ and } \alpha^T x \geq \beta''$$

At a later time, either or both of these subproblems could be split into smaller subsubproblems, and so on. We are constantly solving a problem of the form

$$\text{minimize } c^T x \text{ subject to } x \in S \text{ and } Cx \leq d$$

for some system $Cx \leq d$ of linear inequalities, where C is a matrix and d a vector; subproblems can be discarded if they are found to have minimal weight solutions at least as large as some previously discovered element $x \in S$. The branch and bound method was improved by incorporating spanning trees, giving another method of finding lower bounds for the TSP. The tree can then be used to force certain edges to be excluded or included in the subproblems, giving stronger bounds and thus faster algorithms.

In the 1960s, dynamic programming came into use for TSP computations, based on the idea that in any optimal tour, once some number of cities have been visited, the path through the remaining cities must still be optimal. As such, we can build the tour step by step; first generate a list of all minimum-cost paths through subsets of k cities, then extend them to paths through sets of k+1 cities. While the amount of data that needs to be processed increases rapidly as k grows, it was shown that dynamic programming algorithms can solve any TSP on n cities in $O(n^2 2^n)$ time, a significant improvement over the n! time required to enumerate all tours, and in fact this is the best time complexity for any known algorithm that can solve any instance of the TSP. Unfortunately, this is still impractical for larger TSP instances, so in practice algorithms are needed which are

tailored to a particular type of TSP problem in order to obtain a solution in a reasonable amount of time.

5 The Rest of the Book

As more sophisticated (and faster) linear programming software became available, the focus returned to cutting-plane algorithms, and most of the remainder of the book involves linear programming methods. Chapters five through eleven describe ways of obtaining ever-better cuts. Chapter five, the introduction for this section, gives a very brief introduction to graph theory and a formal summary of linear programming. It then gives an outline of the cutting plane method. There is also a brief discussion on dealing with numerical errors; fixed-point arithmetic is used to avoid rounding errors. Chapter six involves finding subtour cuts using PQ trees; most of the remainder of this section deals with the use of combs. The third section, chapters twelve through sixteen, deals with actually creating software for finding tours; chapter sixteen is specifically on the Concorde software developed for large-scale instances of the TSP. The software, approximately 130,000 lines of C code, is available at www.tsp.gatech.edu. The algorithm was used to find a new best path for the World TSP, an instance containing 1,904,711 cities (available at www.tsp.gatech.edu/world) specified by latitude and longitude, with travel cost being the great circle distance on Earth; the algorithm was terminated in 401.7 days, after 18 local cuts, with distance 0.058% over the the minimum possible cost. In another case, a 1,000,000 city random Euclidian instance, the algorithm processed 28 local cuts in 308.1 days, finishing with a gap of only 0.029%. Chapter seventeen briefly mentions several avenues for future research that may be able to speed up the algorithm. As near-optimal solutions for the largest known TSP problems were obtained using Concorde, the authors have created a new TSP instance for future work: the 526,280,881 celestial objects in the United States Naval Observatory catalog, with the goal being to move a telescope to view every object rather than actually traveling between them. At the time of the book's publication, a tour had already been produced and proven to be within 0.796% of the cost of an optimal tour.

6 Conclusion

I found this book to be interesting and well-written. I would not recommend it to someone looking for a layman's view of current work on the TSP, as this is most definitely a research book, and the reader will want to have at least a basic grasp of set theory and linear algebra before attempting it. That said, the authors have made a strong attempt to provide background material as needed, and the interested reader who is willing to put in the needed effort will find it rewarding. For anyone with the required background who has an interest in the Traveling Salesman Problem, this book is an exceptionally good value and well worth its modest price.

Review by[8] Alice Dean of
Visibility Algorithms in the Plane[9]
by Subir Kumar Ghosh
Cambridge University Press, 2007, Hardcover, 332 pages, $108.00

[8] © Alice M. Dean, 2009
[9] © Alice M. Dean, 2009

Reviewer: Alice M. Dean (adean@skidmore.edu)
Skidmore College

1 Overview

Computational Geometry is a young field, tracing its beginnings to the PhD thesis of Michael Shamos [6] in 1978. Its many applications include, to name a few, computer graphics and imaging, robotics and computer vision, and geometric information systems. Topics studied within computational geometry include arrangements, convex hulls, partitions, triangulation of polygons, Voronoi diagrams, and visibility.

