

Edmund K. Burke · Graham Kendall
Editors

Search Methodologies

Introductory Tutorials in Optimization
and Decision Support Techniques

Second Edition

Search Methodologies

Edmund K. Burke • Graham Kendall
Editors

Search Methodologies

Introductory Tutorials in Optimization and
Decision Support Techniques

Second Edition



Springer

Editors

Edmund K. Burke
University of Stirling
Cottrell Building
Scotland, United Kingdom

Graham Kendall
Vice-Provost (Research
and Knowledge Transfer)
University of Nottingham, Malaysia
and University of Nottingham, UK
Jalan Broga, Semenyih, Malaysia

ISBN 978-1-4614-6939-1

ISBN 978-1-4614-6940-7 (eBook)

DOI 10.1007/978-1-4614-6940-7

Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2013944229

© Springer Science+Business Media New York 2014

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword to the First Edition

This is not so much a foreword as a testimonial. As I embarked on the pleasant journey of reading through the chapters of this book, I became convinced that this is one of the best sources of introductory material on the search methodologies topic to be found. The book's subtitle, "Introductory Tutorials in Optimization and Decision Support Techniques," aptly describes its aim, and the editors and contributors to this volume have achieved this aim with remarkable success.

The chapters in this book are exemplary in giving useful guidelines for implementing the methods and frameworks described. They are tutorials in the way tutorials ought to be designed. I found no chapter that did not contain interesting and relevant information and found several chapters that can only be qualified as delightful.

Those of us who have devoted a substantial portion of our energies to the study and elaboration of search methodologies often wish we had a simple formula for passing along the core notions to newcomers to the area. (I must confess, by the way, that I qualify as a relative newcomer to some of the areas in this volume.) While simplicity, like beauty, to some degree lies in the eyes of the beholder and no universal or magical formula for achieving it exists, this book comes much closer to reaching such a goal than I would have previously considered possible. It will occupy a privileged position on my list of recommended reading for students and colleagues who want to get a taste for the basics of search methodologies and who have an interest in equipping themselves to probe more deeply.

If good books may be likened to ladders that help us ascend to higher rungs of knowledge and understanding, we all know there are nevertheless many books written in technical areas that seem to be more like stumbling blocks or at best broken stepping stools that deposit us in isolated nooks offering no clear access to continued means of ascent. Not so the present book. Its chapters lead to ideal sites for continued exploration and offer compelling motivation for further pursuit of its ideas and frameworks. If my reckoning is not completely amiss, those who read this volume will find abundant reasons for sharing my conviction that we owe its editors and authors a true debt of gratitude for putting this work together.

Boulder, CO, USA

Fred Glover

Foreword to the Second Edition

It gives me great pleasure to write this short foreword to the second edition of this outstanding book. The first edition established this volume as an important international landmark in the exposition and explanation of computational search methodologies and associated issues. As its title indicates, it covers a broad and diverse portfolio of search techniques from an interdisciplinary perspective in an extensive and comprehensive way. In some ways, this is no surprise because the authors that have been brought together in this volume truly represent a collection of the very top figures in the field. It is an extremely impressive group of authors who have the highest possible qualifications to present these chapters in a clear and authoritative way.

As the editors have explained in the Introduction, decision support systems now underpin the core of many modern management approaches in such diverse areas as health care, commerce, industry, science, and government. In many everyday situations across many different working environments, the idea of taking important managerial decisions without some kind of decision support system would be inconceivable. Moreover, it is the techniques and methods that are introduced in this book that often lie at the heart of the search engines upon which these decision support systems depend. Thus, the book ideally fits the current needs of today's increasingly computer-driven world.

The aim of the book is clearly articulated in the Introduction. It aims to present a series of well-written tutorials by the leading experts in their fields. Moreover, it does this by covering practically the whole possible range of topics in the discipline. It enables students and practitioners to study and appreciate the beauty and the power of some of the computational search techniques that are able to effectively navigate through search spaces that are sometimes inconceivably large.

I am convinced that this second edition will build on the success of the first edition and that it will prove to be just as popular. Three main features of the volume add significantly, in my opinion, to its appeal. These can be outlined as follows:

- I have already referred to the set of excellent authors who have written the chapters of the book, but I think that it is worth emphasizing how important this is. All the authors are world-renowned experts in their field.
- There is full and complete coverage of the area of search methodologies. This is complemented in the second edition by additional chapters on Scatter Search, GRASP (Greedy Randomized Adaptive Search Procedures), and Very Large-Scale Neighborhood Search. The choice of these additional chapters was inspired and reflects the ever-changing nature and developing focus of the field.
- I particularly appreciate the uniform approach to the description of topics which originate from such different disciplines as operations research, computer science, mathematics, artificial intelligence, and others. Establishing this common structure across such a large number of authors is testament to the skill and international standing of the editors. Their vision has been realized in an exceptional introductory treatment of the field. I think that tricks of trade, examples, sources of additional information, and the extensive list of appropriate references represent particularly useful resources, and I have regularly directed my own students to them in the first edition. I expect to continue to do so with this new edition. These features particularly add to the usefulness of the book either for self-study or as the basis of a course in the topic.

To summarize, I highly recommend the book for those who require an introduction to the breadth of techniques and approaches that are represented by the field of search methodologies. I have gained a lot of pleasure from reading through this new edition, and I am sure that it will be a valuable resource to teachers, students, and practitioners for many years to come.

Poznan, Poland

Jacek Blazewicz

Preface to the First Edition

We first had the idea for this book over 3 years ago. It grew out of a one-day workshop entitled *Introductory Tutorials in Search, Optimization and Decision Support Methodologies (INTROS)*, which was held in Nottingham in August 2003. The aim of the workshop was to deliver basic introductions to a broad spectrum of search methodologies from across disciplinary boundaries. It was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) and the London Mathematical Society (LMS) and was attended by over one hundred delegates from all over the world. We were very fortunate to have 11 of the world's leading scientists in search methodologies presenting a range of stimulating and highly informative tutorials. All of the INTROS presenters have contributed to this volume, and we have enhanced the content by inviting additional, specifically targeted, complementary chapters. We are pleased to be able to present such a comprehensive, multidisciplinary collection of tutorials in this crucially important research area.

We would like to take this opportunity to thank the many people who have contributed towards the preparation of this book. We owe a great debt of gratitude to the authors of the chapters. As one would expect from such a distinguished group of scientists, they have prepared their excellent contributions in a thoroughly reliable and professional manner. Without them, of course, the book would not have been possible. We are extremely grateful to our copy editor, Piers Maddox, who excelled himself in bringing together, in one coherent structure, the various documents that were sent to him. We are also very grateful to Gary Folven, Carolyn Ford, and their staff at Springer who have provided us with invaluable advice and support during every step of the way. We would like to offer our gratitude to Fred Glover for writing the foreword for this book. His warm praise is particularly pleasing. A special thank you should go to Emma-Jayne Dann and Alison Payne for all the administrative support they have given us, both in the preparation of this volume and in the organization of the INTROS workshop that underpinned it. We are also very thankful to EPSRC and LMS for the financial support they gave us to hold this workshop. Finally, we offer a special thank you to the INTROS delegates for their enthusiasm and their encouragement.

We hope you enjoy reading this volume as much as we have enjoyed putting it together. We are already planning a second edition, and if you have any comments which can help us improve the book, please do not hesitate to contact us. We would welcome your advice.

Nottingham, UK
Nottingham, UK

Edmund K. Burke
Graham Kendall

Preface to the Second Edition

The first edition of this book, which appeared in 2005, grew out of a 1-day workshop entitled Introductory Tutorials in Search, Optimization and Decision Support Methodologies (INTROS), which was held in Nottingham, UK, in August 2003. It had 19 chapters which gave broad coverage of the predominant search methodologies at the time.

This second edition contains updated versions of the chapters, and we have also included three additional chapters (Scatter Search, Chap. 5; GRASP: Greedy Randomized Adaptive Search Procedures, Chap. 11; and Very Large-Scale Neighborhood Search, Chap. 13). These new chapters, along with the revised chapters, we believe, provide significant insight to the most popular search methodologies that are in use today.

There are many people that we need to thank. Without their help and support, this book would not have been possible. We are grateful to Jacek Blazewicz for writing such a generous and positive foreword for this second edition. We are deeply indebted to all the authors for their contributions and for their patience about the time that it took to get this edition into press. The authors represent a distinguished group of scientists who have all prepared, and updated, excellent contributions in a conscientious and professional way. We are also grateful to our excellent copy editor, Piers Maddox. We recognize that we did not always provide him with the material in a form that he would have preferred. However, he dealt with everything very quickly and extremely diligently. We would also like to extend our thanks to the staff at Springer, particularly Matthew Amboy, who have supported the preparation of this book from the very beginning.

We hope you enjoy reading this book and gain as much from it as we gained in editing it.

Stirling, UK
Nottingham, UK

Edmund K. Burke
Graham Kendall

Contents

1	Introduction	1
	Edmund K. Burke and Graham Kendall	
2	Classical Techniques	19
	Kathryn A. Dowsland	
3	Integer Programming	67
	Robert Bosch and Michael Trick	
4	Genetic Algorithms	93
	Kumara Sastry, David E. Goldberg, and Graham Kendall	
5	Scatter Search	119
	Manuel Laguna	
6	Genetic Programming	143
	Riccardo Poli and John Koza	
7	Artificial Immune Systems	187
	Uwe Aickelin, Dipankar Dasgupta, and Feng Gu	
8	Swarm Intelligence	213
	Daniel Merkle and Martin Middendorf	
9	Tabu Search	243
	Michel Gendreau and Jean-Yves Potvin	
10	Simulated Annealing	265
	Emile Aarts, Jan Korst and Wil Michiels	
11	GRASP: Greedy Randomized Adaptive Search Procedures	287
	Mauricio G.C. Resende and Celso C. Ribeiro	

12	Variable Neighborhood Search	313
	Pierre Hansen and Nenad Mladenović	
13	Very Large-Scale Neighborhood Search	339
	Douglas S. Altner, Ravindra K. Ahuja, Özlem Ergun, and James B. Orlin	
14	Constraint Programming	369
	Eugene C. Freuder and Mark Wallace	
15	Multi-objective Optimization	403
	Kalyanmoy Deb	
16	Sharpened and Focused No Free Lunch and Complexity Theory	451
	Darrell Whitley	
17	Machine Learning	477
	Xin Yao and Yong Liu	
18	Fuzzy Reasoning	519
	Costas P. Pappis and Constantinos I. Siettos	
19	Rough-Set-Based Decision Support	557
	Roman Słowiński, Salvatore Greco, and Benedetto Matarazzo	
20	Hyper-heuristics	611
	Peter Ross	
21	Approximations and Randomization	639
	Carla P. Gomes and Ryan Williams	
22	Fitness Landscapes	681
	Colin R. Reeves	
Index		707

Chapter 1

Introduction

Edmund K. Burke and Graham Kendall

1.1 Inter-disciplinary Decision Support: Motivation

Search and optimization technologies underpin the development of decision support systems in a wide variety of applications across industry, commerce, science and government. There is a significant level of diversity among optimization and computational search applications. This can be evidenced by noting that a small selection of applications includes transport scheduling, bioinformatics optimization, personnel rostering, medical decision support and timetabling. Later in this introduction we present some recent survey papers for some of these areas and more examples of relevant applications are available in [Pardalos and Resende \(2002\)](#) and [Leung \(2004\)](#). The potential impact of more effective and efficient decision support methodologies is enormous and can be illustrated by considering just a few of the potential benefits:

- More efficient production scheduling can lead to significant financial savings;
- Higher-quality personnel rosters lead to a more contented workforce;
- Efficient healthcare scheduling will lead to faster treatment (potentially saving lives);
- More effective cutting/packing systems can reduce waste;
- Better delivery schedules can reduce fuel emissions.

This research area has received significant attention from the scientific community across many different academic disciplines. Looking at any selection of key papers

E.K. Burke (✉)

Computational Heuristics, Operational Research and Decision Support Group, Division of Computing and Mathematics, University of Stirling, Stirling, Scotland, UK

e-mail: e.k.burke@stir.ac.uk

G. Kendall

Automated Scheduling, Optimization and Planning Research Group, School of Computer Science, University of Nottingham, Jubilee Campus, Nottinghamshire, UK

Automated Scheduling, Optimization and Planning Research Group, School of Computer Science, University of Nottingham, Malaysia Campus, Jalan Broga, Semenyih, Malaysia

which have impacted upon search, optimization and decision support will reveal that the authors are based in a number of different departments including Computer Science, Mathematics, Engineering, Business and Management (among others). It is clearly the case that the investigation and development of decision support methodologies is inherently multi-disciplinary. It lies firmly at the interface of Operational Research, Computer Science and Artificial Intelligence (among other disciplines). However, not only is the underlying methodology inherently inter-disciplinary but the broad range of application areas also cuts across many disciplines and industries. We firmly believe that scientific progress in this crucially important area will be made far more effectively and far more quickly by adopting a broad and inclusive multi-disciplinary approach to the international scientific agenda in this field.

This observation provides one of the key motivations for this book, which is aimed primarily at first-year postgraduate students and final-year undergraduate students. However, we have also aimed it at practitioners and at the experienced researcher who wants a brief introduction to the broad range of decision support methodologies that are in the literature. In our experience, the key texts for these methodologies lie across a variety of volumes. This reflects the wide range of disciplines that are represented here. We wanted to bring together a series of entry-level tutorials, written by world-leading scientists from across the disciplinary range, in one single volume.

1.2 The Structure of the Book

The first edition of this book was initially motivated by the idea of being able to present first-year PhD students with a single volume that would give them a basic introduction to the various search and optimization techniques that they might require during their program of research. This remains a primary motivation for this second edition.

The book can be read in a sequential manner. However, each chapter also stands alone and so the book can be dipped into when you come across a technique with which you are not familiar, or if you just need to find some general references on a particular topic.

If you want to read the book all the way through, we hope that the way we have ordered the chapters makes sense. In Chaps. 2 and 3 we introduce some classical search and optimization techniques which, although not always suitable (particularly when your problem has a very large search space), are still important to have in your “tool box” of methodologies. Indeed, recent (highly effective) approaches have hybridized such methods with some of the other techniques in this book. Many of the other chapters introduce various search and optimization techniques, some of which have been used for over 30 years (e.g. genetic algorithms, Chap. 4) and some which are relatively new (e.g. artificial immune systems, Chap. 7). Some of the chapters consider more theoretical aspects of search and optimization. The chapter by Darrel Whitley, for example, introduces *Sharpened and Focused No Free Lunch and Complexity Theory* (Chap. 16) whilst Colin Reeves considers *Fitness Landscapes* in Chap. 22.

One element of every chapter is a section called *Tricks of the Trade*. We recognize that it is sometimes difficult to know where to start when you first come across a new problem. Which technique or methodology is the most appropriate? This is a *very* difficult question to answer and forms the basis of much research in the area. *Tricks of the Trade* is designed to give you some guidelines on how you should get started and what you should do if you run into problems. Although *Tricks of the Trade* is towards the end of each chapter, we believe that it could be one of the first sections you read.

We have also asked authors to include a section entitled *Sources of Additional Information*. These sections are designed as useful pointers to resources such as books and web pages. They are intended as the next place to investigate, once you have read the chapter.

As this book is aimed primarily at the beginner (such as a first-year PhD student, final-year undergraduate or practitioner/researcher learning a new technique) we thought it might be useful to explain some basic concepts which many books just assume that the reader already knows. Indeed, our own PhD students and final-year undergraduates often make this complaint. We realize that the following list is not complete. Nor can it ever be, as we are not aiming to write a comprehensive encyclopedia. If you feel that any important terms are missing, please let the editors (authors of this introduction) know and we will consider including them in future editions. All of these concepts are, purposefully, explained in an informal way so that we can get the basic ideas across to the reader. More formal definitions can be found elsewhere (see the *Sources of Additional Information* and *References*), including in later chapters of this book.

1.3 Basic Concepts and Underlying Issues

In this section we present a number of basic terms and issues and offer a simple description or explanation. In explaining the concepts to beginners, we will restrict the formal presentation of these concepts as much as possible.

1.3.1 Artificial Intelligence

Artificial intelligence (AI) is a broad term which can be thought of as covering the goal of developing computer systems which can solve problems which are usually associated with requiring human-level intelligence. There are a number of different definitions of the term and there has been a significant amount of debate about it. However, the philosophical arguments about what is or is not AI do not fall within the remit of this book. The interested reader is directed to the following (small) sample of general AI books: Negnevitsky (2005), Russell and Norvig (2009), Callan (2003), Luger (2005), McCarthy (1996), Cawsey (1998), Rich and Knight (1991), and Nilsson (1998).

1.3.2 Operational Research (Operations Research)

These two terms are completely interchangeable and are often abbreviated to OR. Different countries tend to use one or other of the terms but there is no significant difference. The field was established in the 1930s and early 1940s as scientists in Britain became involved in the *operational* activities of Britain's radar stations. After the war, the field expanded into applications within industry, commerce and government and spread throughout the world. Gass and Harris, in the preface to their excellent *Encyclopedia of Operations Research and Management Science* ([Gass and Harris 2001](#)), present several definitions. However, as with Artificial Intelligence, we are not really concerned with the intricacies of different definitions in this book. The first definition they give says, “Operations Research is the application of the methods of science to complex problems arising in the direction and management of large systems of men, machines, materials and money in industry, business, government and defense”. This presents a reasonable summary of what the term means. For more discussion, and a range of definitions, on the topic, see [Bronson and Naadimuthu \(1997\)](#), [Carter and Price \(2001\)](#), [Hillier and Liberman \(2010\)](#), [Taha \(2010\)](#), [Urry \(1991\)](#), and [Winston \(2004\)](#). For an excellent and fascinating early history of the field see [Kirby \(2003\)](#).

1.3.3 Management Science

This term is sometimes abbreviated to MS and it can, to all intents and purposes, be interchanged with OR. Definitions can be found in [Gass and Harris \(2001\)](#). However, they sum up the use of these terms in their preface when they say, “Together, OR and MS may be thought of as the science of operational processes, decision making and management.”

1.3.4 Feasible and Infeasible Solutions

The idea of feasible and infeasible solutions is intuitive but let us consider the specific problem of cutting and packing, so that we have a concrete example which can be related to. This problem arises in many industries: for example, in the textile industry where pieces for garments have to be cut from rolls of material, in the newspaper industry where the various text and pictures have to be laid out on the page and in the metal industry where metal shapes have to be cut from larger pieces of metal. Of course, all these industries are different but let us consider a generic problem where we have to place a number of pieces onto a larger piece so that the smaller pieces can be cut out. Given this generic problem, a feasible solution can be thought of as all the shapes being placed onto the larger sheet so that none of them overlap and all the pieces lie within the confines of the larger sheet. If some of

the pieces overlap each other or do not fit onto the larger sheet, then the solution is infeasible. Of course, the problem definition is important when considering whether or not a given solution is feasible. For example, we could relax the constraint that says that *all* of the shapes have to be placed on the larger sheet, as our problem might state that we are trying to cut out as many of the smaller shapes as possible, but that it is not critical to include all the smaller pieces. A feasible solution is often defined as one that satisfies the *hard constraints* (see below).

1.3.5 Hard Constraints

For any given problem, there are usually constraints (conditions) that *have* to be satisfied. These are often called *hard constraints*. To continue with the cutting and packing example from above, the condition that no pieces can overlap is an example of a hard constraint. To take a new example, if we consider an employee rostering problem, then an example of a hard constraint is the condition that no employee can be allocated to two different shifts at the same time. If we violate a hard constraint, it leads to an *infeasible* solution.

1.3.6 Soft Constraints and Evaluation Functions

A *soft constraint* is a condition that we would like to satisfy but which is not absolutely essential. To elaborate on the employee scheduling example, we may have a soft constraint that says that we would like employees to be able to express preferences about which shifts they would like to work. However, if this constraint is not fully met, a solution is still feasible. It just means that another solution which does meet the condition more (i.e. more employees have their working preferences met) would be of higher quality. Of course, there could be many competing soft constraints, which may provide a trade-off in the *evaluation function* (measure of the quality of the solution which is also sometimes known as the *objective*, *fitness* or *penalty* function), as the improvement of one soft constraint may cause other soft constraint(s) to become worse. This is the situation where a multi-objective approach might be applicable (see Chap. 15).

Many problems have an evaluation function represented by a summation of the penalty values for the soft constraints. Some problems simply ignore the hard constraints in the evaluation function and just disregard infeasible solutions. Another approach is to set a penalty value for the hard constraints, but to set it very high so that any solution which violates the hard constraints is given a very high penalty value. Of course, the goal here would be to *minimize* the overall penalty value but it is often the case that some search problems try to *maximize* an evaluation function. A further possibility is to have dynamic penalties so that, at the start of a (minimization) search, the hard constraints are given relatively low penalty values, so that the

infeasible search space is explored. As the search progresses, the hard-constraint penalty values are gradually raised so that the search eventually only searches the feasible regions of the search space.