One of the most famous theorems in computational geometry is the *Art Gallery Theorem*, and this theorem also serves as an example of the focus of the book under review. Posed by Victor Klee in 1973, it asks how many stationary guards are required to see all points in an art gallery represented by a simple, n-sided polygon. The answer, given first by Chvátal [3] and later by Fisk [5], using an elegant graph-theoretic proof, is that $\lfloor n/3 \rfloor$ guards are always sufficient and may be necessary. Questions such as this one, of visibility within a polygon, are the subject of **Visibility Algorithms in the Plane**, by S. Ghosh.

2 Summary of Contents

The book, which contains eight chapters, is a thorough and detailed investigation of theorems and algorithms for a variety of types of polygon visibility problems. It is aimed at an audience of graduate students and researchers who already have a background in computational geometry, and it also assumes that the reader has a general knowledge of algorithms and data structures. The first chapter is introductory, and each of the other seven chapters focuses on a particular type or aspect of visibility. Each chapter begins by reviewing relevant theorems and problems; then algorithms and other results are presented. Each chapter ends with a discussion of other issues related to the chapter's topic, including on-line and parallel algorithms.

- *Chapter 1 Background* defines the several notions of visibility that are subsequently discussed in the book. Properties of polygons and triangulations are covered, and the complexity model *real RAM*, which is used in the rest of the book, is introduced. Chap. 1 also covers the Art Gallery Problem.

- *Chapter 2 Point Visibility* considers the problem, given a polygon P and a point $q \in P$, of computing the visibility polygon $V(q)$, which is the polygonal sub-region of P consisting of all points visible from q. In other words, $V(q)$ is the set of all points $p \in P$ such that the straight line from q to p lies entirely within P. Results and algorithms are given for simple and non-simple polygons, and for non-winding and winding polygons.

- *Chapter 3 Weak Visibility and Shortest Paths* considers three variations of visibility along an edge v_iv_{i+1} of a polygon P. P is *completely visible* from v_iv_{i+1} if every point $p \in P$ is visible from every point $w \in v_iv_{i+1}$. P is *strongly visible* from v_iv_{i+1} if v_iv_{i+1} contains a point w such that every point $p \in P$ is visible from w. P is *weakly visible* from v_iv_{i+1} if each point $z \in P$ is visible from at least one point $w_z \in v_iv_{i+1}$. Further, the *weak visibility polygon* of P from

v_iv_{i+1} is the set of all points $z \in P$ such that z is visible to at least one point of v_iv_{i+1}. Weak visibility polygons are characterized in terms of Euclidean shortest paths between vertices, and algorithms for computing weak visibility polygons are given. Algorithms are also given to compute Euclidean shortest paths between points in P.

- *Chapter 4 LR-Visibility and Shortest Paths:* P is an LR-visibility polygon if its boundary contains two points s and t such that every point on the clockwise boundary from s to t is visible to at least one point on the counterclockwise boundary from s to t. LR-visibility polygons, which generalize weak visibility polygons, are characterized in terms of Euclidean shortest paths. Algorithms to recognize LR-visibility polygons, and to compute shortest paths within LR-visibility polygons, are presented.

- *Chapter 5 Visibility Graphs:* The *visibility graph* of a polygon P has as its vertices the vertices of P, with two vertices adjacent if they are visible to each other in P. Algorithms are given to compute visibility graphs of simple polygons and polygons with holes. Algorithms are also given to compute the *tangent visibility graph* of P, which contains those edges of the visibility graph that are relevant for computing Euclidean shortest paths.

- *Chapter 6 Visibility Graph Theory* considers questions complementary to those of Chap. 5, namely, given a graph G, determine whether it is the visibility graph of some polygon, and if so, construct such a polygon. These two questions are called, resp., the graph *recognition* and graph *reconstruction* problems. These are both unsolved problems, and their complexity is unknown as well, except it is known that the reconstruction problem is in *PSPACE* [4]. Necessary conditions for a graph to be a visibility graph are given, and testing algorithms are also given for these conditions. In addition, the chapter gives algorithms for recognizing special classes of visibility graphs.

- *Chapter 7 Visibility and Link Paths:* In contrast to a Euclidean shortest path in a polygon P, a *minimum link path* minimizes the number of line segments in a piecewise linear path joining two points of P, and the *link diameter* of P is the maximum number of links in any minimum link path in P. Algorithms are given to find minimum link paths between two points of P, and also to compute the link diameter and related parameters.

- *Chapter 8 Visibility and Path Queries:* Query problems in computational geometry are problems that require a large number of similar computations within a common polygonal domain. For example, given an arbitrary polygon P with n vertices and a point $q \in P$, there is an algorithm to compute the visibility polygon of q in time $O(n \log n)$ [1]; thus this problem for m points, $\{q_1, q_2, ..., q_m\} \subseteq P$, can be solved in time $O(mn \log n)$. But with $O(n^2)$ preprocessing time, the question for m points in P can be answered in time $O(mn)$ time [2]. This query algorithm is presented, as well as query algorithms for the ray-shooting problem (given $q \in P$, find the point on the boundary of P closest to q), Euclidean shortest path, and minimum link path.

3 Comments and conclusion

The book has ample exercises interspersed in the text. They vary in difficulty from brief thought exercises to research level investigations. The text is conversational in tone, yet clear and rigorous

in its exposition. It has an index and an extensive bibliography; when using the former while preparing this review, I found it was missing a few fundamental terms, my only criticism of the book. In general, this text accomplishes its intended purpose well – providing a graduate-level text on visibility algorithms that can also serve as a useful reference.

References

[1] T. Asano. Efficient algorithms for finding the visibility polygons for a polygonal region with holes. *Trans. IECI Japan*, 8:557–559, 1985.

[2] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, :49–63, 1986.

[3] V. Chvátal. A combinatorial theorem in plane geometry. *J. Combin. Theory Ser. B*, 8:39–41, 1975.

[4] H. Everett. *Visibility graph recognition.* PhD thesis, University of Toronto, 990.

[5] S. Fisk. A short proof of Chvátal's watchman problem. *J. Combin. Theory Ser. B*, 4:374, 1975.

[6] M. Shamos. *Computational geometry.* PhD thesis, Yale University, 978.

Review of[10]
A Course on the Web Graph
Author of Book: Anthony Bonato
Publisher: American Mathematical Society, Providence, Rhode Island, USA
Series: Graduate Studies in Mathematics
Volume 9, Year 2008, 185 pages

.

Reviewer: Elisa Schaeffer, D.Sc., Associate Professor
Universidad Autónoma de Nuevo León (UANL)
San Nicolás de los Garza, Mexico

1 Introduction

In the past ten years, *network modeling* has become highly fashionable, combining elements of traditional study of *random graphs* and *stochastic processes* with methods of *statistical physics*. In the late 1990s, works such as the article of Watts and Strogatz (Nature, 393:440–442, 1998) and Barabási and Albert (Science, 286:509–512, 1999) led to a wave of *complex systems* studies in graph theory, with special emphasis on the study of the structural properties of *natural networks*, that is, graph representations of real-world systems. Combined with the increasing importance of the Internet and the World-Wide Web, studies of these two networks in particular began to flourish, through seminal studies such as that of Faloutsos, Faloutsos and Faloutsos (ACM SIGCOMM'99, pp. 215–262) and Albert, Jeong and Barabási (Nature, 401:130–131, 1999).

[10] Elisa Schaeffer ©2009

Representing the Web as a graph comes about quite naturally. A *graph*, formally, is a pair of two (usually finite) sets, $G = (V, E)$, where the elements of V are called *vertices* or nodes and the elements of E are pairs of vertices called *edges*. In the case of the World-Wide Web, the vertices are the web pages. Two vertices u and v are connected by a (directed) edge (u, v) in the graph if the web page represented by the vertex u has a hyperlink pointing to the web page represented by the vertex v. The computational challenge is due to the amount of vertices and edges in the graph representation of the Web as a whole, and the practical challenge arises from the difficulty of obtaining but a local snapshot of this ever-evolving, massive network.