1.3.7 Deterministic Search

This term refers to a search method or algorithm which always returns the same answer, given exactly the same input and starting conditions. Several of the methods presented in this book are not deterministic, i.e. there is an element of randomness in the approach so that different runs on exactly the same starting conditions can produce different solutions. Note, however, that the term *non-deterministic* can mean something more than simply not being deterministic. See Chap. 16 for an explanation.

1.3.8 Optimization

Within the context of this book, optimization can be thought of as the process of attempting to find the best possible solution amongst all those available. Therefore, the task of optimization is to model your problem in terms of some evaluation function (which represents the quality of a given solution) and then employ a search algorithm to minimize (or maximize, depending on the problem) that objective function. Most of the chapters in this book describe methodologies which aim to optimize some function. However, most of the problems are so large that it is impossible to guarantee that the solution obtained is optimal. The term optimization can lead to confusion because it is sometimes also used to describe a process which returns the guaranteed optimal solution (which is, of course, subtly different from the process which just aims to find the best solution possible).

1.3.9 Local and Global Optimum

Figure 1.1 illustrates the difference between a local and global optimum. A local optimum is a point in the search space where all neighboring solutions are worse than the current solution. In Fig. 1.1, there are four local optima. A global optimum is a point in the search space where *all* other points in the search space are worse than (or equal to) the current one. Of course, in Fig. 1.1, we are considering a *maximization* problem.

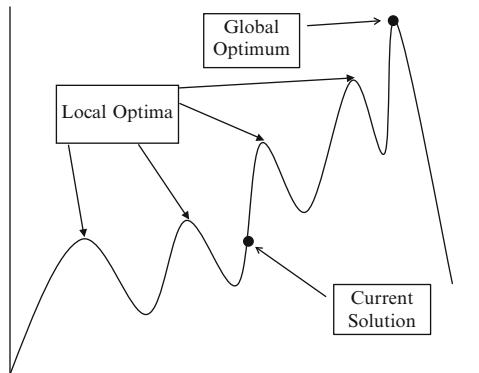


Fig. 1.1 An illustration of local optima and a global optimum

1.3.10 Exhaustive Search

By carrying out an exhaustive search, every possible solution is considered and the optimal (best) one is returned. For small problems, this is an acceptable strategy, but as problems become larger it becomes impossible to carry out such a search. The types of problem that often occur in the real world tend to grow very large very quickly and, of course, we can invent problems which are also too large to allow us to carry out an exhaustive search in a reasonable time. We will illustrate how problem sizes rise dramatically by considering a very well known problem: the traveling salesman problem (often referred to as TSP). This can be thought of as the problem of attempting to minimize the distance taken by a traveling salesman who has to visit a certain number of cities exactly once and return home. See [Johnson and McGeoch \(1997\)](#), [Lawler et al. \(1985\)](#) or [Laporte \(2010\)](#) for more details about the TSP. With a very small number of cities, the number of possible solutions is relatively small and an algorithm can easily check all possibilities (the search space) and return the best one. For example, a problem with five cities has a search space of size 12, so all 12 possibilities can be very easily checked. However, for a 50-city problem (just 10 times the number of cities), the number of solutions rises to about 10^{60} . [Michaelwicz and Fogel \(2004\)](#), in their excellent book on modern heuristics, consider exactly this 50-city problem. They say, “There are only 1,000,000,000,000,000,000 [10^{20}] liters of water on the planet so a 50-city TSP has an unimaginably large search space. Literally, it’s so large that as humans, we simply can’t conceive of sets with this many elements.”

Therefore, for large problems (and large does not have to be that large when considering the inputs), an exhaustive search is simply not an option. However, even if it is a possibility (i.e. the search space is small enough to allow us to carry out an exhaustive search) we must know how to systematically navigate the search space. This is not always possible.

1.3.11 Complexity

This term refers to the study of how difficult search and optimization problems are to solve. It is covered in Chap. 16.

1.3.12 Order (Big O Notation)

This term and associated notation is used in various places in this book and so we define it here. Suppose we have two functions $f(x)$ and $g(x)$ where x is a variable. We say that $g(x)$ is of the order of $f(x)$ written $g(x) = O(f(x))$ if, for some constant value K , $g(x) \leq Kf(x)$ for all values of x which are greater than K . This notation is often used when discussing the time complexity of search algorithms. In a certain sense, $f(x)$ bounds $g(x)$ once the values of x get beyond the value of K .

1.3.13 Heuristics

When faced with the kind of problem discussed in the exhaustive search section above, we have to accept that we need to develop an approach to obtain high-quality solutions—but optimality cannot be guaranteed (without *checking out* all the possibilities). Such an approach is called a heuristic. The following two definitions are often useful:

A heuristic technique (or simply heuristic) is a method which seeks good (i.e. near-optimal) solutions at a reasonable computation cost without being able to guarantee optimality, and possibly not feasibility. Unfortunately, it may not even be possible to state how close to optimality a particular heuristic solution is (Reeves 1996).

A “rule of thumb” based on domain knowledge from a particular application, that gives guidance in the solution of a problem ... Heuristics may thus be very valuable most of the time but their results or performance cannot be guaranteed (Oxford Dictionary of Computing 1996).

There are many heuristic methods available to us. Some examples are Simulated Annealing (Chap. 10), Genetic Algorithms (Chap. 4), Genetic Programming (Chap. 6) and Tabu Search (Chap. 9). The term *approximate* is sometimes used in connection with heuristic methods but it is important not to confuse it with approximation methods (see Chap. 21)

1.3.14 Constructive Heuristics

Constructive heuristics refer to the process of building an initial solution from scratch. Take university examination timetabling as an example. One way to generate a solution is to start with an empty timetable and gradually schedule examinations until they are all timetabled. The order in which the examinations are placed

onto the timetable is often important. Examinations which are more difficult to schedule (as determined by a heuristic measure of difficulty) are often scheduled first in the hope that the *easier* examinations can *fit around* the difficult ones.

Constructive heuristics are usually thought of as being fast because they often represent a single-pass approach.

1.3.15 Local Search Heuristics

Local search can be thought of as a heuristic mechanism where we consider *neighbors* of the current solution as potential replacements. If we accept a new solution from this neighborhood, then we *move* to that solution and then consider its neighbors—see *Hill Climbing* (below) for some initial discussion of this point. What is meant by *neighbor* is dependent upon the problem-solving situation that is being addressed. Some of the techniques described in this book can be described as local search methods: for example, simulated annealing (Chap. 10) and tabu search (Chap. 9). Hill climbing is also a local search method. For more information about local search see [Aarts and Lenstra \(2003\)](#). Note the difference between a constructive heuristic which builds a solution from scratch and a local search heuristic which moves from one solution to another. It is often the case that a constructive heuristic is used to generate a solution which is employed as the starting point for local search.

1.3.16 Hill Climbing

Hill climbing is probably the most basic local search algorithm. It is easy to understand and implement but suffers from getting stuck at a local optimum. In the following discussion, we will assume that we are trying to maximize a certain value. Of course, minimizing a certain value is an analogous problem, but then we would be *descending* rather than *climbing*.

The idea behind hill climbing is to take the current solution and generate a neighbor solution (see local search) and move to that solution only if it has a higher value of the evaluation function. The algorithm terminates when we cannot find a better-quality solution. The problem with hill climbing is that it can easily get stuck in a local optimum (see above). Consider Fig. 1.1.

If the current solution is the one as shown in Fig. 1.1, then hill climbing will only be able to find one of the local optima shown (the one directly above it in this case). At that point, there will be no other better solutions in its neighborhood and the algorithm will terminate.

Both simulated annealing (Chap. 10) and tabu search (Chap. 9) are variations of hill climbing but they incorporate a mechanism to help the search escape from local optima.

1.3.17 Metaheuristics

This term refers to a certain class of heuristic methods. Fred Glover first used it and he defines it as follows ([Glover and Laguna 1997](#)):

A meta-heuristic refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality. The heuristics guided by such a meta-strategy may be high level procedures or may embody nothing more than a description of available moves for transforming one solution into another, together with an associated evaluation rule.

[Osman and Kelly \(1996\)](#) offer the following definition:

A meta-heuristic is an iterative generation process which guides a subordinate heuristic ...

The study and development of metaheuristics has become an extremely important area of research into search methodologies. In common usage, in the literature, the term tends to be used to refer to the broad collection of relatively *sophisticated* heuristic methods that include Simulated Annealing, Tabu Search, Genetic Algorithms, Ant Colony methods and others (all of which are discussed in detail in this book). The term is employed sometimes with and sometimes without the hyphen in the literature. For more information about metaheuristics, see [Glover and Kochenberger \(2003\)](#), [Osman and Kelly \(1996\)](#), [Voss et al. \(1999\)](#), [Ribeiro and Hansen \(2002\)](#), [Resende and de Sousa \(2004\)](#), [Gendreau and Potvin \(2010\)](#), and [Rayward-Smith et al. \(1996\)](#).

1.3.18 Evolutionary Methods

Evolutionary methods can be thought of as representing a subset of the metaheuristic approaches and are typified by the fact that they maintain a *population* of candidate solutions and that these solutions compete for survival. Such approaches are inspired by evolution in nature.

Some of the methods in this book are evolutionary. Chapter 4 (Genetic Algorithms) represents perhaps the best known evolutionary approach but there are many others including Genetic Programming (Chap. 6). In the scientific literature, many metaheuristics are hybridized with other approaches (see, for example, [Blum et al. 2011](#)).

1.3.19 Exact Methods

This term is sometimes used to describe methods which can produce a solution that is guaranteed to be optimal (or which can show that no feasible solution exists).

1.3.20 Hyper-heuristics

Hyper-heuristics can be confused with metaheuristics but the distinction between the two terms is simple. Hyper-heuristics are simply methods which search through a search space of heuristics (or search methods). They can be defined as *heuristics to choose heuristics* or *heuristics which generate heuristics*. Most implementations of metaheuristics explore a search space of solutions to a given problem but they can be (and sometimes are) employed as hyper-heuristics. The term hyper-heuristic only tells you that we are operating on a search space of heuristics. It tells you nothing else. We may be employing a metaheuristic to do this search and we may not. The actual search space being explored may include metaheuristics and it may not (but very little work has actually been done which includes metaheuristics among the search space being addressed). Chapter 21 describes hyper-heuristics in more detail and readers are also referred to [Burke et al. \(2003, 2010, 2013\)](#).

1.3.21 Matheuristics

This term refers to methods that hybridize metaheuristics with mathematical programming techniques. See [Maniezzo et al. \(2010\)](#) and [Jourdan et al. \(2009\)](#) for more details.

Sources of Additional Information

The goal of this book is to present clear basic introductions to a wide variety of search methodologies from across disciplinary boundaries. However, it will be the case that many readers will be interested in a specific application or problem domain. With this in mind, we provide a sample of domains, with a small selection of references to survey and overview papers which might be of interest. Of course, the list is far from exhaustive but its purpose is to point the interested reader to overview papers for a small selection of well studied problem areas.

- Cutting and Packing: [Bennell and Oliveira \(2008, 2009\)](#), [Dyckhoff \(1990\)](#), [Dowsland and Dowsland \(1992\)](#), and [Wäscher et al. \(2007\)](#).
- Employee Scheduling: [Burke et al. \(2004\)](#), [Ernst et al. \(2004\)](#), [Gopalakrishnan and Johnson \(2005\)](#), and [Kwan \(2004\)](#).
- Educational Timetabling: [Burke and Petrovic \(2002\)](#), [Lewis \(2008\)](#), [Qu et al. \(2009\)](#), [Petrovic and Burke \(2004\)](#), and [Schaerf \(1999\)](#).
- Healthcare Scheduling and Optimization: [Cardoen et al. \(2010\)](#) and [Rais and Viana \(2011\)](#).
- Printed Circuit Board Assembly: [Ayob and Kendall \(2008, 2009\)](#).
- Air Transport Scheduling: [Qi et al. \(2004\)](#) and [Gopalakrishnan and Johnson \(2005\)](#).

- Sports Scheduling: [Dinitz et al. \(2007\)](#), [Drexel and Knust \(2007\)](#), [Easton et al. \(2004\)](#), [Kendall et al. \(2010\)](#), [Rasmussen and Trick \(2008\)](#), and [Wright \(2009\)](#).
- Traveling Salesman Problem: [Laporte 2010](#), [Johnson and McGeoch \(1997\)](#), and [Lawler et al. \(1985\)](#).
- Vehicle Routing: [Laporte \(2009\)](#), [Potvin \(2009\)](#), [Marinakis and Migdalas \(2007\)](#), and [Bräysy and Gendreau \(2005a,b\)](#).

In this section we will also provide a list of journals (in alphabetical order) across a range of disciplines that regularly publish papers upon aspects of decision support methodologies. This list is certainly not exhaustive. However, it provides a starting point for the new researcher and that is the sole purpose of presenting it here. We have purposefully not provided URL links to the journals as many will change after going to press, but a search for a journal's title will quickly locate its home page.

- ACM Journal of Experimental Algorithms
- Annals of Operations Research
- Applied Artificial Intelligence
- Applied Intelligence
- Applied Soft Computing
- Artificial Intelligence
- Artificial Life
- Asia-Pacific Journal of Operational Research
- Central European Journal of Operations Research
- Computational Intelligence
- Computational Optimization and Applications
- Computer Journal
- Computers & Industrial Engineering
- Computers & Operations Research
- Decision Support Systems
- Engineering Optimization
- European Journal of Information Systems
- European Journal of Operational Research
- Evolutionary Computation
- 4OR—A Quarterly Journal of Operations Research
- Fuzzy Sets And Systems
- Genetic Programming and Evolvable Machines
- IEEE Transactions on Computers
- IEEE Transactions on Evolutionary Computation
- IEEE Transactions on Fuzzy Systems
- IEEE Transactions on Neural Networks
- IEEE Transactions on Systems Man And Cybernetics Part A—Systems And Humans
- IEEE Transactions on Systems Man And Cybernetics Part B—Cybernetics
- IEEE Transactions On Systems Man And Cybernetics Part C—Applications And Review

- IIE Transactions
- INFOR
- INFORMS Journal on Computing
- Interfaces
- International Journal of Systems Science
- International Transactions on Operational Research
- Journal of Artificial Intelligence Research
- Journal of Global Optimization
- Journal of Heuristics
- Journal of Optimization Theory And Applications
- Journal of Scheduling
- Journal of The ACM
- Journal of The Operational Research Society
- Journal of The Operational Research Society of Japan
- Knowledge-Based Systems
- Machine Learning
- Management Science
- Mathematical Methods of Operations Research
- Mathematics of Operations Research
- Mathematical Programming
- Naval Research Logistics
- Neural Computation
- Neural Computing & Applications
- Neural Networks
- Neurocomputing
- Omega—International Journal of Management Science
- Operations Research
- Operations Research Letters
- OR Spectrum
- RAIRO—Operations Research
- SIAM Journal on Computing
- SIAM Journal on Optimization
- Soft Computing
- Transportation Research
- Transportation Science

This bibliography presents a selection of volumes and papers which give an overview of search and optimization methodologies and some well studied search/optimization problems. More detailed bibliographies and sources of additional information are presented throughout the book.

References

- Aarts E, Lenstra JK (eds) (2003) Local search in combinatorial optimization. Princeton University Press, Princeton, New Jersey, USA (first published by Wiley 1997)
- Ayob M, Kendall G (2008) A survey of surface mount device placement machine optimisation: machine classification. *Eur J Oper Res* 186:893–914
- Ayob M, Kendall G (2009) The optimisation of the single surface mount device placement machine in printed circuit board assembly: a survey. *Int J Syst Sci* 40:553–569
- Bennell JA, Oliveira JF (2008) A tutorial in nesting problem: the geometry. *Eur J Oper Res* 184:397–415
- Bennell JA, Oliveira JF (2009) A tutorial in irregular shape packing problems. *J Oper Res Soc* 60:S93–S105
- Blum C, Puchinger J, Raidl G, Roli A (2011) Hybrid metaheuristics in combinatorial optimization: a survey. *Appl Soft Comput* 11:4135–4151
- Bräysy O, Gendreau M (2005a) Vehicle routing problem with time windows, part I: route construction and local search algorithms. *Transp Sci* 39:104–118
- Bräysy O, Gendreau M (2005b) Vehicle routing problem with time windows, part II: metaheuristics. *Transp Sci* 39:119–139
- Bronson R, Naadimuthu G (1997) Operations research, Schaum's outlines, 2nd edn. McGraw-Hill, New York
- Burke EK, Petrovic S (2002) Recent research directions in automated timetabling. *Eur J Oper Res* 140:266–280
- Burke EK, Kendall G, Newall JP, Hart E, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. In: Glover F, Kochenberger G (eds) *Handbook of metaheuristics*, chap 16. Kluwer, Dordrecht, pp 457–474
- Burke EK, De Causmaecker P, Vanden Berghe G, Van Landeghem R (2004) The state of the art of nurse rostering. *J Sched* 7:441–499
- Burke EK, Hyde M, Kendall G, Ochoa G, Ozcan E, Woodward JRA (2010) A Classification of hyper-heuristic approaches. In: *Handbook of metaheuristics*. Kluwer, Dordrecht, pp 449–468
- Burke EK, Gendreau M, Hyde M, Kendall G, Ochoa G, Ozcan E, QUR (2013) Hyper-heuristics: a survey of the state of the art. *J Oper Res Soc*, doi:10.1057/jors.2013.71
- Callan R (2003) Artificial intelligence. Palgrave Macmillan, London
- Cardoen B, Demeulemeester E, Beliën J (2010) Operating room planning and scheduling: a literature review. *Eur J Oper Res* 201:921–932
- Carter MW, Price CC (2001) Operations research: a practical introduction. CRC, Boca Raton
- Cawsey A (1998) The essence of artificial intelligence. Prentice-Hall, Englewood Cliffs

- Dinitz JH, Fronček D, Lamken ER, Wallis WD (2007) Scheduling a tournament. In: Colbourn CJ, Dinitz JH (eds) *Handbook of combinatorial designs*, 2nd edn. CRC, Boca Raton, pp 591–606
- Dowsland KA, Dowsland WB (1992) Packing problems. *Eur J Oper Res* 56:2–14
- Drexl A, Knust S (2007) Sports league scheduling: graph- and resource-based models. *Omega* 35:465–471
- Dyckhoff H (1990) A typology of cutting and packing problems. *Eur J Oper Res* 44:145–159
- Easton K, Nemhauser GL, Trick MA (2004) Sports scheduling. In: Leung JT (ed) *Handbook of scheduling*. CRC, Boca Raton, 52.1–52.19
- Ernst AT, Jiang H, Krishnamoorthy M, Owens B, Sier D (2004) An annotated bibliography of personnel scheduling and rostering. *Ann Oper Res* 127:21–144
- Gass SI, Harris CM (2001) *Encyclopaedia of operations research and management science*. Kluwer, Dordrecht
- Gendreau M, Potvin J-Y (eds) (2010) *Handbook of metaheuristics*, 2nd edn. Springer, Berlin
- Glover F, Kochenberger G (eds) (2003) *Handbook of metaheuristics*. Kluwer, Dordrecht
- Glover F, Laguna M (1997) *Tabu search*. Kluwer, Dordrecht
- Gopalakrishnan B, Johnson EL (2005) Airline crew scheduling: state-of-the-art. *Ann Oper Res* 140:305–337
- Hillier FS, Liberman GJ (2010) *Introduction to operations research*, 9th edn. McGraw-Hill, New York
- Johnson DS, McGeoch LA (1997) The travelling salesman problem: a case study. In: Aarts E, Lenstra JK (eds) (2003) *Local search in combinatorial optimization*. Princeton University Press, Princeton, New Jersey, USA, pp 215–310
- Jourdan L, Basseur M, Talbi E-G (2009) Hybridizing exact methods and metaheuristics: a taxonomy. *Eur J Oper Res* 199:620–629
- Kendall G, Knust S, Ribeiro CC, Urrutia S (2010) Scheduling in sports: an annotated bibliography. *Comput Oper Res* 37:1–19
- Kirby MW (2003) *Operational research in war and peace: the British experience from the 1930s to 1970*. Imperial College Press, London
- Kwan R (2004) Bus and train driver scheduling. In: Leung JY-T (ed) *Handbook of scheduling*, chap 51. Chapman and Hall/CRC, Boca Raton
- Laporte G (2009) Fifty years of vehicle routing. *Transp Sci* 43:408–416
- Laporte G (2010) A concise guide to the traveling salesman problem. *J Oper Res Soc* 61:35–40
- Lawler EL, Lenstra JK, Rinnooy Kan AHG, Shmoys DB (eds) (1985) *The travelling salesman problem: a guided tour of combinatorial optimization*. Wiley, New York (reprinted with subject index 1990)
- Leung JY-T (ed) (2004) *Handbook of scheduling*. Chapman and Hall/CRC, Boca Raton
- Lewis R (2008) A survey of metaheuristic-based techniques for university timetabling problems. *OR Spectr* 30:167–190