In his 2008 textbook, Anthony Bonato reviews the fundamentals of modeling and analyzing the Web graph and summarizes many of the most important results published in the past decade of Web graph studies. His approach is rather mathematical, but completely accessible to researchers and graduate students with previous exposure to basic discrete mathematics and probability theory.

2 Summary of contents

Bonato starts out in Chapter 1 with a brief review on the basic concepts of graph theory and probability that are well-written for the target audience — Mathematics graduate students — or simply any reader with previous knowledge on the topics who seeks to refresh his memory, whereas a Computer Science graduate student with less exposure to mathematics may feel lost at times with the terminology and the numerous definitions that are needed throughout the book.

In Chapter 2, with the basic terminology covered, Bonato discusses estimations of the total number of webpages as well as the average distance between any two webpages or the diameter of the Web graph, being the maximum distance between any two webpages. Structural properties of the Web graph such as *power-law degree distributions* and the small-world property are explained as well as the presence of *clusters* of webpages, known as the *community structure* of the Web. Bonato also briefly draws the connections to other networks in which these properties have been found and studied.

The study that Bonato gives of *random graph models* in Chapters 3 and 4 is an excellent summary of the immense quantity of scientific work done by mathematicians, physicists and computer scientists in the past 60 years, going from the uniform models studied by Erdös and Rényi and the *probabilistic method* that permits to characterize many interesting properties of random graphs to the models and methods designed to replicate some characteristics of the Web graph in particular, such as the Barabási-Albert method of *preferential attachment* and variants, *linearized chord diagram* model of Bollobás et al. and the *copying model* of Kleinberg et al. that introduce vertices and edges into a graph in an incremental fashion to grow a graph that structurally resembles the Web graph in some predefined sense. These models are throughly analyzed and many results are included with the proofs to illustrate the mathematics required in their derivation.

Also models that permit the deletion of vertices and models that incorporate a geometric or spatial sense of proximity are explained, as well as a mathematically more easily approachable non-incremental model of Chung and Lu that creates a graph to match a given degree distribution, building on previous work of Molloy and Reed. Bonato provides the reader with an analytic to-do list with some of the open questions in the modeling of the Web graph. Chapter 3 includes a good summary of the probabilistic techniques needed to analyze properties of random graph models for those that have not had the pleasure of reading the excellent textbook "Probabilistic Method" by Alon and Spencer (Wiley, 2000).

Bonato's review of the methods to search the Web (Chapter 5) are of interest even to those that are not into modeling, as he provides a mathematical, graph-theoretical explanation of the algorithms for searching the Web, namely for ranking the web pages. The mathematical background of linear algebra and stochastic processes included makes the chapter on Web search nearly a stand-alone for those readers who only seek to know how Google works. The classic algorithms of the Web search literature (PageRank, HITS and SALSA) are all explained in detail and briefly analyzed with respect to sensibility to perturbations and initial conditions.

In Chapter 6, Bonato attends the dynamic nature of the Web by considering the Web graph to be an infinite (although countable) graph instead of a massive, finite system. This approach permits to arrive at types of emergent behavior not present in finite graphs and a better understanding of those models of the Web graph that evolve over time as their limit behavior can be studied.

Chapter 7 closes the book with a discussion of future directions and the birth of the so-called "Internet Mathematics", which is growing to be a discipline in its own right, including topics such as *spectral analysis* (studying the eigenvalues of different matrix representations of graphs) and *epidemiological modeling* (in particular virus propagation).

3 Opinion

Overall the text is well-structured and pleasant to read for mathematically oriented readers and very fit to be taken as a text book also for less mathematical audience. Each chapter comes with a set of exercises, ranging from simple to challenging. For the non-mathematical computer scientists, perhaps an easier approach to the Web Graph was taken by Baldi, Fransconi and Smyth in their textbook "Modeling the Internet and the Web: Probabilistic Methods and Algorithms" (Wiley, 2003). Having both texts at the library is recommendable for anyone planning to enter the field, complimenting them with purely mathematical texts on graph theory (basic, algebraic and algorithmic) and probability theory (with emphasis on stochastic processes).