- Luger GFA (2005) Artificial intelligence: structures and strategies for complex problem solving, 5th edn. Addison-Wesley, New York
- Maniezzo V, Stützle T, Voss S (eds) (2010) Matheuristics. Springer, Berlin
- Marinakis Y, Migdalas A (2007) Annotated bibliography in vehicle routing. *Oper Res* 7:27–46
- McCarthy J (1996) Defending AI research: a collection of essays and reviews. CSLI Publications, Stanford
- Michalewicz Z, Fogel DB (2004) How to solve it: modern heuristics, 2nd edn. Springer, Berlin
- Negnevitsky M (2005) Artificial intelligence: a Guide to intelligent systems, 2nd edn. Addison-Wesley, New York
- Nilsson, N (1998) Artificial intelligence: a new synthesis. Morgan Kaufmann, San Mateo
- Osman IH, Kelly JP (eds) (1996) Metaheuristics: theory and applications. Kluwer, Dordrecht
- Oxford Dictionary of Computing (1996) Oxford dictionary of computing, 4th edn. Oxford University Press, Oxford
- Pardalos PM, Resende MGC (eds) (2002) Handbook of applied optimization. Oxford University Press, Oxford
- Petrovic S, Burke EK (2004) University timetabling. In: Leung JY-T (ed) (2004) Handbook of scheduling, chap 45. Chapman and Hall/CRC, Boca Raton
- Potvin J-Y (2009) Evolutionary algorithms for vehicle routing. *INFORMS J Comput* 21:518–548
- Qi X, Yang J, Yu G (2004) Scheduling problems in the airline industry. In: Leung JY-T (ed) Handbook of scheduling, chap 51. Chapman and Hall/CRC, Boca Raton
- Qu R, Burke EK, McCollum B, Merlot LGT, Lee SY (2009) A survey of search methodologies and automated system development for examination timetabling. *J Sched* 12:55–89
- Rais A, Viana A (2011) Operations research in healthcare: a survey. *Int Trans Oper Res* 18:1–31
- Rasmussen RV, Trick MA (2008) Round robin scheduling—a survey. *Eur J Oper Res* 188:617–636
- Rayward-Smith VJ, Osman IH, Reeves CR, Smith GD (1996) Modern heuristic search methods. Wiley, New York
- Reeves CR (1996) Modern heuristic techniques. In: Rayward-Smith VJ, Osman IH, Reeves CR, Smith GD (eds) Modern heuristic search methods. Wiley, New York, pp 1–25
- Resende MGC, de Sousa JP (eds) (2004) Metaheuristics: computer decision making. Kluwer, Dordrecht
- Ribeiro CC, Hansen P (eds) (2002) Essays and surveys in metaheuristics. Kluwer, Dordrecht
- Rich E, Knight K (1991) Artificial intelligence, 2nd edn. McGraw-Hill, New York
- Russell S, Norvig P (2009) Artificial intelligence: a modern approach, 3rd edn. Prentice-Hall, Englewood Cliffs
- Schaerf A (1999) A survey of automated timetabling. *Artif Intell Rev* 13:87–127

- Taha HA (2010) Operations research: an introduction, 9th edn. Prentice-Hall, Englewood Cliffs
- Urry S (1991) An introduction to operational research: the best of everything. Longmans, London
- Voss S, Martello S, Osman IH, Roucairol C (eds) (1999) Meta-heuristics: advances and trends in local search paradigms for optimization. Kluwer, Dordrecht
- Wäscher G, Haußner H, Schumann H (2007) An improved typology of cutting and packing problems. *Eur J Oper Res* 183:1109–1130
- Winston WL (2004) Operations research: applications and algorithms, 4th edn. Duxbury, Pacific Grove
- Wright MB (2009) Fifty years of OR in sport. *J Oper Res Soc* 60:S161–S168

Chapter 2

Classical Techniques

Kathryn A. Dowsland

2.1 Introduction

The purpose of this chapter is to provide an introduction to three classical search techniques—branch and bound, dynamic programming and network flow programming—all of which have a well established record in the solution of both classical and practical problems. All three have their origins in, or prior to, the 1950s and were the result of a surge in interest in the use of mathematical techniques for the solution of practical problems. The timing was in part due to developments in Operations Research in World War II, but was also spurred by increasing competition in the industrial sector and the promise of readily accessible computing power in the foreseeable future. A fourth technique belonging to this class, that of Integer Programming, is covered in Chap. 3. Given their age, it is not surprising that they no longer generate the same level of excitement as the more modern approaches covered elsewhere in this volume, and as a result they are frequently overlooked. This effect is reinforced as many texts such as this omit them—presumably because they have already been covered by a range of sources aimed at a wide variety of different abilities and backgrounds. In this volume we provide an introduction to these well-established classics alongside their more modern counterparts. Although they have shortcomings, many of which the more recent approaches were designed to address, they still have a role to play both as stand-alone techniques and as important ingredients in hybridized solution methods.

The chapter is meant for beginners and it is possible to understand and use the techniques covered without any prerequisite knowledge. However, some of the examples in the chapter are based on problems in graph theory. In all cases the problems and specialist terms are defined in full, but a basic knowledge of graph theory terminology such as that provided in Balakrishnan (1997) would be useful. Some of

K.A. Dowsland (✉)
Gower Optimal Algorithms Ltd, Swansea, UK
e-mail: k.a.dowsland@btconnect.com

the examples also belong to a class of problems known as linear programs (LP) and some of the discussion in the section on network flows makes use of the relationship between network flow problems and linear programming problems. Although knowledge of LPs is not necessary to understand the algorithms or examples as these are all couched in purely combinatorial terms we start the chapter with a brief introduction to linear programming. Further details can be found in [Anderson et al. \(1997\)](#).

The chapter is organized as follows. The overview of linear programming is followed by three sections introducing branch and bound, dynamic programming and network flow programming. In each case an introductory description is followed by two or more examples of their use in solving different problems, including a worked numerical example in each case. Each section ends with a brief discussion of more advanced issues. Section 2.6 looks at some problems that frequently occur as sub-problems in the solution of more complex problems and suggests algorithms based on the techniques covered in Sects. 2.3–2.5 for their solution. Section 2.7 takes a brief look at potential future applications and Sect. 2.8 provides some hints and tips on how to get started with each of the techniques. Additional sources of information not covered in the references are given at the end of the chapter.

2.2 Linear Programming

2.2.1 *Introduction*

This section provides a brief overview of those aspects of linear programming (LP) that are relevant to the remainder of this chapter. We start by outlining the basic features of an LP model and then go on to look at an important concept of such models—that of duality. We do not go into any detail with regard to solution algorithms for two reasons. Firstly, they are not necessary in order to understand the material presented in the remainder of the chapter. Secondly, LP packages and solution code are available from a wide variety of sources so that it is no longer necessary for a potential user to develop their own solution code.

2.2.2 *The Linear Programming Form*

A linear programming problem is an optimization problem in which both the objective (i.e. the expression that is to be optimized) and the constraints on the solution can be expressed as a series of linear expressions in the decision variables. If the problem has n variables then the constraints define a set of hyper-planes in n -dimensional space. These are the boundaries of an n -dimensional region that defines the set of feasible solutions to the problem and is known as the feasible region. The following example illustrates the form of a simple linear programming problem.

2.2.2.1 A Simple Example

A clothing manufacturer makes three different styles of T-shirt. Style 1 requires 7.5 min cutting time, 12 min sewing time, 3 min packaging time and sells at a profit of £3. Style 2 requires 8 min cutting time, 9 min sewing time, 4 min packaging time and makes £5 profit. Style 3 requires 4 min cutting time, 8 min sewing time and 2 min packaging time and makes £4 profit. The company wants to determine production quantities of each style for the coming month. They have an order for 1,000 T-shirts of style 1 that must be met, and have a total of 10,000 man hours available for cutting, 18,000 man hours for sewing and 9,000 man hours available for packaging. Assuming that they will be able sell as many T-shirts as they produce in any of the styles how many of each should they make in order to maximize their profit?

We can formulate the problem mathematically as follows. First we define three decision variables x_1 , x_2 and x_3 representing the number of T-shirts manufactured in styles 1, 2 and 3 respectively. Then the whole problem can be written as

$$\text{Maximize } 3x_1 + 5x_2 + 4x_3 \quad (2.1)$$

$$\text{subject to } 7.5x_1 + 8x_2 + 4x_3 \leq 10,000 \quad (2.2)$$

$$12x_1 + 9x_2 + 8x_3 \leq 18,000 \quad (2.3)$$

$$3x_1 + 4x_2 + 2x_3 \leq 9,000 \quad (2.4)$$

$$x_1 \geq 1,000 \quad (2.5)$$

$$x_1, x_2, x_3 \geq 0. \quad (2.6)$$

Expression (2.1) defines the profit. This is what we need to maximize and is known as the *objective function*. The remaining expressions are the *constraints*. Constraints (2.2)–(2.4) ensure that the hours required for cutting, sewing and packaging do not exceed those available. Constraint (2.5) ensures that at least 1,000 T-shirts of style 1 are produced and constraint (2.6) stipulates that all the decision variables must be non-negative. Note that all the expressions are linear in the decision variables, and we therefore have a linear programming formulation of the problem.

2.2.2.2 The General LP Format

In general, a linear program is any problem of the form

$$\begin{aligned} \max \text{ or } \min \quad & \sum_{i=1}^n c_i x_i \\ \text{such that} \quad & \sum_{i=1}^n a_{1i} x_i \sim b_1 \\ & \vdots \\ & \sum_{i=1}^n a_{mi} x_i \sim b_m \end{aligned} \quad (2.7)$$

where \sim is one of \geq , $=$ or \leq .

The important point about a linear programming model is that the feasible region is a convex space and the objective function is a convex function. Optimization theory therefore tells us that as long as the variables can take on any real non-negative values (possibly bounded above) then the optimal solution can be found at an extreme point of the feasible region. It is also possible to derive conditions that tell us whether or not a given extreme point is optimal. Standard LP solution approaches based on these observations can solve problems involving many thousands of variables in reasonable time. As we will see later in this chapter there are special cases where these techniques work even when the variables are constrained to take on integer or binary values. The general case where the variables are constrained to be integer, known as integer programming, is more difficult and is covered in Chap. 3.

Although it makes sense when formulating linear programmes to use the flexibility of the formulation above in allowing either a maximization or minimization objective and any combination of inequalities and equalities for the constraints, much linear programming theory (and therefore the solution approaches based on the theory) assume that the problem has been converted to a standard form in which the objective is expressed in terms of a maximization problem, all the constraints apart from the non-negativity conditions are expressed as equalities, all right-hand side values $b_1 \dots b_m$ are non-negative and all decision variables are non-negative. A general formulation can be converted into this form by the following steps.

1. If the problem is a minimization problem the signs of the objective coefficients $c_1 \dots c_n$ are changed.
2. Any variable not constrained to be non-negative is written as the difference between two non-negative variables.
3. Any constraint with a negative right-hand side is multiplied by -1 .
4. Any constraint which is an inequality is converted to an equality by the introduction of a new variable, (known as a slack variable) to the left-hand side. The variable is added in the case of a \leq constraint and subtracted in the case of a \geq constraint. The formulation is often written in matrix form:

$$\begin{aligned} & \max \quad CX \\ & \text{s.t. } AX = b \\ & \quad X \geq 0 \end{aligned} \tag{2.8}$$

where $C = (c_1 \dots c_n)$, $b = (b_1 \dots b_m)^T$, $X = (x_1 \dots x_n)^T$ and $A = (a_{ij})_{m \times n}$.

2.2.3 Duality

An important concept in linear programming is that of duality. For a maximization problem in which all the constraints are \leq constraints and all variables are non-negative, i.e. a problem of the form

$$\begin{aligned} \max \quad & CX \\ \text{s.t. } & AX \leq b \\ & X \geq 0 \end{aligned} \tag{2.9}$$

the dual is defined as

$$\begin{aligned} \min \quad & b^T Y \\ \text{s.t. } & A^T Y \geq C^T \\ & Y \geq 0. \end{aligned} \tag{2.10}$$

The original problem is known as the *primal*. Note that there is a dual variable y_i associated with each constraint in the primal problem and a dual constraint associated with each variable x_i in the primal. (The dual of a primal with a more general formulation can be derived by using rules similar to those used to convert a problem into standard form to convert the primal into the above form, with equality constraints being replaced by the equivalent two inequalities.)

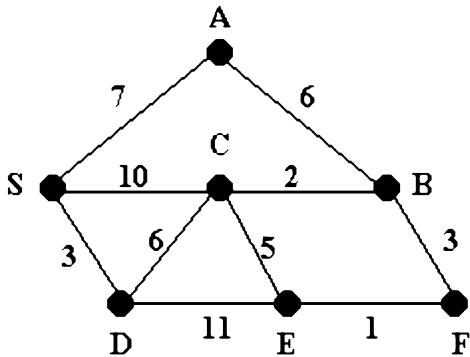
The dual has the important property that the value of the objective in the optimal solution to the primal problem is the same as the optimal solution for the dual problem. Moreover, the *theorem of complimentary slackness* states that if both dual and primal have been converted to standard form, with $s_1 \dots s_m$ the slack variables in the primal problem and $e_1 \dots e_n$ the slack variables in the dual, then if $X = (x_1, \dots, x_n)$ is feasible for the dual and $Y = (y_1, \dots, y_m)$ is feasible for the primal, X and Y are optimal solutions to the primal and dual respectively if and only if $s_i y_i = 0 \forall i = 1, m$ and $e_j x_j = 0 \forall j = 1, n$. This relationship is an important feature in the specialist solution algorithm for the minimum cost network flow algorithm presented later in this chapter and underpins other LP based solution approaches.

2.2.4 Solution Techniques

Solution approaches for linear programming fall into two categories. Simplex type methods search the extreme points of the feasible region of the primal or the dual problem until the optimality conditions are satisfied. The technique dates from the seminal work of [Dantzig \(1951\)](#). Since then the basic technique has been refined in a number of ways to improve the overall efficiency of the search and to improve its operation on problems with specific characteristics. Variants of the simplex approach are available in a number of specialist software packages, as a feature of some spreadsheet packages, and as freeware from various web-based sources.

Although they perform well in practice simplex based procedures suffer from the disadvantage that they have poor worst case time-performance guarantees. This deficiency lead to the investigation of interior point methods, so called because they search a path of solutions through the interior of the feasible region in such a way as to arrive at an optimal point when they hit the boundary. Practical interior point

Fig. 2.1 Shortest path network



methods can be traced back to the work of Kamarkar (1984), although Khachian (1979) ellipsoid method was the first LP algorithm with a polynomial time guarantee. Recent interior point methods have proved efficient, in particular for large LPs, and there has also been some success with hybridizations of interior point and simplex approaches. While the choice of solution approach may be important for very large or difficult problems, for most purposes standard available code based on simplex type approaches should suffice.

2.3 Branch and Bound

2.3.1 Introduction

When faced with the problem of finding an optimum over a finite set of alternatives an obvious approach is to enumerate all the alternatives and then select the best. However, for anything other than the smallest problems such an approach is computationally infeasible. The rationale behind the branch and bound algorithm is to reduce the number of alternatives that need to be considered by repeatedly partitioning the problem into a set of smaller subproblems and using local information in the form of bounds to eliminate those that can be shown to be sub-optimal. The simplest branch-and-bound implementations are those based on a constructive approach in which partial solutions are built up one element at a time. We therefore start by introducing the technique in this context before going on to show how it can be extended to its more general form.

Assume that we have a problem whose solutions consist of finite vectors of the form (x_1, x_2, \dots, x_k) where k may vary from solution to solution. Those combinations of values that form feasible vectors are determined by the problem constraints. The set of all possible solutions can be determined by taking each feasible value for x_1 , then for each x_1 considering all compatible values for x_2 , then for each partial solution (x_1, x_2, \dots) considering all compatible x_3 , etc. This process can be represented

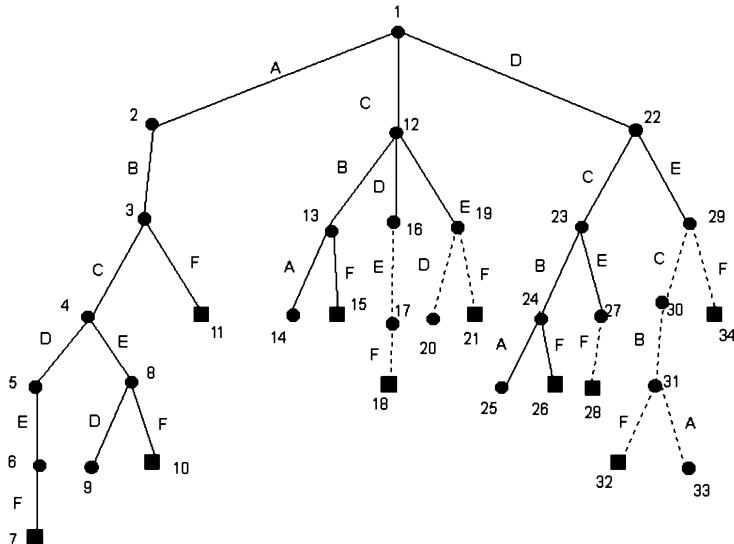


Fig. 2.2 Tree enumerating all simple routes

as a tree in which the branches at level i correspond to the choices for x_i given the choices already made for x_1, \dots, x_{i-1} , and the nodes at level i correspond to the partial solutions of the first i elements. This is illustrated with reference to Figs. 2.1 and 2.2. Figure 2.2 is the tree enumerating all simple routes (i.e. routes without loops) from S to F in the network shown in Fig. 2.1. The branches at level 1 represent all the possibilities for leaving S and the branches at lower levels represent all the possibilities for extending the partial solution represented by the previous branches by one further link. The node at the top of the tree is sometimes referred to as the *root*, and the relationship between a given node and one immediately below is sometimes referred to as *parent/child* or *father/son*. All nodes that can be reached from the current node by traveling down the tree are referred to as *descendants* and nodes without any descendants are *terminal nodes*.

If we wish to search the tree in an ordered fashion we need to define a set of rules determining the order in which the branches are to be explored. This is known as the *branching strategy*. The two simplest strategies are known as *depth-first search* and *breadth-first search*. Depth-first search (also known as branch and backtrack) moves straight down a sequence of branches until a terminal node is reached before backtracking up to the nearest junction. If there are any unexplored branches below this junction it will select one of these, again continuing downwards until a terminal node is reached. If all children of the current node have been explored the search backtracks to the previous node and continues from there. In contrast, breadth-first search enumerates all the branches at one level before moving on to the next level. Depth-first search is likely to find a feasible solution early and it does not have the vast storage requirements of breadth-first search. However, breadth-first search

allows comparisons to be made across the tree, facilitating removal of dominated sub-solutions. In Fig. 2.2 the nodes are numbered in depth-first search order using a strategy of ordering the branches at each level from left to right. For a breadth-first search the search order would be 1, 2, 12, 22, 3, 13, 16, 19, 23, 29, 4, 11, etc.

As mentioned above, although it is possible to enumerate the whole tree in a small example like this, the size of the tree grows explosively with problem size, and for most real-life problems complete enumeration is not possible. For example, complete enumeration of all feasible allocations of n tasks to n machines would result in a tree with $n!$ terminal nodes, i.e. $\sim 2.4 \times 10^{18}$ terminal nodes for 20 machines. The branch-and-bound process allows us to prove that some partial solutions cannot lead to optimal solutions and to cut them and their descendants from the search tree—a process known as *pruning*. This is achieved through the use of upper and lower bounds satisfying $\text{lower_bound} \leq z \leq \text{upper_bound}$, where z is the optimal solution obtained over all descendants of the current node. For a minimization problem the upper bound is a quantity UB such that we know we can do at least as well as UB . This is usually the best feasible solution found so far. The lower bound at node i , LB_i , is an optimistic estimate of the best solution that can be obtained by completing the partial solution at node i , i.e. we know that in exploring below node i we cannot do better than LB_i . Nodes where $LB_i \geq UB$ need not be explored further and we say that they are *fathomed*. When all nodes are fathomed the upper bound is the optimal solution. For a maximization problem the roles of the upper and lower bounds are reversed.

2.3.2 Branch and Bound Based on Partial Solutions

The success of a branch-and-bound implementation for a large problem depends on the number of branches that can be pruned successfully. This is largely dependent on the quality of the bounds, but the branching strategy can also have a significant effect on the number of nodes that need to be explored. The basic branch-and-bound approach and the influence of bound quality and branching strategy will be illustrated with reference to our shortest-path example.

2.3.2.1 Example 1: Finding the Shortest Path

We start by illustrating the use of simple bounds to prune the tree using the branching strategy defined by the node numbering. Our upper bound will be the cost of the shortest path found to date. Thus at the start we have $UB = \infty$. For the local lower bound at each node in the search tree we simply use the sum of the costs on the links traveled so far. Figure 2.3 gives details of the order in which the nodes are visited and the upper and lower bounds at each node.