For newcomers to the field of Internet Mathematics, also the well-selected bibliography of Bonato's book will come handy in selecting what to read in the vast amount of literature produced on the topic in the past decade (it is no longer feasible to read "everything relevant" published on the Web graph in order to begin studying it). I personally eagerly await for the opportunity to give a course based on Bonato's book on postgraduate level - not so much for teaching about the Web graph in particular, but rather using it as a wonderfully rich case study to teach mathematical network modeling and the power of graph-theoretical methods in analyzing, predicting and explaining structural properties of large and complex real-world systems.

Review[11] of
Higher Arithmetic: An Algorithmic Introduction to Number Theory
Author of Book: H. M. Edwards
Publisher: American Mathematical Society
Student Mathematical Library Vol. 45, 2008
168 pages, Softcover

Review by
Brittany Terese Fasy brittany@cs.duke.edu

and David L. Millman dave@cs.unc.edu

1 Introduction

Higher Arithmetic presents number theory through the use of algorithms. The motivating theme of the book is to find all solutions of $A\Box + B = \Box$[12] In other words, given two numbers A and B find all pairs (x, y) such that, $Ax^2 + B = y^2$. This book is written for a computer science or a mathematics undergraduate student to understand number theory. The students should be familiar with algorithms and proofs before reading this book, but the knowledge of Big O notation is not necessary.

Before proceeding, we make a note here. For the purposes of this book (and hence this review), only the non-negative integers will be called numbers.

2 Summary

This book is broken into 31 short chapters (about three pages each). Each chapter is accompanied by study questions and computational exercises. The exercises range in difficulty and in scope. Each chapter is concise and pointed, introducing exactly one new concept to the reader. For this reason, the act of going through the exercises is important for completeness of understanding the material, especially for an undergraduate student.

In the first half of the book, Edwards revisits the basics of arithmetic by relating it to counting. He then proceeds to cover the traditional number theory topics, including: the Euclidean Algorithm, simultaneous congruence, the Chinese Remainder Theorem, the Fundamental Theorem of Algebra, and Euler's totient function. He describes how to compute the order of a number under modular arithmetic, a method for primality testing, and the RSA encryption algorithm. He also presents some non-traditional topics as extensions of those previously listed. The presentation of many of these ideas is unique in that he is working strictly with nonnegative integers. For example, Edwards describes an augmented Euclidean Algorithm that finds numbers, u, v, x, y of

$$\begin{aligned} d + ub &= va \\ d + xa &= yb \end{aligned}$$

where a and b are non-zero numbers and d is their greatest common divisor.

As each new topic is introduced, the author builds a formal definition, and thoroughly explains each proof. In particular, Chapter eight is well balance between the presentation of algorithms and proofs. The chapter begins by describing a practical algorithm for computing a^b by giving an example and explanation of the algorithm. Next, the author considers the problem of finding all the solutions to $a^b \equiv 1 \mod c$ with $a, c > 1$ and $b > 0$. He shows that the problem has a solution if and only if a is relatively prime to c. This chapter concludes by posing the problem of given two non-zero relatively prime numbers a and c find the order of $a \mod c$. In other words, what is the smallest integer k for which $a^k \equiv 1 \mod c$?

In the later chapters of the book, Edwards moves forward from the traditional introductory number theory topics to more advanced topics. Chapter 15 uses Jacobi's table of indices mod p

[12] This is not a typo or a weird LaTeX thing. The author uses $\Box$ to mean a square.

to find the solutions to $ax^2 + bx + c \equiv 0 \mod p$. In particular, he shows that if the index i of a number mod p is odd it does not have a square root mod p.

In chapter 19, the author derives all solutions to $A\Box + B = \Box$. A primitive solution to $A\Box + B = \Box$ is a solution to the problem $Ax^2 + B = y^2$ where x and y are relatively prime. The Comparison Algorithm, which determines if two modules are equivalnet, is used to find all the primitive solutions to $A\Box + B = \Box$. The last part of this chapter constructs all the solutions of this equation from the primitive solutions.

Next, he turns his attention to proving Euler's Law:

> The value of $C_p(A)$ depends only on the value of $p \mod 4A$.

He introduces the quadratic character of $A \mod p$ denoted $C_p(A)$, properties of stable modules, and the Law of Quadratic Reciprocity. In Chapter 25, Edwards presents The Main Theorem, which provides a criterion for when a is a square mod p. Then, Euler's Law is proven as a consequence of The Main Theorem. The book concludes with a brief discussion of the composition of binary quadratic forms.