The first seven branches correspond to the construction of the first path SABCDEF and the lower bound column gives the cost incurred by the partial

	Node	LB	UB		Node	LB	UB		Node	LB	UB
1	1	0	∞	16	4	15	21	31	19	15	15
2	2	7	∞	17	3	13	21	32	12	10	15
3	3	13	∞	18	11	16	16	33	1	0	15
4	4	15	∞	19	3	13	16	34	22	3	15
5	5	21	∞	20	2	7	16	35	23	9	15
6	6	32	∞	21	1	0	16	36	24	11	15
7	7	33	33	22	12	10	16	37	25	17	15
8	6	32	33	23	13	12	16	38	24	11	15
9	5	21	33	24	14	18	16	39	26	14	14
10	4	15	33	25	13	12	16	40	24	11	14
11	8	20	33	26	15	15	15	41	23	9	14
12	9	31	33	27	13	12	15	42	27	14	14
13	8	20	33	28	12	10	15	43	23	9	14
14	10	21	21	29	16	16	15	44	22	3	14
15	8	20	21	30	12	10	15	45	29	14	14

Fig. 2.3 Using simple bonds to prune the tree

solution at each stage. Note that when we reach node 7 we have our first complete path and so the upper bound is reduced to the cost of the path, i.e. 33. We now know that the optimal solution cannot be worse than 33. From node 7 we backtrack via nodes 6 and 5 to the junction at node 4 before continuing down the tree. Each time a cheaper path is completed the upper bound is reduced but the local lower bounds remain below the upper bound until we reach node 16. Here the cost of the partial solution $SCD = 16$ and the upper bound = 15. Thus this node is fathomed, and we backtrack immediately to node 12. Similarly, nodes 19, 27 and 29 are fathomed by the bounds and the branches below them can be pruned. When the search is complete the cost of the optimal solution = $UB = 14$ and is given by the path to node 26, SDCBF. All the branches represented by the dotted lines in Fig. 2.2 have been pruned and the search has been reduced from 33 to 23 branches.

This is already a significant reduction but we can do better by strengthening the bounds. The bound is based on the cost to date and does not make use of any estimate of the possible future cost. This can easily be incorporated in two ways. First, we know that we must leave the current vertex along a link to an unvisited vertex. Thus, we will incur an additional cost of at least as much as the cost of the cheapest such link. Similarly, we must eventually enter F. Thus, we must incur an additional cost at least as great as the cheapest link into F from the current vertex or a previously unvisited vertex. However, we cannot simply add both these quantities to the original bound as at vertices adjacent to F this will incur double counting. This highlights the need for caution when combining different bounds into a more powerful bound. We can now define our new lower bound LB_i as follows.

Let (x_1, \dots, x_j) be the current partial solution. Define $L_1 = \text{cost of path } (x_1, \dots, x_j)$, $L_2 = \text{cheapest link from } x_j \text{ to an unvisited vertex}$ and $L_3 = \text{cheapest link into F from any vertex other than those in path } (x_1, \dots, x_{j-1})$. Then $LB_i = L_1 + L_2 + L_3$ if x_j is not adjacent to F, and $LB_i = L_1 + \max(L_2, L_3)$ otherwise. Figure 2.4 gives details of the search using this new bound.

Note how the lower bound is now higher at each node and a good estimate of the cost of each path is obtained before the path is completed. Lower bounds of ∞ are

	Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB
1	1	4(0,3,1)	∞	13	8	21(20,1,1)	33	26	15	15	15
2	2	14(7,6,1)	∞	14	10	21	21	27	13	∞	15
3	3	15(13,2,1)	∞	15	8	∞	21	28	12	16(10,5,1)	15
4	4	21(15,5,1)	∞	16	4	∞	21	29	1	4(0,3,1)	15
5	5	33(21,11,1)	∞	17	3	16(13,3,1)	21	30	22	10(3,6,1)	15
6	6	33(32,1,1)	∞	18	11	16	16	31	23	12(9,2,1)	15
7	7	33	33	19	3	∞	16	32	24	14(11,3,1)	15
8	6	∞	33	20	2	∞	16	33	25	∞	15
9	5	∞	33	21	1	4(0,3,1)	16	34	24	14(11,3,1)	15
10	4	21(15,5,1)	33	22	12	13(10,2,1)	16	35	26	14	14
11	8	21(20,1,1)	33	23	13	15(12,3,1)	16	36	24	17(11,6,1)	14
12	9	∞	33	24	14	∞	16	37	23	15(9,5,1)	14
			25	13		15(12,3,1)	16	38	22	15(3,11,1)	14

Fig. 2.4 Search using improved bounds

	Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB
1	1	4(0,3,1)	∞	6	24	17(11,6,1)	14	11	1	11(0,10,1)	14
2	22	10(3,6,1)	∞	7	23	15(9,5,1)	14	12	12	13(10,2,1)	14
3	23	12(9,2,1)	∞	8	22	15(3,11,1)	14	13	13	15(12,3,1)	14
4	24	14(11,3,1)	∞	9	1	8(0,7,1)	14	14	12	17(10,6,1)	14
5	26	14	14	10	2	14(7,6,1)	14	15			

Fig. 2.5 Search using improved branching strategy

recorded whenever the branches from a particular node are exhausted. There is more work incurred in calculating this bound, not only because the actual calculation is more complex, but also because the bound at a given node may change when returning to the node in a backtracking step. This strengthens the bound and reduces the total number of branches searched to 19.

Finally, we consider the branching strategy. So far we have used simple depth-first search taking the branches at each node in the given order. In general, the efficiency of the search will be increased if the upper bound can be reduced sooner, or if the levels of the tree can be organized so as to incur high lower bounds closer to the root of the tree. Here we apply a strategy that will favor the former, and bias the search towards finding shorter routes first. This is achieved by exploring the branches at each node in increasing cost order instead of from left to right. Figure 2.5 shows the result of the search using this strategy.

Now the search starts by selecting the D branch from the root node and quickly finds the optimal solution of 14. This results in early pruning of the nodes from the other two branches and the whole search is completed in seven branches.

This example has illustrated how the size of the search tree is dependent on both the quality of the bounds and the branching strategy. However, it should be noted that this is not the most efficient way of solving the shortest-path problem and better approaches are suggested in Sect. 2.5. Nevertheless, since early papers on the technique in the early 1960s, it has proved successful in solving a wide range of both classical and practical problems. Examples include algorithms for a range of graph-theoretic problems, such as node coloring (Brown 1972; Zykov 1949),

clique and independent set problems (Bron and Kerbosch 1973), location problems (Erlenkotter 1978; Jarvinen et al. 1972) and the TSP (Little et al. 1963; Held and Karp 1970; Balas and Christofides 1981), and for several classical combinatorial optimization problems such as knapsack problems (Martello and Toth 1981), set covering and set partitioning (Garfinkel and Nemhauser 1969) and generalized assignment problems (Ross and Soland 1975). We use one of these, Brown's graph-coloring algorithm, to consolidate the ideas of the last section.

2.3.2.2 Example 2: Brown's Algorithm for Graph Coloring

An example of a problem that has been tackled using a variety of branch-and-bound approaches is that of graph coloring. The graph-coloring problem is that of minimizing the number of colors needed to color the vertices of a graph so that no two adjacent vertices are given the same color. The graph-coloring problem is an interesting example as it is the underlying model for many timetabling and scheduling problems. Brown's (1972) algorithm is an example of a branch-and-bound approach based on partial solutions. The algorithm is a straightforward application of the branch-and-bound process using simple bounds. As with the shortest-path problem, its efficiency can be improved by applying some intelligence to the branching strategy. However, the strength of the algorithm lies in its backtracking strategy that essentially prunes away branches by backtracking up several levels at a time where appropriate. The algorithm can be summarized as follows.

The branches in the tree correspond to the decision to color a given vertex in a given color. In the simplest version, the vertices are pre-ordered and the branches at level i correspond to choosing a color for the i th vertex. The colors are also ordered and the branches at each node are searched in color order. The upper bound is given by the number of colors in the best solution to date and the lower bound on each partial solution is given by the number of colors used up to that point. If the upper bound is equal to Q , then when the search backtracks to a vertex v_i for which there are no unexplored branches corresponding to colors below Q in the ordering, that node is obviously bounded and a further backtracking step must be executed. Rather than simply backtracking to the previous node the search recognizes the fact that in order to progress it is necessary for an alternative color to become free for v_i . This will only be achieved if a neighbor of v_i is uncolored. Therefore, the search backtracks up the tree until a neighbor of v_i is encountered before attempting a forward branch. If further backtracking, say at vertex v_j , is required before v_i has been successfully re-colored then re-coloring a neighbor v_j may also achieve the desired result so v_j 's neighbors are added to those of v_i in defining a potential backtracking node. In order to manage this backtracking strategy in a complex search, those vertices that are neighbors of backtracking vertices are stored in a queue in reverse order and branching takes place from the first vertex in the queue. A formal definition of the algorithm is given below. The list of identifiers, J , is the set of nodes

which trigger the bounding condition and *Queue* is an ordered list of the neighbors of elements in J :

Step 0. Define orderings

Order the vertices $1, 2, \dots, n$ and colors c_1, c_2, \dots

$\Gamma^-(i)$ denotes those neighbors of vertex i which precede i in the ordering.

Step 1. Find initial coloring

Color the vertices in order using the lowest indexed feasible color.

Step 2. Store new solution

Let q be the number of colors used. Set the upper bound equal to q and store the current q -coloring.

Set list of identifiers, $J = \emptyset$.

Step 3. Backtrack

3.1 Find first vertex corresponding to local $LB = q$

Let i^* be the first vertex colored q .

3.2 Update list of backtracking vertices and corresponding queue of neighbors

Remove all $j < i^*$ from J .

Set $J = J \cup \{i^*\}$.

Set $Queue = \bigcup_{j \in J} \Gamma^-(j)$ in reverse order.

3.3 Backtrack to level defined by first vertex on the queue

Let i' be the first vertex on the queue. Let q' be its color.

If $i' = k$ and vertices $1, 2, \dots, k$ are colored c_1, c_2, \dots, c_k then STOP. Stored solution is optimal.

Otherwise uncolor all $i \geq i'$.

Step 4. Branch

4.1 Recolor i'

Color i' in the first feasible color $\{q' + 1, q' + 2, \dots, q - 1\}$.

If no feasible color set $i^* = i'$ and goto 3.2.

4.2 Recolor remaining vertices

Attempt to color vertices $i = i' + 1, n$ in colors 1 to $q - 1$ using first feasible color.

If vertex i requires color q then set $i^* = i$ and goto 3.2.

Otherwise go to step 2.

Note that Steps 1 and 4.2 amalgamate several forward branches into one step, building on the partial coloring until the bounding condition is reached. Similarly, in Step 3.3 several backtracking steps are amalgamated. Note also that the lower bounds are not stored explicitly as the first node with a local lower bound of q will always correspond to the point where color q was used for the first time.

The algorithm is illustrated with reference to the graph in Fig. 2.6 using the given ordering of the vertices and colors in alphabetical order:

Step 1. Initial coloring = 1A, 2B, 3A, 4C, 5A, 6D, 7B, 8D. $q = 4$.

Step 2. $J = \emptyset$.

Step 3. $i^* = 6$. $J = \{6\}$. $Queue = \{4, 2, 1\}$.

Backtrack to node 4. $q' = C$. Partial coloring = 1A, 2B, 3A.
 Step 4. Vertex 4 already colored in $q - 1$. $i^* = 4$. goto 3.2.
 Step 3. $J = \{6, 4\}$. $Queue = \{3, 2, 1\}$.
 Backtrack to node 3. $q' = A$. Partial coloring = 1A, 2B.
 Step 4. Color 3 in color C and continue coloring 1A, 2B, 3C, 4A, 5B, 6C, 7A, 8C.
 $q = 3$.
 Step 2. $J = \emptyset$.
 Step 3. $i^* = 3$. $J = \{3\}$. $Queue = \{2\}$. Stopping condition reached and solution with $q = 3$ is an optimal solution in three colors.

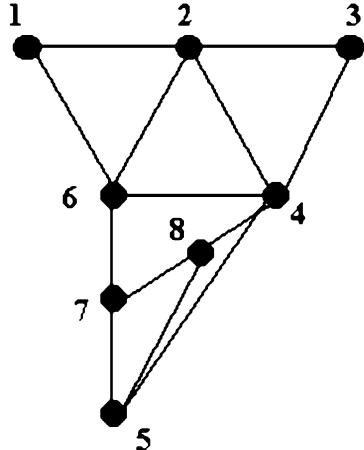
As with the shortest-path implementation, the efficiency of the search can be improved by an intelligent ordering of the vertices to encourage good solutions to be found more quickly. The simplest improvement is to pre-order the vertices in decreasing degree order. However, there is no reason why the ordering of the vertices should remain the same throughout the tree, and greater efficiency gains can be obtained using some form of dynamic ordering such as selecting the vertex with the largest number of colors already used on its neighbors to color next—a strategy known as DSATUR. It is also worth considering the stopping condition in terms of the vertex ordering. The condition is valid because backtracking beyond the point where vertices 1 to k are colored in colors 1 to k will simply result in equivalent colorings with a different permutation of colors. This condition will be achieved more quickly if the ordering starts with a large clique. Thus a good strategy is to find a large clique and place these vertices in a fixed order at the start and then to use a dynamic ordering for the remaining vertices.

2.3.3 A Generalization

So far the discussion has focused on search trees based on building up partial solutions. Such approaches have proved popular for a variety of problems. However, they are just a special case of a more general strategy in which the branches correspond to adding constraints to the problem. In the case of a partial solution the constraints take the form $x_i = a_i$. The more general format underpins the branch-and-bound strategy for integer programming and will be treated in detail in Chap. 3. Therefore we will not go into detail here. Instead we will briefly illustrate the approach with an alternative tree search approach to the graph-coloring problem.

2.3.3.1 Zykov's Algorithm for Graph Coloring

Consider any two non-adjacent vertices v_i and v_j . In any solution to the graph-coloring problem there are two possibilities. Either they will be allocated the same color or they will be allocated different colors. The optimal solution subject to them being in the same color is the optimal coloring in a graph with v_i and v_j merged

Fig. 2.6 Coloring example

into a single vertex, while the optimal solution in the latter case is the optimal coloring in the original graph with edge (v_i, v_j) added. Obviously the better of these two solutions is optimal with respect to the original problem. We also observe that if we continue a sequence of adding edges and/or merging vertices in an arbitrary graph then we will eventually be left with a complete graph (i.e. a graph in which every vertex is adjacent to every other). A complete graph with n vertices obviously requires n colors. These observations form the basis of [Zykov's algorithm \(1949\)](#) in which there are two branches at each level of the tree corresponding to the decision as to whether two non-adjacent vertices will be allocated the same or different colors. The two child nodes represent the coloring problems in the two suitably modified graphs and the terminal nodes will all be complete graphs. The smallest complete graph defines the optimal coloring. This is illustrated in Fig. 2.7, which shows the search tree that results from the problem of finding the optimal coloring of the sub-graph defined by vertices 1, 2, 3, 4 and 6 of the graph in Fig. 2.6.

The left-hand branch at each level constrains two non-adjacent vertices to be the same color and the child node is obtained by taking the graph at the parent node and merging the two vertices. The right-hand branch constrains the same two vertices to be different colors and the child is formed by adding an edge between the two relevant vertices in the parent graph. Branching continues until the resulting child is a complete graph. Here the terminal nodes reading from left to right are complete graphs of size 4, 3, 4, 4 and 5 respectively. The optimal solution is given by the complete graph on three vertices in which vertices 1 and 4 are allocated to one color, 3 and 6 to a second color and vertex 2 to the third.

A suitable upper bound is again the best solution found so far. A lower bound on the optimal coloring in each sub-graph can be defined by the largest clique it contains (a clique is a set of vertices such that each vertex in the set is adjacent to every other). Finding the largest clique is itself a difficult problem but a heuristic can be used to get a good estimate. In Fig. 2.7 using a depth-first search and exploring

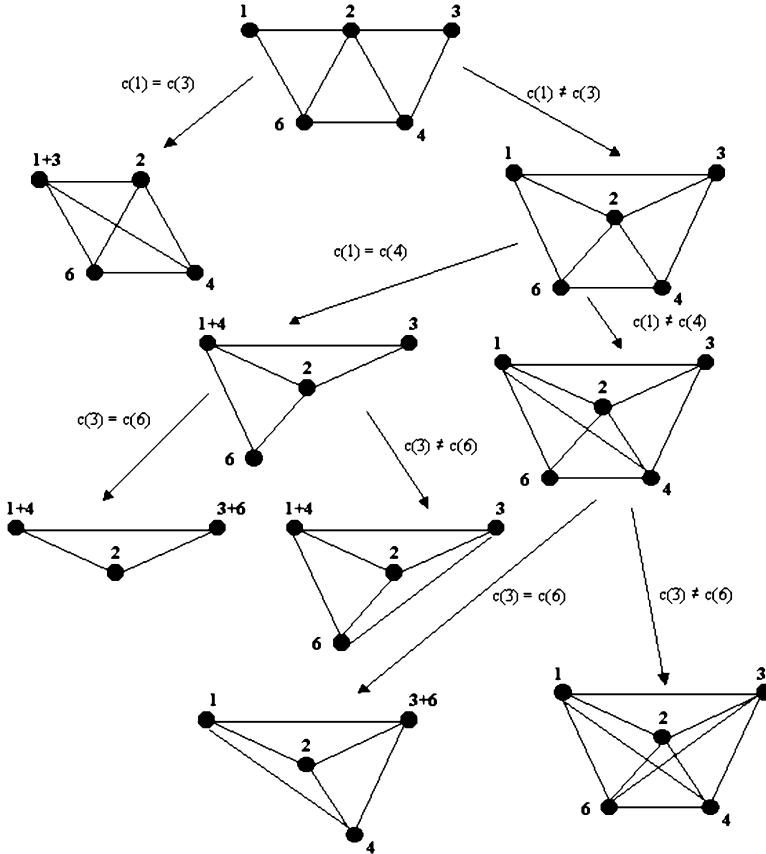


Fig. 2.7 Zykov's search tree for optimal coloring

the $c(i) = c(j)$ branch first at each node, we might recognize that the parent of the node representing the optimal solution contains two cliques of size 3. Similarly, its parent contains cliques of size 3. Thus we can backtrack straight up to the route node saving a total of four branches. Branching strategies can be designed to produce dense sub-graphs quickly thereby increasing the chances of finding large cliques early in the tree. More recent versions of the algorithm make use of theoretical results that state that certain classes of graph, known as perfect graphs, are easy to color. Branching strategies are designed to produce graphs belonging to one of the classes of perfect graph as early in the tree as possible. These can then be colored optimally, thus saving all the branches required in order to reduce them to complete graphs. See [Golumbic \(1980\)](#) for a wide-ranging treatment of perfect graphs and associated algorithms.

2.3.4 Other Issues

2.3.4.1 Bounds

The most important feature of a branch-and-bound algorithm is probably the quality of the bounds and it is usually worth putting considerable effort into ensuring that these are as tight as possible. In the case of lower bounds this is often achieved by exploiting as much information about the problem as possible. For example, [Dowsland \(1987\)](#) used a clique model to solve a class of packing problems known as the pallet loading problem. She combined the bounds in a published maximum clique algorithm with bounds derived from geometric aspects of the physical problem and showed that the percentage of problems solved within a given time frame rose from 75 to 95 %. In some cases a problem may become easy to solve if some of the constraints are removed. This is a process known as relaxation and the solution to the relaxed problem will always provide a valid bound on the solution to the original problem. This approach was used by [Christofides and Whitlock \(1977\)](#) in their solution to a guillotine cutting problem in which a large stock-sheet of material must be cut into a set of smaller rectangular pieces, using a sequence of guillotine cuts, so as to maximize the value of the cut pieces. In their version of the problem the demand for pieces of each dimension was constrained, whereas the unconstrained version of the problem is relatively easy to solve. Their solution uses a tree search in which each branch represents a cut, and the nodes define the set of rectangles in a partial solution. Bounds are obtained by solving the unconstrained problems for each of the sub-rectangles in the partial solution. Although such relaxation bounds can be quite effective, there is often a gap between them and the solution to the constrained problem. This gap can sometimes be reduced by incorporating a suitable penalty for the violated constraints into the objective function. This is the basis of Lagrangian relaxation, which has proved a popular bound for a variety of branch-and-bound algorithms. For example, [Beasley \(1985\)](#) uses the approach for a non-guillotine version of the cutting stock problem. In Lagrangian relaxation an iterative approach is used to set parameters that will increase the tightness of the bound. Such an approach is obviously time-consuming but for moderately-sized problems in a wide variety of application areas the computational effort is well worth the number of branches it saves. Details of Lagrangian relaxation can be found in [Fisher \(1985\)](#).

The search efficiency is also affected by the upper bound. As we have seen, pruning is more effective when good solutions are found early. Earlier pruning may result if a heuristic is used before the start of the search to find a good solution that can be used as an upper bound at the outset. Similarly, using a heuristic to complete a partial solution and provide a local upper bound should also help in fathoming nodes without branching all the way down to terminal nodes.

It is also worth noting that bounding conditions based on information other than numeric upper and lower bounds may be useful in avoiding infeasible solutions or solutions that are simply permutations or sub-sets of solutions already found. An example of this approach is the maximal clique algorithm of [Bron and Kerbosch](#)

(1973) which includes bounding conditions based on relationships between the branches already explored and those yet to be visited from a given node.

2.3.4.2 Branching

There are also issues concerning branching strategies that have not been discussed. We have assumed that the search is always carried out in a depth-first manner. An alternative that can be successful if the bounds are able to give a good estimate of the quality of solutions below each branch is to use a best-first strategy, in which nodes at different levels across the breadth of tree may be selected to be evaluated next according to an appropriate definition of *best*. In our examples, the ordering of branches was geared towards finding good solutions as early as possible. An alternative strategy is to select branches that will encourage bounding conditions to be satisfied sooner rather than later. This approach is taken by Bron and Kerbosch in their branch-and-bound algorithm for finding cliques in a graph. At each node the next branch is chosen so as to encourage the bounding condition to occur as early as possible. Comparisons between this version and a version in which the branches are selected in the natural order show that the advantage, in terms of computation time, of using the more complex strategy increases rapidly with problem size.

2.3.4.3 Miscellaneous

Although good bounds and branching strategies have resulted in many successful branch-and-bound algorithms, it should be noted that the size of the search tree will tend to grow exponentially with problem size. It is therefore important to make sure that the underlying tree is as small as possible. For example, thought should be given to avoiding branches that lead to solutions that are symmetric to each other if at all possible. It may also be possible to apply some form of problem reduction in a pre-processing phase. For example, Garfinkel and Nemhauser (1969) outline a set of reductions for both set-covering and set-partitioning problems. It should also be noted that such a strategy may provide further reductions when applied to the subproblems produced within the search itself.

Finally, it is worth noting that in many implementations the optimal solution is found quickly and most of the search time is spent in proving that this is in fact the optimal solution. Thus if there is insufficient time to complete the search, the best solution to date can be taken as a heuristic solution to the problem. An alternative that is useful if a heuristic solution with a performance guarantee is required is to replace the upper bound with $UB(1 - \alpha)$. The tighter bound should enable the search to be completed more quickly and will guarantee a solution within $\alpha \times 100\%$ of the optimum.

2.4 Dynamic Programming

2.4.1 Introduction

Like branch and bound, dynamic programming (DP) is a procedure that solves optimization problems by breaking them down into simpler problems. It solves the problem in stages, dealing with all options at a particular stage before moving on to the next. In this sense it can often be represented as a breadth-first search. However, unlike the levels of the tree in branch and bound which partition the problem by adding constraints, the stages in DP are linked by a recursive relationship. The name dynamic programming derives from its popularity in solving problems that require decisions to be made over a sequence of time periods. Even when this is not the case, the name dynamic programming is still widely used, but the term *multi-stage programming* is sometimes used as an alternative.

The basis of DP is Bellman's *principle of optimality* (Bellman 1957) states that "the sub-policy of an optimal policy is itself optimal with regard to start and end states". As an illustration, consider the shortest-route problem. If we are told that in Fig. 2.1 the optimal route from S to F goes via E then we can be sure that part of the route from S to E is the optimal route between S and E, and that part from E to F is the optimal route from E to F. In other words, each sub-path of the optimal path is itself the shortest path between its start and end points. Any DP implementation has four main ingredients. These are stages, states, decisions and policies. At each stage, for each feasible state we make a decision as to how to achieve the next stage. The decisions are then combined into sub-policies that are themselves combined into an overall optimal policy. DP is a very general technique that has been applied at varying levels of complexity. These have been classified into four levels: deterministic, stochastic, adaptive and residual. Our treatment here will be limited to deterministic problems.

The design of a DP algorithm for a particular problem involves three tasks; the definition of the stages and states, the derivation of a simple formula for the cost/value of the starting stage/state(s) and the derivation of a recursive relationship for all states at stage k in terms of previous stages and states.

The definition of the stages and states will obviously depend on the problem being tackled, but there are some definitions that are common. Stages are frequently defined in terms of time periods from the start or end of the planning horizon, or in terms of an expanding subset of variables that may be included at each stage. Common definitions of states are the amount of product in stock or yet to be produced, the size or capacity of an entity such as stock sheet, container, factory or budget, or the destination already reached in a routing problem.

2.4.2 Developing a DP Model

2.4.2.1 Forward Recursion and the Unbounded Knapsack Problem

We will first illustrate the concepts of DP by reference to a classical optimization problem—the unbounded knapsack problem. The problem can be stated as follows. Given a container of capacity b and a set of n items of size w_i and value v_i for $i = 1, n$ such that the number of each item available is unbounded, maximize the value of items that can be packed into the container without exceeding the capacity.

The problem can be formulated as follows:

$$\max \sum_{i=1}^n v_i x_i \quad (2.11)$$

$$\text{s.t. } \sum_{i=1}^n w_i x_i \leq b \quad (2.12)$$

for $x_i \geq 0$ and integer, where x_i equals the number of copies of item i in the solution.

We can formulate this problem as DP as follows. Define $F_k(S)$ to be the maximum value for a container of capacity S using items of sizes 1 to k . Here the items available represent the stages and the capacity available the states. $F_1(S)$ is the value that can be obtained if the only pieces available are those of type 1.

This is given by

$$F_1(S) = \int \left(\frac{S}{w_1} \right) v_1. \quad (2.13)$$

All that remains is to define a recursive relationship for $F_k(S)$ in terms of previous stages and states. This is achieved as follows. The optimal solution either makes use of at least one item of type k , or it does not contain any items of type k . In the latter case $F_k(S) = F_{k-1}(S)$. In the former case one copy of item k takes up w_k units of capacity and adds v_k units of value. Bellman's principle tells us that the remaining $S - w_k$ units of capacity must be packed optimally. This packing may contain further items of type k and is given by $F_k(S - w_k)$. Thus we have for $k > 1$:

$$\begin{aligned} F_k(S) &= \max\{F_{k-1}(S), F_k(S - w_k) + v_k\} \quad \text{for } S \geq w_k \\ F_k(S) &= F_{k-1}(S) \quad \text{otherwise.} \end{aligned} \quad (2.14)$$

The solution to the overall problem is given by $F_n(b)$. We will illustrate the procedure with the following example.

Let $n = 3$, $b = 19$, $w_1, w_2, w_3 = 3, 5$ and 7 respectively and $v_1, v_2, v_3 = 4, 7$ and 10 respectively.

The values of $F_k(S)$ for $k = 1, 3$ and $S = 0, 19$ are given in Table 2.1. The values in column $k = 1$ are first calculated using Eq. (2.3). Then the subsequent columns are calculated in order starting from the top and working down using Eq. (2.4). For $S \geq w_k$ the appropriate value is obtained by comparing the value in row S in the previous column with the sum of v_k and the value in the current column w_k rows up.

Table 2.1 Unbounded knapsack calculations

S/k	1	2	3	S/k	1	2	3
0	0	0	0	10	12	14	14
1	0	0	0	11	12	15	15
2	0	0	0	12	16	16	17
3	4	4	4	13	16	18	18
4	4	4	4	14	16	19	20
5	4	7	7	15	20	21	21
6	8	8	8	16	20	22	22
7	8	8	10	17	20	23	24
8	8	11	11	18	24	25	25
9	12	12	12	19	24	26	27

The value of 27 in row 19, column 3 tells us that the optimal value is 27. In order to determine how this solution is achieved we need to work backwards. We need to find out whether this value came from $F_2(19)$ or $F_3(19 - 7) + 10$. The latter option is the correct one. We therefore record one item of type 3 and check the source of the value 17 in $F_3(12)$. This is either $F_2(12)$ or $F_3(5) + 10$. Once again the latter option is correct. We record a second item of type 3 and move to $F_3(5)$. $F_3(5) = F_2(5)$ so we check the source of the value of $F_2(5) = F_2(0) + 7$. Thus we record an item of type 2. As we have reached the top row of the table corresponding to capacity = 0 the solution is completed and is given by $x_1 = 0, x_2 = 1, x_3 = 2$.

2.4.2.2 Backward Recursion and a Production Planning Problem

In the above example the recursion worked in a forward direction with the stages corresponding to an increasing subset of variables. Our second example is taken from the field of production planning and is typical of many multi-period problems in that it is solved using backwards recursion, i.e. by working backwards from the end of the planning period. The problem can be summarized as follows.

Production of a single product is to be planned over a fixed horizon of n time periods. At the start there are S_0 units in stock and no stock is required at the end. Each time period t_i has a known demand d_i which *must* be met. Up to Q units can be produced in any one time period. The cost of making q units is given by $c(q)$ and economies of scale mean that $c(q)$ is not linear in q . Surplus units can be stored from one time period to the next at a warehousing cost of w per unit. There are natural definitions of the stages and states for this problem in terms of the time periods and stock levels. However, there is no simple formula for deciding what should be done in time period 1. Instead we re-order the time periods in reverse order and relate stage k to period $(n - k)$. If we start the last period with S units in stock, then we must meet the demand d_n exactly as we are to finish without any surplus. Thus we must produce $d_n - S$ units. The formula for the optimal policy at the starting stage is therefore

Table 2.2 Production costs and demands

Units	Production costs					
	0	1	2	3	4	5
Cost (£1000s)	0	7	13	16	20	24
Demands						
Period	1	2	3	4		
Demand	3	6	1	2		

$$F_0(S) = c(d_n - S). \quad (2.15)$$

We now need to define a recursive formula for $F_k(S)$ in terms of previous stages. If we start period $n - k$ with S units in stock and make q units we end with $S + q - d_{n-k}$ in stock. This will incur a warehousing cost and will define the starting stock for the next time period. The optimal policy from this point on has already been calculated as $F_{k-1}(S + q - d_{n-k})$. Thus the recursive formula is given by

$$F_k(S) = \min_{d_{n-k} - S \leq q \leq Q} \{c(q) + w(S + q - d_{n-k}) + F_{k-1}(S + q - d_{n-k})\}. \quad (2.16)$$

The lower limit on q ensures that production is sufficient to meet demand.

We also need to define the set of states that need to examined at each stage k . This can be limited in three ways. First, there is no point in having more stock than can be sold in the remaining time periods. Second, it is not possible to have more stock than could be produced up to that time period less than already sold. Third, it is not feasible to have a stock level that will not allow demand in future periods to be met. Thus for period $n - k$ we have $\text{MIN}_k \leq S \leq \min\{\text{MAX1}_k, \text{MAX2}_k\}$, where

$$\text{MAX1}_k = \sum_{i=n-k}^n d_i, \quad \text{MAX2}_k = S_0 + \sum_{i=1}^{n-k-1} (Q - d_i)$$

and

$$\text{MIN}_k = \max \left\{ 0, \max_{n-k \leq j \leq n} \left\{ \sum_{i=n-k}^j (d_i - Q) \right\} \right\}.$$

Once again we illustrate the formulation with a concrete example. Let $n = 4$, $Q = 5$, $S_0 = 1$, $w = 2,000$ and production costs and demands as given in Table 2.2. Working in units of £1,000 the calculations for the stages are then

Stage 0:

$$\text{MAX1}_0 = 2, \text{MAX2}_0 = 6, \text{MIN}_0 = 0$$

$$F_0(0) = c(2) = 13, F_0(1) = c(1) = 7, F_0(2) = c(0) = 0.$$

Stage 1:

$$\text{MAX1}_1 = 3, \text{MAX2}_1 = 2, \text{MIN}_1 = 0$$

$$\begin{aligned} F_1(0) &= \min\{c(1) + 0 \cdot w + F_0(0), c(2) + 1 \cdot w + F_0(1), c(3) + 2 \cdot w + F_0(2)\} \\ &= \min\{7 + 0 + 13, 13 + 2 + 7, 16 + 4 + 0\} = 20 \end{aligned}$$

$$F_1(1) = \min\{0+0+13, 7+2+7, 13+4+0\} = 13$$

$$F_1(2) = \min\{0+2+7, 7+4+0\} = 9$$

Stage 2:

$$MAXI_2 = 9, MAX2_1 = 3, MIN_1 = 1$$

$$F_2(1) = \min\{24+0+20\} = 44$$

$$F_2(2) = \min\{20+0+20, 24+2+13\} = 39$$

$$F_2(3) = \min\{16+0+20, 20+2+13, 24+4+9\} = 35$$

Stage 3: We do not need to calculate limits on S as we know that starting stock equals 1:

$$F_3(1) = \min\{16+2+44, 20+4+39, 24+6+35\} = 62.$$

Note that in many cases the full range of values for q from $S - d_{n-k}$ to Q have not been included in the minimization as they would lead to overstocking or under stocking. For example, in calculating $F_1(0)$ we do not consider any value of q above 3 as this would give more than two units at the start of the last time period. Similarly, we do consider q less than 3 in $F_3(1)$ as we need at least one unit in stock at the start of time period 2.

As with the knapsack problem, the calculations give the cost of the optimal solution but we need to work backwards in order to derive the optimal solution. Starting with $F_3(1)$ we note that the minimum value resulted from manufacturing three units, which leaves one unit in stock once the demand for three units has been met. Thus the policy from time period 2 onwards is given by $F_2(1)$. This is optimized by producing five units, leaving zero in stock. We therefore move to $F_1(0)$ which is optimized in two ways—producing 1 and leaving 0 in stock or producing 3 and leaving 2 in stock. This implies that there are two optimal solutions. The former is completed using $F_0(0)$ and the latter using $F_0(2)$. The two solutions are summarized in Table 2.3.

2.4.3 Other Issues

One of the main criticisms of a DP approach is that the number of subproblems that need to be solved is dependent not only on the stages but also on the states. While the number of stages is usually related to the size of the problem in the traditional sense (i.e. as a function of the number of variables) the number of states are frequently related to the size of the constants in the problem. For example, in the knapsack problem the number of states depends on the capacity of the container, while the number of states for the production planning problem is essentially bounded by a function of the maximum production capacity Q . For real-life problems such quantities may be extremely large. This is often exacerbated by the fact that the states may be multi-dimensional. For example in the standard DP formulation for two-dimensional cutting problems the states are defined by rectangles of dimension $X \times Y$. Our two examples were also relatively simple in that the recursive

Table 2.3 The two optimal solutions

Production plan 1					
Period	Starting stock	Make	Sell	Closing stock	Cost (£) production + warehousing
1	1	3	3	3	18,000
2	1	5	6	0	24,000
3	0	3	1	2	20,000
4	2	0	2	0	0
Total					62,000

Production plan 2					
Period	Starting stock	Make	Sell	Closing stock	Cost (£) production + warehousing
1	1	3	3	3	18,000
2	1	5	6	0	24,000
3	0	1	1	0	7,000
4	0	2	2	0	13,000
Total					62,000

relationship relied only on the solutions at the previous stage. Many DP formulations require recursive relationships that use all previous stages, thus necessitating the results of all previous calculations to be stored, resulting in pressure on available memory in addition to long computation times. It is therefore important that some thought is given to reducing the number of states. [Findlay et al. \(1989\)](#) use a model similar to our production planning example to plan daily production in an oil field so as to meet a quarterly production target. The states at each stage are given by the amount still to be produced before the end of the quarter. Thus in their basic model the number of states is given by the number of days in the quarter multiplied by the total quarterly target. However, there are upper and lower bounds on daily production and by using these to produce three bounds on the feasible states at each stage, the total size of the search space can be reduced to less than half its original size.

Although the need to calculate and store all sub-solutions is often seen as a drawback of DP, it can also be viewed as an advantage, as there is no need to carry out a whole new set of calculations if circumstances change. For example, in the production planning example, if for some reason we only managed to make two units instead of three in the third period, we could adopt the optimal policy from that point on simply by selecting the policy given by $F_1(1)$ instead of $F_1(2)$. This flexibility is cited as one of the reasons for the choice of DP as a solution technique by [Findlay et al. \(1989\)](#), as oil production is regularly affected by problems that may cause a shortfall in production on a particular day. Another example of the usefulness of being able to access the solutions to all subproblems without additional computational effort arises in the solution of two-dimensional cutting problems. The bounds used by [Christofides and Whitlock \(1977\)](#) in their branch-and-bound algorithm cited in

the previous section are calculated using a DP approach. The bound at the root node requires the solution to the unconstrained guillotine cutting problem in a rectangle of dimensions $X \times Y$ and the bounds at the other nodes require solutions to the same problem in smaller rectangles. These are precisely the problems solved in the various stages. Therefore, once the bound at the root node has been calculated, bounds for all the other nodes are available without further computation.

It is also worth emphasizing that DP is a very general approach. While this can be regarded as one of its strengths, it can also be a drawback in that there are few rules to guide a beginner in its use for a completely new problem. In many cases it is relatively easy to define the stages of an implementation but it is more difficult to find a suitable definition for the states. Although there are examples of DP being used to solve a variety of problems, the vast majority still lie in the areas of multi-period planning, routing and knapsack type problems where it is relatively easy to adapt existing approaches. We have already mentioned the production planning problem tackled by [Findlay et al. \(1989\)](#). Other examples are a multi-period model for cricketing strategy ([Clarke and Norman 1999](#)), a model for optimizing the route taken in orienteering ([Hayes and Norman 1984](#)), and a multiple-choice knapsack model for pollution control ([Bouzaher et al. 1990](#)).

2.5 Network Flow Programming

2.5.1 Introduction

Network flow programming deals with the solution of problems that can be modeled in terms of the flow of a commodity through a network. At first glance it appears that such models might be very limited in their application, perhaps encompassing areas such as the flow of current in electrical networks, the flow of fluids in pipeline networks, information flow in communications networks and traffic flow in road or rail networks. However, their scope is far wider. They not only encompass a wide range of graph and network problems that appear to have little to do with flows, such as shortest path, spanning tree, matching and location problems, but also model a wide range of other problems ranging from scheduling and allocation problems to the analysis of medical x-rays. Network flow problems can be categorized as integer programming problems with a special structure. For the basic network flow models that deal with homogeneous flows this structure impacts on the solution process in two ways. First, the constraint matrix of the LP formulation has the property that it is *totally unimodular*. This implies that any solution at the extreme points of the feasible region will be integer valued. From a practical point of view this means that as long as all the constants in a problem are integer valued then solution via the simplex method will also be integer valued. Thus integer programs that have the special structure of a network flow problem can be solved without recourse to any of the specialist integer programming techniques described in Chap. 3. However the

structure of the problem also means that it can be solved directly by combinatorial algorithms that are simpler to implement than the full simplex algorithm. The inspiration for and the verification of the optimality of these procedures is rooted in the underlying LP theory. We will start by looking at the maximum flow problem, the simplest but least flexible of the network flow formulations, in order to introduce the basic concepts and building blocks that will be used in the solution of a more flexible model, the minimum cost flow problem.

2.5.2 The Maximum Flow Problem

2.5.2.1 Introduction

The maximum flow problem is that of maximizing the amount of flow that can travel from source S to sink T in a network with capacities or upper bounds on the flow through each of its arcs. The problem can be stated as follows:

Let S = source, T = sink

x_{ij} = flow in arc (i, j)

V = total flow from S to T

u_{ij} = upper bound on arc (i, j)

$$\max V \text{ s.t. } \sum_j \underset{(i,j) \in A}{x_{ij}} - \sum_k \underset{(k,i) \in A}{x_{ki}} \begin{cases} = v & \text{if } i = S \\ = -v & \text{if } i = T \\ = 0 & \text{for all other } i. \end{cases} \quad (2.17)$$

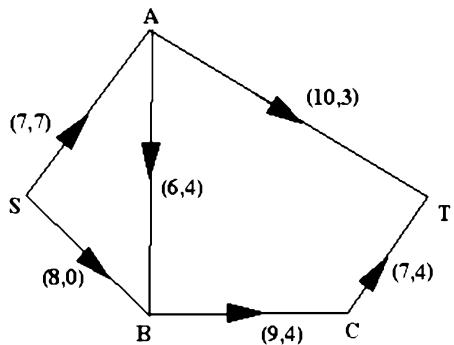
$$x_{ij} \leq u_{ij} \text{ for all } (i, j) \in A; \quad x_{ij} \geq 0. \quad (2.18)$$

Constraints (2.17) ensure that the total flow arriving at the sink and the flow leaving the source are both equal to the objective V , while at all other nodes the amount of flow entering the node is equal to that leaving.

As stated above, the problem could be solved by applying the simplex algorithm to the above formulation. However, a simpler and more intuitive algorithm is the Ford–Fulkerson labeling algorithm (Ford and Fulkerson 1956) which is based on the idea of flow-augmenting chains. Given a feasible flow in a network (i.e. a flow satisfying constraints (2.17) and (2.18)), a flow-augmenting chain from S to T is a chain of arcs (in any orientation) such that the flow can be increased in forward arcs and decreased in backward arcs. This concept will be illustrated with reference to Fig. 2.8.