3 Opinion

A review of a number theory book would be incomplete without comparing it to other texts that are used in introductory number theory classes. While other books, such as [2], are purely mathematical, and only investigate topics such as RSA encryption as an afterthought or an application at the end of the textbook, this book introduces RSA and other applications quite early. In fact, half way through the book, almost all topics in a typical elementary number theory class have been covered. Furthermore, he completely tackles the essence of a first semester number theory course from the perspective of solving the problem $A\Box + B = \Box$.

Other number theory books, such as [1], assume a more mathematically sophisticated audience. Although knowledge of abstract algebra would enhance the reader's understanding of the material, it is not a prerequisite for reading *Higher Arithmetic.*

Many universities use number theory as the first class that students are introduced to formal proofs. This book does not address the fact that some students may not be comfortable with the methods of proving theorems. At the same time, however, the proofs are generally well-written and easy to follow. This means that the students should be able to read and understand the proofs without having a formal introduction to proof techniques.

Higher Arithmetic assumes that the students are familiar with algorithms and writing programs. When he introduces a new algorithm, he begins by giving examples and descriptions of the algorithms. After the algorithm is formally introduced, a short discussion usually follows. Furthermore, efficiency improvements (such as removing repetitive computations) to the algorithm are also discussed. Many of the computational exercises are implementations of the algorithms. This gives the student further insights into the algorithm. This method makes a computer scientist very comfortable by describing the basics in terms of algorithms, and makes non-computer scientists comfortable by using examples to describe the algorithms. It is worth noting, however, that the book does not refer to Big O notation that is commonly used to describe the running time of algorithms.

Although the text is dense, Edwards strives to instill an intuitive feeling for the material. This is most prevalent in the exercises. Each chapter is accompanied by a set of exercises, including

both study/analytical questions and computational problems. In the beginning, these exercises are classified as either study or computational. As the book progresses, however, this distinction is no longer made. The author feels that written work is important to clarify new ideas. In his experience, previous students had enjoyed and benefited from the computational problems.

Many of the exercises focus on having the student implement algorithms or explain the concepts. One question asks the student to explain a result "to an eight year old." The problems often ask the student to be creative. For instance, in chapter two, the student is asked to describe how s/he thinks Archimedes came up with bounds for $\sqrt{3}$. Some of the results of the exercises are revisited or extended in later chapters. Although this makes the text, along with the exercises, more cohesive, this method results in the exercises having information in them that could be lost by the casual reader. For example, chapter five exercise eight gives a formal algorithm for solving $ax \equiv c \mod b$.

There are some nuances of the text that are worth noting. The use of capital letters A, B to represent one-dimensional variables is not standard. Thus, the equation $A\Box + B = \Box$ looks remarkably like solving a system of equations, which it is not. There were some other notational quirks of the book, including the use of the symbol $\Box$ for a square number, the use of $a \equiv b \mod [n, m]$ for double congruences, and the use of fractions in congruences: $\frac{1}{6} \equiv 2 \mod 11$ since $1 \equiv 6 \cdot 2 \mod 11$. Within the context of this book, however, these notations do seem natural. There are several awkward moments in the text, including the following problem from Chapter 21:

> Given a number A, not a square, for which primes p is A a square mod p?

It may take the reader a moment to realize that the question is asking: for what values of p is A congruent to a square number mod p?

Despite the issues described above, this book is a good introduction to number theory for a computer scientist. This book:

- links number theory with computer science.
- uses programming in the exercises.
- is well suited for self-study.
- provides plenty of examples and motivation.

We particularly liked how the focus of the book created a cohesive collection of theorems and examples. Everything was introduced for a reason.

In sum, we recommend this book for an introduction to number theory for a computer science or mathematics student familiar with algorithms, or for self-study for a computer scientist. Additionally, this book would be well suited for an independent study, because there are plenty of examples and exercises (with solutions) for the student to explore.

References

[1] G.H.Hardy, E.M.Wright, *An Introduction to the Theory of Numbers*, Oxford Science P, New York, 2003.

[2] J.K.Strayer, *Elementary Number Theory*, Waveland P, Prospect Heights, IL, 2002.