The two-part labels on the arcs represent the capacity and the flow respectively. A total of seven units of flow travels from S to T. We can increase this flow along a chain of arcs in two ways. We could take chain $\{(S,B), (B,C), (C,T)\}$ made up entirely of forward arcs and increase the flow by three units. The limit of the increase

Fig. 2.8 Maximum flow principle



is three as any larger increase would violate the capacity of arc (C,T). Alternatively, we could take the chain $\{(S,B), (A,B), (A,T)\}$, in which arc (A,B) is a backward arc as it is oriented in the opposite direction to the chain. Using this chain we can increase the flow by four units by increasing the flow in arcs (S,B) and (C,T) and decreasing the flow in arc (A,B). This will have the effect of diverting four units of flow from (A,B) to (A,T), thus allowing an additional four units to arrive at B from S. The limit of four units derives from the fact that this will reduce the flow in (A,B) to zero. Ford and Fulkerson proved the result that a flow is optimal if and only if it has no flow-augmenting chain. This suggests that the maximum flow problem can be solved by repeatedly finding a flow-augmenting chain and augmenting the flows in the chain, until no flow-augmenting chain exists. The Ford–Fulkerson labeling algorithm provides a mechanism for doing this while guaranteeing to identify a flow-augmenting chain if it exists. It builds up one or more partial chains by successively labeling nodes with a two-part label (p_i, b_i) where p_i defines the predecessor of node i in the chain and b_i is an upper bound on the capacity of the chain up to node i . The objective is either to reach node T in which case a flow-augmenting chain has been found, or to terminate without reaching T, in which case no flow-augmenting chain exists.

2.5.2.2 The Ford–Fulkerson Labeling Algorithm

For maximum flow from source S to sink T.

Notation:

- x_{ij} is the current flow in arc (i, j)
- u_{ij} is the capacity of arc (i, j) .

Step 1. Find an initial feasible flow. (All flows = 0 will do.)

Find a flow-augmenting chain as follows.

Step 2. Label S $(-, \infty)$ and set all other nodes as unlabeled.

Step 3. Select a forward arc (i, j) from a labeled to an unlabeled node such that $u_{ij} - x_{ij} > 0$, or select a backward arc (j, i) from an unlabeled node to a labeled such that $x_{ij} > 0$.

If no such arc exists STOP current flow is maximal.

Step 4. If forward arc label j ($i, \min\{b_i, u_{ij} - x_{ij}\}$).

If backward arc label i ($-j, \min\{b_j, x_{ij}\}$).

Step 5. If T not labeled go to step 3.

Otherwise adjust flow as follows.

Step 6. Trace path back from T using labels p_i to determine preceding node.

Increase flows in forward arcs on the path by b_T and decrease flows in backward arcs on the path by b_T .

Step 7. Go to step 2.

We illustrate the algorithm with reference to Fig. 2.8. We start with the given flow:

Labeling: S($-, \infty$), B(S, $\min(\infty, 8 - 0) = 8$), C(B, $\min(8, 9 - 4) = 5$), T(C, $\min(5, 7 - 4) = 3$), $b_T = 3$.

Thus we can augment the flow by three units. Using the labels p_i and working back from p_T we get chain S, B, C, T. All arcs are forward so the flow is increased by three units in each to give (S, A) seven units, (S, B) three units, (A, B) four units, (A, T) three units, (B, C) seven units, (C, T) seven units.

Attempt to find a flow-augmenting chain given updated flows:

Labeling: S ($-, \infty$), B(S, 5), C(B, 2), A($-B$, 4), T(A, 4).

Note that in this small example we can see that labeling C will lead to a dead end. However, we have labeled it here to show that in the general case all labeled nodes need not appear in the final augmenting chain. The chain is given by SBAT where the link from B to A is backward so that flow is decreased by four units on this link and increased in all others. This gives: (S, A) seven units, (S, B) seven units, (A, B) zero units, (A, T) seven units, (B, C) seven units, (C, T) seven units.

Attempt to find a flow-augmenting chain given updated flows:

Labeling: S ($-, \infty$), B(S, 1), C(B, 1).

No further nodes are available for labeling. Thus the current flow of 14 units is optimal.

2.5.3 Minimum Cost Flow Problem

2.5.3.1 Introduction

Having introduced the basic concepts via the maximum flow problem we now move on to the more flexible model of the minimum cost flow in a closed network. This

problem consists of cyclic network, i.e. a network without a source and sink and has upper and lower bounds on the capacities of the arcs as well as a cost associated with each arc. The objective is to find a feasible flow in the network such that the total cost is minimized. The LP formulation to the problem is as follows.

Let x_{ij} , be the flow in arc (i, j) , u_{ij} the upper bound on arc (i, j) , l_{ij} the lower bound on arc (i, j) and c_{ij} the cost:

$$\min \sum_{(i,j) \in A} c_{ij}x_{ij} \text{ such that } \sum_{(i,j) \in A} x_{ij} - \sum_{(k,i) \in A} x_{ki} = 0 \forall i \quad (2.19)$$

$$x_{ij} \leq u_{ij}, \forall (i, j) \in A \quad (2.20)$$

$$x_{ij} \geq l_{ij} \forall (i, j) \in A \quad (2.21)$$

$$x_{ij} \geq 0 \forall (i, j) \in A$$

Constraint (2.19) ensures that the flow into each node equals that flowing out, while constraints (2.10) and (2.11) are the upper and lower bound constraints respectively. As with the maximum flow problem this problem can be solved using the simplex method, but there are also a number of specialist solution techniques. Here we introduce one of these, the out-of-kilter algorithm.

2.5.3.2 The Out-of-Kilter Algorithm

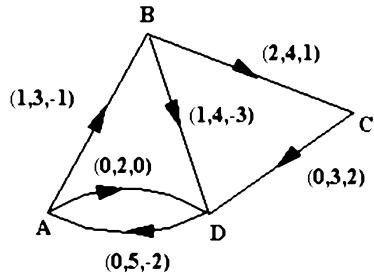
The derivation of the out-of-kilter algorithm ([Minty 1960; Fulkerson 1961](#)) is based on LP theory, but the algorithm can be implemented without any prior LP knowledge. We associate a real number $\pi(i)$ with each node i . These numbers are sometimes called *node potentials* and are simply the dual variables associated with the flow balancing constraints (2.9). LP theory states that the solution is optimal if and only if the following conditions are satisfied for each arc (i, j) :

Complementary slackness or kilter conditions:

- If $x_{ij} = l_{ij}$ then $c_{ij} + \pi(i) - \pi(j) > 0$
- If $l_{ij} < x_{ij} < u_{ij}$ then $c_{ij} + \pi(i) - \pi(j) = 0$
- If $x_{ij} = u_{ij}$ then $c_{ij} + \pi(i) - \pi(j) < 0$.

For a given arc we can represent these conditions diagrammatically by a two-dimensional plot in which x_{ij} is plotted on the x -axis and $c_{ij} + \pi(i) - \pi(j)$ is plotted on the y -axis. The set of all points satisfying the kilter conditions form two vertical lines defined by $x = l_{ij}$ for $y \geq 0$ and $x = u_{ij}$ for $y \leq 0$, connected by a horizontal line segment from $(l_{ij}, 0)$ to $(u_{ij}, 0)$. This is called the *kilter line* and the diagram is a *kilter diagram*. Figure 2.10 shows the six kilter diagrams relating to the arcs of the network in Fig. 2.9.

Fig. 2.9 Minimum cost flow problem (the three-part labels are (l_{ij}, u_{ij}, c_{ij}))



The bold lines are the kilter lines. The markers on the diagrams are plots of $(x_{ij}, c_{ij} + \pi(i) - \pi(j))$ for different flows and potentials. When the marker lies on the kilter line the corresponding arc is said to be *in kilter*, if not it is said to be *out of kilter*. We will refer to this figure again when we illustrate the out-of-kilter algorithm.

The out-of-kilter algorithm works with solutions that satisfy the flow balance constraints (2.19), but may violate the upper and lower bounds. By changing the flows or the potentials it gradually moves each arc into kilter without moving any other arc away from the kilter line in the process. It can be stated as follows.

Out-of-kilter algorithm for min. cost flow in a closed network:

Find an initial flow satisfying the flow balance equations and a set of node potentials $\pi(i) \forall i$. Let $y(i, j) = c_{ij} + \pi(i) - \pi(j)$ (note that all flows and potentials equal to 0 will do).

If $y(i, j) > 0$, $x_{\min}(i, j) = \min(x_{ij}, l_{ij})$, $x_{\max} = \max(x_{ij}, l_{ij})$.

If $y(i, j) = 0$, $x_{\min}(i, j) = \min(x_{ij}, l_{ij})$, $x_{\max}(i, j) = \max(x_{ij}, u_{ij})$.

If $y(i, j) < 0$, $x_{\min}(i, j) = \min(x_{ij}, u_{ij})$, $x_{\max}(i, j) = \max(x_{ij}, u_{ij})$.

While any arcs out of kilter and procedure is successful do.

Attempt to update flows.

Select and out-of-kilter arc (p, q) .

If $x(p, q) > x_{\min}(p, q)$ then set $s = p$, $t = q$, $v = x_{pq} - x_{\min}(p, q)$.

If $x(p, q) < x_{\max}(p, q)$ then set $s = q$, $t = p$, $v = x_{\max}(p, q) - x_{pq}$.

Attempt to find a flow-augmenting chain from s to t to carry up to v additional units of flow without using (p, q) and without exceeding $x_{\max}(i, j)$ in forward arcs or falling below $x_{\min}(i, j)$ in backward arcs.

Note: this can be achieved by starting the labeling algorithm with $s(-, v)$ and respecting x_{\max} and x_{\min} when adjusting the flows.

If successful increase flow in the chain and increase/decrease flow in (p, q) by b_t . Otherwise *attempt to update node potentials as follows.*

Let L be the set of all arcs (i, j) labeled at one end and not the other such that $l_{ij} \leq x_{ij} \leq u_{ij}$.

For those arcs in L labeled at i : if $y(i, j) > 0$ set $\delta_{ij} = y(i, j)$, otherwise set $\delta_{ij} = \infty$.

For those arcs in L labeled at j : if $y(i, j) < 0$ set $\delta_{ij} = -y(i, j)$, otherwise set $\delta_{ij} = \infty$.

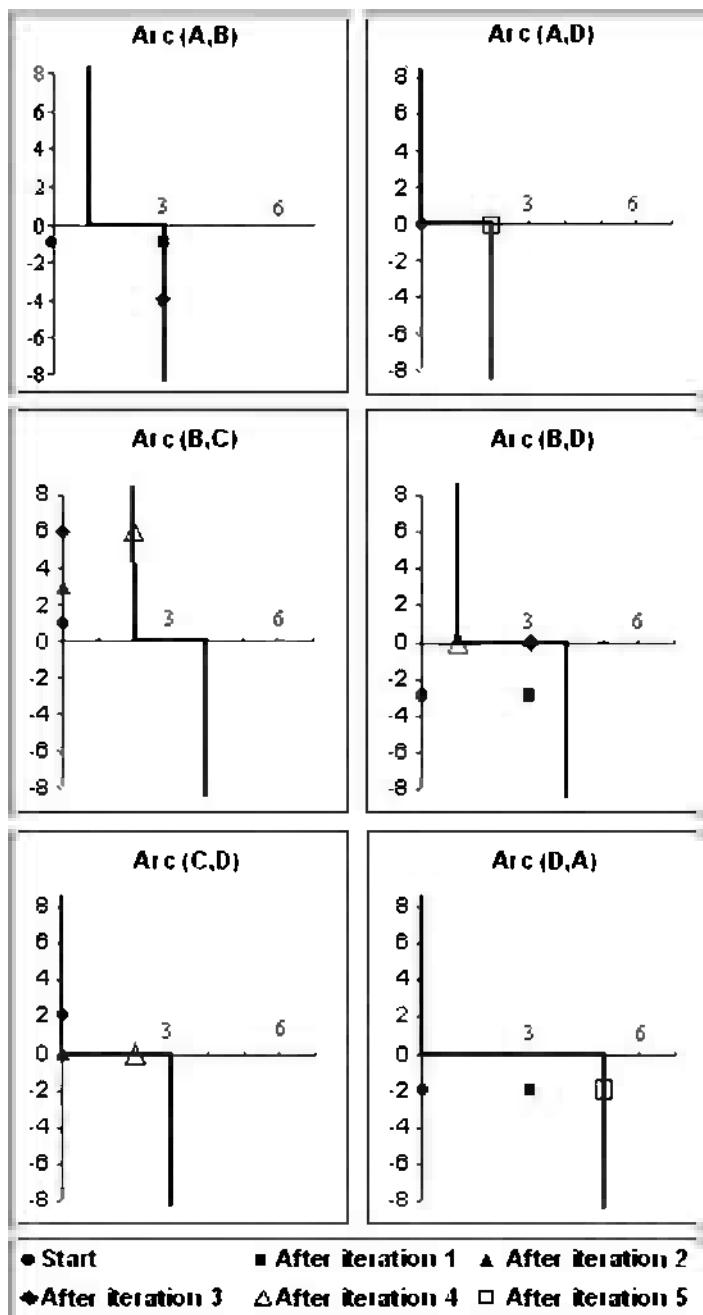


Fig. 2.10 Kilter diagrams for Fig. 2.9 problem

Set $\delta = \min\{\delta_{ij} : (i, j) \in L\}$.

If $\delta = 0$ then stop—no feasible flow.

Otherwise set $\pi(k) = \pi(k) + \delta$ for all unlabeled nodes and update $y(i, j)$ for all arcs labeled at one end and not the other.

Repeat.

When the algorithm terminates, either all the arcs are in kilter and the current flows are optimal or no feasible solution exists.

We illustrate the algorithm with reference to the network in Fig. 2.9 and the kilter diagrams in Fig. 2.10. Note that x_{\min} and x_{\max} are simply the minimum and maximum flow values that ensure that an arc never crosses or moves further away from the kilter line in a horizontal direction and δ serves the same purpose for moves in a vertical direction.

Initialization:

We start with all flows and potentials equal to zero. Thus $c_{ij} + \pi(i) - \pi(j)$ is simply the arc cost for all arcs. This situation is given by the solid circles in Fig. 2.10. Note that arcs (A, D) and (C, D) are already in kilter, but arc (D, A) is not, even though its flow lies within its lower and upper bounds.

Iteration 1:

We select out-of-kilter arc (A, B).

We would like to increase the flow in this arc by three units. Thus $v = 3$, $s = B$, $t = A$.

Labeling: B(−, 3), D(B, 3), A(D, 3).

Labeling has been successful. Therefore we increase flow in chain {(B, D), (D, A)} and in arc (A, B) by three units and update x_{\min} and x_{\max} ∨ updated arcs. This is shown by the solid squares in Fig. 2.10.

Note that arc (A, B) is now in kilter.

Iteration 2:

Select out-of-kilter arc (B, C), $s = C$, $t = B$, $v = 2$.

Labeling: C(−, 2) no further labeling possible as we cannot increase flow in (C, D) without moving away from the kilter line.

$L = \{(C, D)\}$ and as (C, D) is labeled at C $\delta_{CD} = 2$. Note that this is the maximum distance this arc can move down without leaving the kilter line.

Unlabeled nodes are A, B, D. Increase potentials on these nodes by two to give $\pi(A) = 2$, $\pi(B) = 2$, $\pi(C) = 0$, $\pi(D) = 2$. This will change the $y(i, j)$ values for arcs (B, C) and (C, D) as shown by the solid triangles. Note that for (C, D) this also changes x_{\max} .

Iteration 3:

We try again with arc (B, C), $s = C$, $t = B$, $v = 2$.

Labeling: C(−, 2), D(C, 2), A(D, 2) no further labeling possible as flow in (A, B) is at full capacity and decreasing flow in (B, D) will move away from the kilter line.

$L = \{(A, B), (B, D)\}$. $\delta_{AB} = \infty$, $\delta_{BD} = 3$, $\delta = 3$.

Unlabeled node B. Increase $\pi(B)$ by 3 giving $\pi(A) = 2$, $\pi(B) = 5$, $\pi(C) = 0$, $\pi(D) = 2$ and changing positions of arcs (A, B), (B, C) and (B, D) as given by the solid diamonds. Arc (B, D) is now in kilter and has a new value for x_{\min} .

Iteration 4:

We try again with arc (B, C) $s = C$, $t = B$, $v = 2$.

Labeling: C(–, 2), D(C, 2), B(–D, 2). t is labeled.

Adjust flows in chain {(C, D), (B, D)} and increase flow in (B, C) by two units as shown by the unfilled triangles. Arc (B, C) is now in kilter.

Iteration 5:

Select only remaining out-of-kilter arc (D, A), $s = A$, $t = D$, $v = 2$.

Labeling; A(–, 2), D(A, 2). t is labeled. Increase flow in (D, A) and (A, D) by two units as shown by the unfilled squares.

All arcs are in kilter, therefore solution is optimal. Flows are as given by the last update for each arc, i.e. $x_{AB} = 3$, $x_{AD} = 2$, $x_{BC} = 2$, $x_{BD} = 1$, $x_{CD} = 2$ and $x_{DA} = 5$, with a total cost of –13.

2.5.4 Other Issues

The previous sections introduced two relatively simple algorithms for the maximum flow and minimum cost flow problems. The out-of-kilter algorithm also has the advantage that it is easy to find an initial solution as the upper and lower bounds do not need to be satisfied. However, these are not necessarily the most efficient algorithms in each case. For example, it is possible to design pathological cases of the max flow problem for which the number of iterations made by the Ford–Fulkerson algorithm is only bounded by the capacity on the arcs. There are also inherent inefficiencies in the algorithms in that subsequent iterations may need to recalculate labels already calculated in previous iterations. A more efficient algorithm is the network simplex algorithm which can be found in [Ahuja et al. \(1993\)](#). The algorithm is so called because it can be shown that the iterations used to improve an initial solution correspond exactly to those computed by the simplex method, but the special structure of the problem means that the algorithm can be expressed purely in network terms without recourse to the simplex tableau. For large problems the additional efficiencies of the network simplex algorithm may pay off, but for small to moderate problems the out-of-kilter algorithm should be fast enough, and has the advantage that it is easy to implement from scratch and that code is available from a number of sources.

As already mentioned, the efficiency of network flow solution algorithms means that it is worthwhile attempting to model any new problem as a network flow problem. A wide range of problems can be modeled using the two formulations already given. However, the scope of network flow approaches is even wider when we consider problems that can be modeled using the dual formulations of network flow problems. Examples of this type of model include [Mamer and Smith's \(1982\)](#) approach to an infield repair kit problem and [Johnson's \(1968\)](#) open-cast mine plan-

ning problem. Even when the network model is not flexible enough to incorporate all the constraints in a problem it may still provide an effective optimization tool. For example [Glover et al. \(1982\)](#) model the problem of determining the allocation of space on aircraft to different fare structures as a minimum cost flow problem. Their model does not incorporate all the constraints needed to model the problem and thus infeasible solutions may be returned. If this occurs, the solution is excluded from the model and the solution process reiterated until a feasible solution results. The authors comment that this was far more efficient than attempting to include all such constraints directly into an IP model.

We have limited our focus to problems in which flows are homogeneous and there is no gain or leakage of flow along an arc. Problems in which multiple commodities share the arc capacities and problems where flow is not preserved along an arc have also been widely studied. Unlike the simple problems covered here both these problems are NP-complete. Algorithms for solving such problems are beyond the scope of this chapter but can be found in many network flow texts. [Ahuja et al. \(1993\)](#) give a comprehensive treatment of network flows including models, algorithms and practical applications.

2.6 Some Useful Models

The previous sections have outlined three general approaches to optimization problems. In this section we focus on two classes of problem that frequently occur as subproblems in the solution of larger or more complex problems over a wide range of application areas. In each case solution approaches based on the methods covered in the previous sections are described.

2.6.1 Shortest-Path Problems: DP Approaches

The first class of common subproblems are those involving shortest paths. As observed in Sect. 2.3.2, the branch-and-bound shortest-path algorithm presented there is not the most efficient way of tackling such problems. In this section more efficient approaches based on DP are presented. We start by considering the problem of finding the shortest path from a source vertex s to all other vertices in a graph.

2.6.1.1 Bellman's Shortest-Path Algorithm

This can be solved by Bellman's shortest-path algorithm in which the stages are defined by the number of links allowed in the path and the states are defined by the vertices. The formulae for the starting state and the recursive relationship can be defined as follows:

$$F_0(v) = 0 \text{ if } v = s, \quad F_0(v) = \infty \text{ otherwise,}$$

$$F_k(v) = \min \left\{ F_{k-1}(v) \min_{w \in Q_{k-1}, (w,v) \in E} F_{k-1}(w) + c_{vw} \right\},$$

where Q_k is the set of vertices whose values were updated at stage k and E is the set of links in the network.

Bellman's shortest-path algorithm can then be stated as follows:

```

Set  $F_0(v) \forall v \in V$ 
While  $Q_k \neq \emptyset$  and  $k \leq n$  do
  Calculate  $F_k(v) \forall v \in V$  and determine  $Q_k$ 
End while

```

If $Q_k = \emptyset$ then $F_k(v)$ defines the length of the shortest path from s to $v \forall v$.

If $k = n$ and $Q_k \neq \emptyset$ then the network contains negative cost circuits and shortest paths cannot be defined.

If paths from every vertex to every other are required then rather than execute Bellman's algorithm n times it is more efficient to use Floyd's shortest-path algorithm ([Floyd 1962](#)).

2.6.1.2 Floyd's Shortest-Path Algorithm

This is also a DP type approach but in this case $F_k(i, j)$ represents the shortest path between i and j allowing only vertices 1 to k as intermediate points. The initial states are given by $F_0(i, j) = C_{ij}$, where C_{ij} is the cost of link (i, j) , and the recursive relationship by $F_k(i, j) = \min\{F_{k-1}(i, j), F_{k-1}(i, k) + F_{k-1}(k, j)\}$. As $F_k(i, i) = 0 \forall k$ unless the network contains a negative cost circuit neither $F_k(i, k)$ or $F_k(k, j)$ will be updated during iteration k . Therefore the subscript k is usually dropped in practice and matrix $F(i, j)$ is overwritten at each iteration. The algorithm can then be stated as follows.

Floyd's algorithm for the shortest path between all pairs of vertices in an arbitrary graph:

Step 1

$$k = 0, F(i, j) = c(i, j) \forall (i, j)$$

Step 2

$$\forall k = 1, n$$

$$\forall i = 1, n \text{ s.t. } i \neq k \text{ and } F(i, k) \neq \infty$$

$$\forall j = 1, n \text{ s.t. } j \neq k \text{ and } F(k, j) \neq \infty$$

$$F(i, j) = \min\{F(i, j), F(i, k) + F(k, j)\}$$

if $F(i, i) < 0$ for any i STOP. (*negative cost circuit detected.*)

end loops

It should be noted that if all costs are non-negative then an algorithm due to Dijkstra (1959) is more efficient than Bellman's algorithm. Dijkstra's algorithm can be found in most basic texts covering graph and network algorithms (e.g. Ahuja et al. 1993).

2.6.2 Transportation Assignment and Transhipment Problems: Network Flow Approaches

In this section we consider a second commonly occurring class of subproblems: the family of transportation type problems, including the assignment and transhipment problems. All three problems can be modeled as minimum cost flow problems. Once the appropriate model has been derived, solutions can be obtained using the out-of-kilter algorithm or any other minimum cost flow algorithm. The focus of this section is therefore on defining appropriate models.

2.6.2.1 The Transportation Problem

The transportation problem is that of determining the amount of product to be supplied from each of a given set of supply points to each of a given set of demand points, given upper bounds on availability at each supplier, known demands at each demand point and transportation costs per unit supplied by supplier i to demand point j . The problem can be formulated as follows:

$$\begin{aligned} \min & \sum_{i=1}^n \sum_{j=1}^m c_{ij} x_{ij} \\ \text{s.t. } & \sum_{i=1}^n x_{ij} \geq d_j \\ & \sum_{j=1}^m x_{ij} \leq s_i. \end{aligned} \tag{2.22}$$

We can model this problem as a minimum cost flow network by defining nodes, arcs, lower and upper bounds and costs as follows:

Nodes:

A dummy source node S.

A dummy sink node T.

One node for each supplier, $i = 1, n$.

One node for each demand point j , $j = 1, m$.

Arcs:

An arc from each supplier node i to each demand node j with lower bound = 0, upper bound = s_i and cost = c_{ij} . Note if all $c_{ij} \geq 0$ then the upper bound can be replaced by $\min\{s_i, d_j\}$.

An arc from S to each supplier node with lower bound = 0, upper bound = s_i and cost = 0.

An arc from each demand node, j , to T with lower bound = d_j upper bound = M , where M is some suitably large number and cost = 0. Note if all $c_{ij} \geq 0$ then the upper bound can be replaced by d_j .

An arc from T to S with lower bound = 0, upper bound = $\sum_{i=1}^n s_i$, cost = 0.

The minimum cost flow in the network defined above will give the optimal solution to the transportation problem, and the flows in the arcs (i, j) define the value of the variables x_{ij} .

Many management science and operational research texts cover the stepping-stone algorithm or MODI for the transportation problem. It is interesting to note that the rules used by the MODI method for determining those x_{ij} that should be considered for increase are precisely the kilter conditions that define those arcs that lie to the left of the kilter line in the network flow model, and that the stepping-stone process for finding a suitable path to update the solution corresponds to finding a flow-augmenting chain. Many texts also suggest a heuristic method known as Vogel's approximation method to obtain an initial solution. This approach can be taken to find an initial flow in the above model, thus reducing the number of iterations required when compared with a starting solution of 0.

2.6.2.2 The Assignment and Transhipment Problems

Several relatives of the transportation problem are also encountered as subproblems. Here we consider the assignment and transhipment problems.

The assignment problem is that of assigning tasks to resources. It is assumed that there are n tasks and n resources and a cost c_{ij} associated with the assignment of task i to resource j . The objective is to assign each task to exactly one resource and each resource to exactly one task such that the total cost of the assignments is minimized. This problem can be regarded as a transportation problem in which $n = m$ and $s_i = d_j = 1 \forall i$ and $\forall j$. Thus any assignment problem can be modeled using the transportation model above with the appropriate values for s_i and d_j .

The transhipment problem is a transportation problem in which goods may travel from supplier to demand point via any or none of a set of intermediate points—known as transhipment points. These points may also be supply points or demand points in their own right. The network flow model for the transportation problem can easily be adapted to the transhipment model by adding new nodes for the transhipment points and adding arcs from each supply point and to each demand point with appropriate bounds and transhipment costs. If a transhipment point is also a supply point or demand point then it is also connected to S or T respectively.

2.6.2.3 Other Useful Models

The above classes of problem are two of the most frequently occurring subproblems that can be solved using the methods covered in this chapter. Two other classes of

problem that deserve special mention are the binary and bounded knapsack problems and matching problems in bipartite graphs.

We have already presented a DP approach for the unbounded knapsack problem in Sect. 2.3. The binary version of the problem occurs when at most one piece of each type is available. An effective solution approach to this problem is a tree search in which there are two branches from each node, one corresponding to fixing the value of a variable x_i at 0 and the other to fixing the same variable at 1. Upper and lower bounds are easily obtained, simply by sorting the variables in v_i/w_i order and fitting and adding each to the knapsack in turn, until the capacity exceeded. In its basic form the algorithm is an implementation of the standard Integer Programming branch-and-bound algorithm as discussed in Chap. 3. However, the bounds can be improved by adding problem specific information. [Martello and Toth \(1990\)](#) give details of several variants of the method and suggest a series of increasingly powerful bounds. They also show how the method can be extended to the more general case in which the variables are not binary, but are restricted by upper bounds.

Matching problems also occur widely. A matching in a graph is a set of edges no two of which have a vertex in common. Maximum matching problems in arbitrary graphs can be solved using the blossom algorithm due to Edmonds, and described in [Ahuja et al. \(1993\)](#). However, if the graph is bipartite, i.e. the vertices can be partitioned into two subsets such that there are no edges between any two vertices in the same subset, then the problem can be solved more simply using network flow type algorithms. Such models are useful in practice as bipartite graphs often appear in allocation or scheduling problems.

2.7 Promising Areas for Future Application

In this section we outline some potential areas for the future application of the methods described in this chapter. All three approaches have been used extensively in the solution of a broad range of problems for several decades. The increase in computer power available to individuals and organisations since their introduction has lead to a continuously expanding range of problems that can be solved to optimality within a feasible amount of time. At the same time theoretical advances in potential application areas have lead to improved bounds, once again increasing the scope of branch-and-bound approaches. There are currently many researchers active in each of the areas. Thus it is likely that we will continue to see new theoretical developments as well as new branch-and-bound or DP implementations for old problems and the application of classical approaches to new practical problems. However, there is also considerable potential for combining these techniques with some of the more modern approaches covered elsewhere in this book. One option is to enhance the performance of a tree search algorithm by using a solution obtained by a powerful heuristic to provide a good upper bound. Alternatively the techniques described in both this chapter and Chap. 3 can be used to enhance the performance of a heuristic approach. As evidenced by the survey article of [Fernandes and Lourenço \(2007\)](#), this has developed into a popular and promising area of research. Such inte-

gration can be at one of three levels: pre- or post-processing, true hybridization and cross-fertilization of ideas. We take a brief look at each of these in turn.

2.7.1 Pre- and Post-processing

The simplest form of integration is to use a classical approach as part of a staged solution to a problem. Tree search approaches are often used to enumerate all the variables required for the optimization phase of the problem. For example in crew scheduling problems the set of feasible tours of duty satisfying all the necessary constraints are enumerated first and then these are used as input to an optimization algorithm, or in the case of timetabling and scheduling problems the allocation of events to rooms may be carried out in a post-processing phase once the schedule has been determined. In other cases pre-processing may be used to reduce the size of the solution space. For example, [Dowsland and Thompson \(2000\)](#) solve a nurse scheduling problem using tabu search. Before calling the tabu search routine they use a tree search approach to solve a modified knapsack problem that enables them to determine the precise number of additional nurses required to cover the weekly demand on a ward. This allows them to minimize the size of the solution space and to simplify the evaluation function in the tabu search part of the solution process.

2.7.2 True Hybrids

A greater degree of integration is provided by true hybrid approaches in which a classical approach is embedded into a modern search tool or vice versa. Many researchers have tried this. There are several instances of the integration of branch and bound with genetic algorithms. For example [Cotta et al. \(1995\)](#) use a tree search to find the best child from a set of possibilities given by a loosely defined crossover, while [Nagar et al. \(1995\)](#) use a genetic algorithm to search out a promising set of ranges on the values of a series of variables and then use a tree search to find the optimal solution within the range. Classical techniques have also been embedded into neighborhood search approaches such as simulated annealing, tabu search and variable-depth search. For example, there are many problems in which the variables can be partitioned into two sets, A and B , such that if the values of A are fixed the problem of optimizing B reduces to a network flow problem. The size of the solution space can be reduced to cover only the variables in A , with each solution being completed by solving for the variables in B . Network flow type problems are especially amenable to this sort of role, as neighborhood moves will typically involve changing a cost or bound, or adding or deleting an arc. All of these changes can be accommodated in the out-of-kilter algorithm using the previous solution to initialize the solution process for its neighbor(s), thus minimizing the computational effort in resolving each new subproblem. Examples of this type of strategy include

[Hindi et al. \(2003\)](#) who solve transhipment subproblems to complete solutions in their variable depth search approach to the lot sizing problem, and [Dowsland and Thompson \(2000\)](#) who use a network flow problem to allocate nurses on days to the morning or afternoon shifts. This reduces the number of variables in the search space for each nurse by up to a factor of 32.

As discussed in other chapters, many neighborhood searches can be improved by extending the size of the neighborhood. One way of doing this is to introduce chains of moves. Rather than select a chain at random or enumerate all chains in the neighborhood, it makes sense to find optimal or improving chains directly using some form of shortest-path algorithm. For example, [Dowsland \(1998\)](#) uses a mixture of tree search and Floyd's algorithm to find improving chains of moves for a nurse scheduling problem. More recently, [Abdullah et al. \(2007\)](#) have used a similar strategy in a tabu search solution to a timetabling problem. Other forms of compound moves may not involve chains. In such cases other techniques can be used to find optimal moves. For example, [Potts and van de Velde \(1995\)](#) use DP to search for good moves in neighborhoods made up of multiple swaps for the TSP. [Gutin \(1999\)](#) also considers the TSP but utilizes a bipartite matching problem in order to determine the best option from a neighborhood defined by removing and reinserting k vertices in the tour. Other researchers have worked with this neighborhood for different definitions of k . Further examples can be found in [Ahuja et al. \(2002\)](#).

2.7.3 Cross-fertilization

As well as true hybrids such as those outlined above there are also examples of cross-fertilization of ideas in which an ingredient from a classical approach has been embedded into a more modern method. One example is the incorporation of bounds into a heuristic search in order to avoid or leave non-promising areas of the solution space. [Hindi et al. \(2003\)](#) use this strategy in their variable depth approach to avoid wasting time in solving transhipment problems to complete solutions that can be shown to be bounded, while ([Dowsland 1998](#)) uses bounds in combination with a tabu list to avoid wasting time in areas of the search space that cannot lead to a better solution than the best found so far. Similar approaches have been taken with genetic algorithms by [Tamura et al. \(1994\)](#) and [Dowsland et al. \(2006\)](#) who use mutation and crossover respectively to destroy partial solutions that exceed a tree search type bound. A second example of cross-fertilization is in the use of ejection chains, a concept suggested by [Glover and Laguna \(1997\)](#) which is a generalization of alternating chains—a term used for the way in which flow is updated in certain classes of network flow problems.

2.8 Tricks of the Trade

2.8.1 Introduction

For newcomers to the field the prospect of applying any of the above techniques to a given problem can appear daunting. This section suggests a few tips on overcoming this feeling and getting started on a basic implementation, and then going on to identify possible areas for algorithm improvement. We start with a few general observations that apply to all three techniques and follow this with more specialist advice for each of the techniques in turn.

1. *Get a basic understanding of the technique and how it might be applied to a given problem.* This involves reading suitable books and articles. Due to the relatively long history of the techniques covered in this chapter there is a wealth of introductory material available. Although some of the seminal material and/or early texts in each field provide valuable insights and technical detail they often use specialist terminology and can be difficult to understand. Therefore they are probably best left until some practical experience has been gained, and the best place to start would be one of the more up-to-date texts given in the References and Sources of Additional Information at the end of this chapter. These will provide a thorough background but may not include examples that are closely related to the reader's own problem. It is therefore desirable to supplement these sources with journal articles related to the relevant application area.
2. *Don't reinvent the wheel.* There are many published articles describing implementations of these techniques to classical combinatorial optimization problems. If your problem can be formulated as, or is closely related to, one of these problems then it is likely that an effective implementation has already been published, and in many cases suitable code may also be readily available. Sometimes the relationship to a classical problem is obvious from the problem definition, but this is not always the case. It is therefore well worth considering different ways of modeling a problem e.g. using graph-theoretic models or trying different LP type formulations and comparing these with well-known classical problems.
3. *Don't be too pessimistic.* Although the complexity of most branch and bound and dynamic programming algorithms is likely to be exponential (or at best pseudo-polynomial) they can still be effective tools for solving moderately sized problems to optimality, and may well compete with the more modern methods described later in this book when used as heuristics. While there may be many good practical reasons for selecting a heuristic rather than an exact approach to a problem, many real-life NP-hard problems have proved amenable to solution by both branch and bound and dynamic programming. However, this may require a little ingenuity on the part of the algorithm designer. If the initial implementation seems too slow or memory intensive it is well worth spending some time and effort trying to make improvements. The situation with regard to network flow programming is somewhat different in that the simple models covered in detail in

this chapter can all be solved to guaranteed optimality in polynomial time and are the obvious first choice approaches for problems that have the required structure.

2.8.2 *Tips for Branch and Bound*

The issues arising in developing and implementing a branch and bound approach can be broadly divided into three categories:

1. *Representations.* For most problems there are several different potential tree search representations and search strategies each of which will impact differently on solution time and quality. In order to appreciate this it is worth reading articles that use different representations for the same problem. Before starting to code any implementation think about what you have read in relation to your own problem. You should also think about whether you want to apply a depth-first search or if there may be advantages in a more memory-intensive breadth or best-first search.
2. *Coding.* Although it is relatively easy for a beginner to construct a tree search on paper it can be difficult to translate this into computer code. We therefore recommend finding code or detailed pseudo-code for a similar tree structure to the one you are planning on and using it as a template for your own program structure. It is also often helpful to code the three operations of branching, backtracking and checking the bounding conditions separately. It is also a good idea to start with very simple bounds and a branching strategy based on a natural ordering of the vertices and test on a small problem, building up more sophisticated bounds and branching strategies as necessary.
3. *Performance.* Once the basic code is working the following pointers will help in getting the best out of an implementation.

Analyze the time taken to calculate the bounds and their effectiveness in cutting branches and consider how the trade-off influences overall computation time. While computationally expensive bounds or branching strategies are likely to increase solution times for small problems they may well come into their own as problem size grows.

In the same way as it is important to test heuristics for solution quality on problem instances that closely match those on which they are to be used in terms of both problem size and characteristics, so it is important to ensure that a branch and bound approach will converge within a realistic time-scale on typical problem instances.

Remember that branch and bound can be used as a heuristic approach, either by stopping after a given time or by strengthening the bound to search for solutions that improve on the best so far by at least $\alpha\%$. In this case it is important to consider whether you want to bias the search towards finding good solutions early or precipitating the bounding conditions earlier.

2.8.3 Tips for Dynamic Programming

Due to its generality, getting started with dynamic programming can be difficult but the following pointers may be of assistance:

1. *Representation.* When faced with a new problem simply deciding on the definition of stages and states that form the basis of a correct dynamic programming formulation can be difficult, and access to a successful formulation to a similar problem can be invaluable. As a starting point it is worth considering if the problem can be classified as a multi-period optimization problem, routing problem, or knapsack type problem. If so then there are a wealth of examples in standard texts or journal articles that should help.

Remember that the objective is to produce something that is considerably more efficient than complete enumeration. Therefore it is important to ensure that the calculations relating to the first stage are trivial, and that there is a relatively simple recursive relationship that can be used to move from one stage to the next. If there is no apparent solution when defining the stages in a forwards direction then consider the possibility of backward recursion.

2. *Performance.* DP can be expensive both in terms of computational time and storage requirements and the main cause of this is the number of states. Therefore once a basic structure has been determined it is worthwhile considering ways of cutting down on the number of states that need to be evaluated. It is also worthwhile ensuring that only those stages/states that may be required for future reference be stored. If the environment requires the solution of several similar problems—e.g. if the DP is being used to calculate bounds or optimize large neighborhoods—consider whether or not the different problems could all be regarded as subproblems of one large problem, thereby necessitating just one DP to be solved as a pre-processing stage. As with branch and bound it is important to ensure that the time and memory available are sufficient for typical problem instances.

2.8.4 Tips for Network Flow Programming

The problems facing a beginner with network flow programming are different in that there are standard solution algorithms available off-the-peg, so that the only skill involved is that of modeling. Because these algorithms are readily available and operate in polynomial time then it is certainly worth considering whether any new problem might be amenable to modeling in this way. Some tips for recognizing such problems are given below.

If the problem involves physical flows through physical networks then the model is usually obvious. However, remember that network flow models simply define an abstract structure that can be applied to a variety of other problems. Typical pointers to possible network flow models are allocation problems (where flow from i to j

represents the fact that i is allocated to j), sequencing problems (where flow from i to j represents the fact that i is a predecessor of j), and problems involving the selection of cells in matrices where flow from $(i$ to j represents the selection of cell (i, j)). However, this list is by no means exhaustive.

Other tips for modeling problems as network flows are to remember that network links usually represent important problem variables and that even if the given problem does not have an obvious network flow structure the dual problem might. In addition, although typical practical problems will be too large and complex to draw the whole network it is worthwhile making a simplified sketch of potential nodes and links. If the problem has complexities such as hierarchies of resources, multiple pricing structures etc. this may be facilitated by simplifying the problem first and then trying to expand it without compromising the network flow structure.

2.9 Conclusions

Although the classical techniques described in this chapter were developed to meet the challenges of optimization within the context of the computer technology of the 1950s they are still applicable today. The rapid rate of increase of computing power per unit cost in the intervening years has obviously meant a vast increase in the size of problems that can be tackled. However, this is not the sole reason for the increase. Research into ways of improving efficiency has been continuous both on a problem-specific and generic basis—for example all the techniques lend themselves well to parallelization. Nevertheless, they do have drawbacks. The scope of network flow programming is limited to problems with a given structure, while the more general methods of branch and bound and DP may require vast amounts of computer resource. DP solutions to new problems are often difficult to develop, and it may not be easy to find good bounds for a branch-and-bound approach to messy practical problems. Hence the need for the more recent techniques described elsewhere in this volume.

However, it should be noted that these techniques have not eclipsed the classical approaches, and there are many problems for which one of the techniques described here is still the best approach. Where this is not the case and a more modern approach is appropriate it is still possible that some form of hybrid may enhance performance by using the strengths of one approach to minimize the weaknesses of another.

Sources of Additional Information

Basic material on dynamic programming and network flows can be found in most undergraduate texts in management science and OR. Their treatment of branch and bound tends to be limited to integer programming. The generic version of branch

and bound covered in this chapter can be found in most texts on combinatorial algorithms/optimization, or in subject-oriented texts (e.g. algorithmic graph theory, knapsack problems).

Below are a small sample of relevant sources of information:

- <http://www2.informs.org/Resources/> (INFORMS OR/MS Resource Collection—links to B&B, DP and Network Flow sources)
- <http://people.brunel.ac.uk/mastjeb/jeb/or/contents.html> (J. Beasley, Imperial College)
- <http://jorlin.scripts.mit.edu/> (J. Orlin, MIT)
- <http://math.illinoisstate.edu/sennott/> (Sennott 1998)
- <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/> (Lectures and Videos on DP and other algorithms)
- http://en.wikipedia.org/wiki/Dynamic_programming
- http://en.wikipedia.org/wiki/Branch_and_bound
- Bather, J. (2000). Decision Theory: An Introduction to Dynamic Programming and Sequential Decisions. Wiley.
- Bellman, R. (2003). Dynamic Programming. Dover Publications.
- Christofides, N. (1975). Graph Theory—An Algorithmic Approach. Academic Press.
- Hu, T. C. (1981). Combinatorial Algorithms. Addison-Wesley.
- Lawler, E. L. et al. (1990). The Travelling Salesman Problem. Wiley.
- Nemhauser, G. L., Wolsey, L. A. (1988). Integer and Combinatorial Optimisation. Wiley.
- Papadimitriou, C. H. (1982). Combinatorial Optimisation: Algorithms and Complexity. Prentice-Hall.
- Reingold, E. M. et al. (1977). Combinatorial Algorithms. Prentice-Hall.
- Sennott, L. I. (1998). Stochastic Dynamic Programming, Wiley.
- Taha, H. A. (2002). Operations Research. Prentice-Hall.

References

- Abdullah S, Ahmadi S, Burke EK, Dror M and Mc Collum B (2007) A tabu-based large neighbourhood search methodology for the capacitated examination timetabling problem. *J Oper Res Soc* 58:1494–1502
- Ahuja RK, Magnanti TL, Orlin JB (1993) Network flows: theory, algorithms and applications. Prentice-Hall, Englewood Cliffs
- Ahuja RK, Ergun O, Orlin JB, Punnen AO (2002) A survey of very large-scale neighbourhood search techniques. *Discret Appl Math* 123:75–102
- Anderson DR, Sweeney DJ, Williams TA (1997) Introduction to management science: quantitative approaches to decision making. West Publishing, Eagan
- Balakrishnan VK (1997) Schaum's outline of graph theory (Schaum's Outline Series). Schaum Publishers, Mequon

- Balas E, Christofides N (1981) A restricted Lagrangian approach to the travelling salesman problem. *Math Prog* 21:19–46
- Beasley JE (1985) An exact two-dimensional non-guillotine cutting tree-search procedure. *Oper Res* 33:49–64
- Bellman R (1957) Dynamic programming. Princeton University Press, Princeton
- Bouzaher A, Braden JB, Johnson GV (1990) A dynamic programming approach to a class of non-point source pollution control problems. *Manage Sci* 36:1–15
- Bron C, Kerbosch J (1973) Finding all cliques of an un-directed graph—alg 457. *Commun ACM* 16:575–577
- Brown JR (1972) Chromatic scheduling and the chromatic number problem. *Manage Sci* 19:456–463
- Christofides N, Whitlock C (1977) An algorithm for two-dimensional cutting problems. *Oper Res* 25:30–44
- Clarke SR, Norman JM (1999) To run or not?: some dynamic programming models in cricket. *J Oper Res Soc* 50:536–545
- Cotta C, Aldana JF, Nebro AJ, Troya JM (1995) Hybridising genetic algorithms with branch and bound techniques for the resolution of the TSP. In: Poras CC et al (eds) Proceedings of the international conference on artificial neural networks and genetic algorithms, Ales, pp 277–280
- Dantzig GB (1951) Maximization of a linear function of variables subject to linear inequalities. In: Koopmans TC (ed) Activity analysis of production and allocation. Wiley, New York
- Dijkstra EW (1959) A note on two problems in connection with graphs. *Numer Math* 1:269
- Dowsland KA (1987) An exact algorithm for the pallet loading problems. *EJOR* 31:78–84
- Dowsland KA (1998) Nurse scheduling with tabu search and strategic oscillation. *EJOR* 106:393–407
- Dowsland KA, Thompson JM (2000) Solving a nurse scheduling problem with knapsacks, networks and tabu search. *J Oper Res Soc* 51:825–833
- Dowsland KA, Herbert EA, Kendall G (2006) Using tree search bounds to enhance a genetic algorithm approach to two rectangle packing problems. *EJOR* 168:390–402
- Erlenkotter D (1978) A dual-based procedure for uncapacitated facility location. *Oper Res* 26:992–1009
- Fernandes S, Lourenço HR (2007) Hybrids combining local search heuristics with exact algorithms. In: Rodriguez F, Mélian B, Moreno JA, Moreno JM (eds) Proc V Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB’2007, Tenerife, 14–16 Feb 2007, pp 269–274
- Findlay PL, Kobbacy KAH, Goodman DJ (1989) Optimisation of the daily production rates for an offshore oil field. *J Oper Res Soc* 40:1079–1088
- Fisher ML (1985) An applications oriented guide to Lagrangian relaxation. *Interfaces* 15:10–21
- Floyd RW (1962) Algorithm 97—shortest path. *Commun ACM* 5:345

- Ford LR, Fulkerson DR (1956) Maximal flow through a network. *Can J Math* 18:399–404
- Fulkerson DR (1961) An out-of-kilter method for minimal cost flow problems. *SIAM J Appl Math* 9:18–27
- Garfinkel RS, Nemhauser GL (1969) The set partitioning problem: set covering with equality constraints. *Oper Res* 17:848–856
- Glover F, Laguna M (1997) Tabu search. Kluwer, Dordrecht
- Glover F, Glover R, Lorenzo J, Mcmillan C (1982) The passenger mix problem in the scheduled airlines. *Interfaces* 12:73–79
- Golumbic MC (1980) Algorithmic graph theory and perfect graphs. Academic, New York
- Gutin GM (1999) Exponential neighbourhood local search for the travelling salesman problem. *Comput OR* 26, 313–320
- Hayes M, Norman JM (1984) Dynamic programming in orienteering—route choice and the siting of controls. *J Oper Res Soc* 35:791–796
- Held M, Karp RM (1970) The travelling salesman problem and minimum spanning trees. *Oper Res* 18:1138–1162
- Hindi KS, Fleszar K, Charalambous C (2003) An effective heuristic for the CLSP with setup times. *J Oper Res Soc* 54:490–498
- Jarvinen P, Rajala J, Sinervo H (1972) A branch and bound algorithm for seeking the p-median. *Oper Res* 20:173
- Johnson TB (1968) Optimum pit mine production scheduling. Technical report, University of California, Berkeley
- Kamarkar NK (1984) A new polynomial-time algorithm for linear programming. *Combinatorica* 4:373–395
- Khachiyan LG (1979) A polynomial algorithm in linear programming. *Dokl Akad Nauk SSSR* 244:1093–1096 (in Russian) (English transl.: Sov Math Dokl 20:191–194(1979))
- Little JDC, Murty KG, Sweeney DW, Karel C (1963) An algorithm for the travelling salesman problem. *Oper Res* 11:972–989
- Martello S, Toth P (1981) A branch and bound algorithm for the zero-one multiple knapsack problem. *Discret Appl Math* 3:275–288
- Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. Wiley, New York
- Mamer JW, Smith SA (1982) Optimising field repair kits based on job completion rate. *Manage Sci* 28:1328–1334
- Minty GJ (1960) Monotone networks. *Proc R Soc* 257A:194–212
- Nagar A, Heragu SS, Haddock J (1995) A meta-heuristic algorithm for a bi-criteria scheduling problem. *Ann OR* 63:397–414
- Potts CN, van de Velde SL (1995) Dynasearch—iterative local improvement by dynamic programming. Part 1: the TSP. Technical report, University of Twente
- Ross GT, Soland RM (1975) A branch and bound algorithm for the generalised assignment problem. *Math Prog* 8:91–103

- Tamura H, Hirahara A, Hatono I, Umano M (1994) An approximate solution method for combinatorial optimisation—hybrid approach of genetic algorithm and Lagrangian relaxation method. *Trans Soc Instrum Control Eng* 130:329–336
- Zykov AA (1949) On some properties of linear complexes. *Math Sb* 24:163–188

Chapter 3

Integer Programming

Robert Bosch and Michael Trick

3.1 Introduction

Over the last 25 years, the combination of faster computers, more reliable data and improved algorithms has resulted in the near-routine solution of many integer programs of practical interest. Integer programming models are used in a wide variety of applications, including scheduling, resource assignment, planning, supply chain design, auction design, and many, many others. In this tutorial, we outline some of the major themes involved in creating and solving integer programming models.

The foundation of much of analytical decision making is linear programming. In a linear program, there are *variables*, *constraints*, and an *objective function*. The variables, or decisions, take on numerical values. Constraints are used to limit the values to a feasible region. These constraints must be linear in the decision variables. The objective function then defines which particular assignment of feasible values to the variables is optimal: it is the one that maximizes (or minimizes, depending on the type of the objective) the objective function. The objective function must also be linear in the variables.

Linear programs can model many problems of practical interest, and modern linear programming optimization codes can find optimal solutions to problems with hundreds of thousands of constraints and variables. It is this combination of modeling strength and solvability that makes linear programming so important.

Integer programming adds additional constraints to linear programming. An integer program begins with a linear program, and adds the requirement that some or all of the variables take on integer values. This seemingly innocuous change greatly

R. Bosch
Oberlin College, Oberlin, OH, USA

M. Trick (✉)
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: trick@cmu.edu

increases the number of problems that can be modeled, but also makes the models more difficult to solve. In fact, one frustrating aspect of integer programming is that two seemingly similar formulations for the same problem can lead to radically different computational experience: one formulation may quickly lead to optimal solutions, while the other may take an excessively long time to solve.

There are many keys to successfully developing and solving integer programming models. We consider the following aspects:

- Be creative in formulations
- Find integer programming formulations with a strong relaxation
- Avoid symmetry
- Consider formulations with many constraints
- Consider formulations with many variables
- Modify branch-and-bound search parameters.

To fix ideas, we will introduce a particular integer programming model, and show how the main integer programming algorithm, branch-and-bound, operates on that model. We will then use this model to illustrate the key ideas to successful integer programming.

3.1.1 Facility Location

We consider a facility location problem. A chemical company owns four factories that manufacture a certain chemical in raw form. The company would like to get into the business of refining the chemical. It is interested in building refining facilities, and it has identified three possible sites. The table below contains variable costs, fixed costs, and weekly capacities for the three possible refining facility sites, and weekly production amounts for each factory. The variable costs are in dollars per week and include transportation costs. The fixed costs are in dollars per year. The production amounts and capacities are in tonnes per week.

Var. cost	Site			Prod.
	1	2	3	
Factory 1	25	20	15	1,000
Factory 2	15	25	20	1,000
Factory 3	20	15	25	500
Factory 4	25	15	15	500
Fixed cost	500,000	500,000	500,000	
Capacity	1,500	1,500	1,500	

The decision maker who faces this problem must answer two very different types of questions: questions that require numerical answers (e.g. how many tonnes of chemical should factory i send to the site- j refining facility each week?) and questions that require yes–no answers (e.g. should the site- j facility be constructed?). While we can easily model the first type of question by using continuous decision variables (by letting x_{ij} equal the number of tonnes of chemical sent from factory

i to site j each week), we *cannot* do this with the second. We need to use integer variables. If we let y_j equal 1 if the site- j refining facility is constructed and 0 if it isn't, we quickly arrive at an integer programming formulation of the problem:

$$\begin{aligned} \text{minimize} \quad & 52 \cdot 25x_{11} + 52 \cdot 20x_{12} + 52 \cdot 15x_{13} \\ & + 52 \cdot 15x_{21} + 52 \cdot 25x_{22} + 52 \cdot 20x_{23} \\ & + 52 \cdot 20x_{31} + 52 \cdot 15x_{32} + 52 \cdot 25x_{33} \\ & + 52 \cdot 25x_{41} + 52 \cdot 15x_{42} + 52 \cdot 15x_{43} \\ & + 500,000y_1 + 500,000y_2 + 500,000y_3 \\ \text{subject to} \quad & x_{11} + x_{12} + x_{13} = 1,000 \\ & x_{21} + x_{22} + x_{23} = 1,000 \\ & x_{31} + x_{32} + x_{33} = 500 \\ & x_{41} + x_{42} + x_{43} = 500 \\ & x_{11} + x_{21} + x_{31} + x_{41} \leq 1,500y_1 \\ & x_{12} + x_{22} + x_{32} + x_{42} \leq 1,500y_2 \\ & x_{13} + x_{23} + x_{33} + x_{43} \leq 1,500y_3 \\ & x_{ij} \geq 0 \quad \text{for all } i \text{ and } j \\ & y_j \in \{0, 1\} \quad \text{for all } j. \end{aligned}$$

The objective is to minimize the yearly cost, the sum of the variable costs (which are measured in dollars per week) and the fixed costs (which are measured in dollars per year). The first set of constraints ensures that each factory's weekly chemical production is sent somewhere for refining. Since factory 1 produces 1,000 tonnes of chemical per week, factory 1 must ship a total of 1,000 tonnes of chemical to the various refining facilities each week. The second set of constraints guarantees two things: (1) if a facility is open, it will operate at or below its capacity, and (2) if a facility is not open, it will not operate at all. If the site-1 facility is open ($y_1 = 1$) then the factories can send it up to $1,500y_1 = 1,500 \times 1 = 1,500$ tonnes of chemical per week. If it is not open ($y_1 = 0$), then the factories can send it up to $1,500y_1 = 1,500 \times 0 = 0$ tonnes per week.

This introductory example demonstrates the need for integer variables. It also shows that with integer variables one can model simple logical requirements (if a facility is open, it can refine up to a certain amount of chemical; if not, it can't do any refining at all). It turns out that with integer variables one can model a whole host of logical requirements. One can also model fixed costs, sequencing and scheduling requirements, and many other problem aspects.

3.1.2 Solving the Facility Location IP

Given an integer program (IP), there is an associated linear program (LR) called the *linear relaxation*. It is formed by dropping (relaxing) the integrality restrictions. Since (LR) is less constrained than (IP), the following are immediate:

- If (IP) is a minimization problem, the optimal objective value of (LR) is less than or equal to the optimal objective value of (IP).

- If (IP) is a maximization problem, the optimal objective value of (LR) is greater than or equal to the optimal objective value of (IP).
- If (LR) is infeasible, then so is (IP).
- If all the variables in an optimal solution of (LR) are integer valued, then that solution is optimal for (IP) too.
- If the objective function coefficients are integer valued, then for minimization problems the optimal objective value of (IP) is greater than or equal to the ceiling of the optimal objective value of (LR). For maximization problems, the optimal objective value of (IP) is less than or equal to the floor of the optimal objective value of (LR).

In summary, solving (LR) can be quite useful: it provides a bound on the optimal value of (IP), and may (if we are lucky) give an optimal solution to (IP).

For the remainder of this section, we will let (IP) stand for the facility location integer program and (LR) for its linear programming relaxation. When we solve (LR), we get

Objective					
x_{11}	x_{12}	x_{13}			3,340,000
x_{21}	x_{22}	x_{23}	.	1,000	.
x_{31}	x_{32}	x_{33}	1,000	.	.
x_{41}	x_{42}	x_{43}	.	500	.
y_1	y_2	y_3	.	500	.
				$\frac{2}{3}$	$\frac{2}{3}$

This solution has factory 1 send all 1,000 tonnes of its chemical to site 3, factory 2 send all 1,000 tonnes of its chemical to site 1, factory 3 send all 500 tonnes to site 2, and factory 4 send all 500 tonnes to site 2. It constructs two-thirds of a refining facility at each site. Although it costs only 3,340,000 dollars per year, it cannot be implemented; all three of its integer variables take on fractional values.

It is tempting to try to produce a feasible solution by rounding. Here, if we round y_1 , y_2 and y_3 from $2/3$ to 1, we get lucky (this is certainly not always the case!) and get an integer feasible solution. Although we can state that this is a good solution—its objective value of 3,840,000 is within 15% of the objective value of (LR) and hence within 15% of optimal—we can't be sure that it is optimal.

So how can we find an optimal solution to (IP)? Examining the optimal solution to (LR), we see that y_1 , y_2 and y_3 are fractional. We want to force y_1 , y_2 and y_3 to be integer valued. We start by *branching* on y_1 , creating two new integer programming problems. In one, we add the constraint $y_1 = 0$. In the other, we will add the constraint $y_1 = 1$. Note that any optimal solution to (IP) must be feasible for one of the two subproblems.

After we solve the linear programming relaxations of the two subproblems, we can display what we know in a tree, as shown in Fig. 3.1.

Note that the optimal solution to the left subproblem's LP relaxation is integer valued. It is therefore an optimal solution to the left subproblem. Since there is no point in doing anything more with the left subproblem, we mark it with an “ \times ” and focus our attention on the right subproblem.

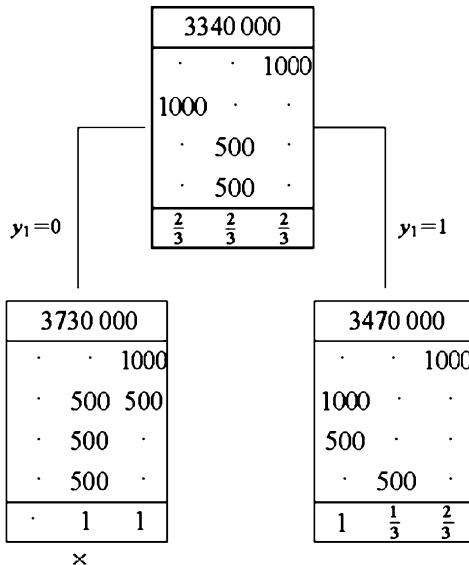


Fig. 3.1 Intermediate branch-and-bound tree

Both y_2 and y_3 are fractional in the optimal solution to the right subproblem's LP relaxation. We want to force both variables to be integer valued. Although we could branch on either variable, we will branch on y_2 . That is, we will create two more subproblems, one with $y_2 = 0$ and the other with $y_2 = 1$. After we solve the LP relaxations, we can update our tree, as in Fig. 3.2.

Note that we can immediately “ \times out” the left subproblem; the optimal solution to its LP relaxation is integer valued. In addition, by employing a *bounding* argument, we can also “ \times out” the right subproblem. The argument goes like this: Since the objective value of its LP relaxation ($3,636,666\frac{2}{3}$) is greater than the objective value of our newly found integer feasible solution (3,470,000), the optimal value of the right subproblem must be higher than (worse than) the objective value of our newly found integer feasible solution. So there is no point in expending any more effort on the right subproblem.

Since there are no active subproblems (subproblems that require branching), we are done. We have found an optimal solution to (IP). The optimal solution has factories 2 and 3 use the site-1 refining facility and factories 1 and 4 use the site-3 facility. The site-1 and site-3 facilities are constructed. The site-2 facility is not. The optimal solution costs 3,470,000 dollars per year, 370,000 dollars per year less than the solution obtained by rounding the solution to (LR).

This method is called *branch and bound*, and is the most common method for finding solutions to integer programming formulations.

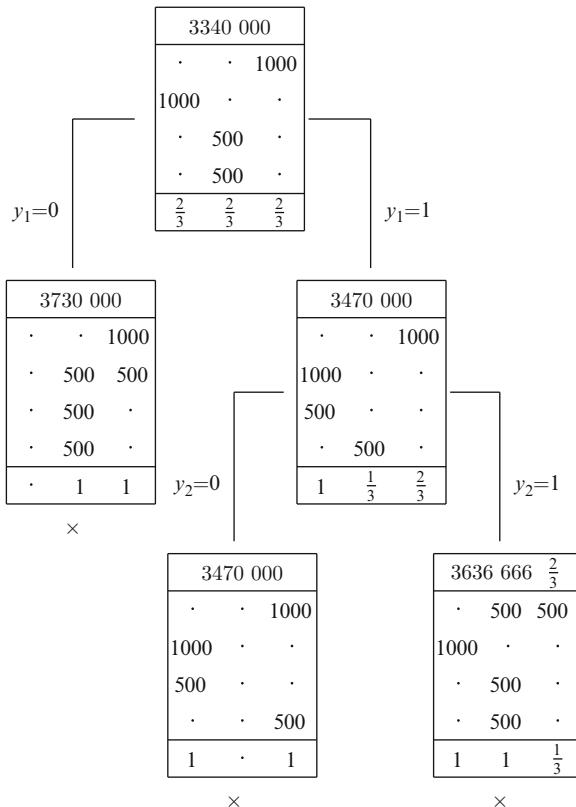


Fig. 3.2 Final branch-and-bound tree

3.1.3 Difficulties with Integer Programs

While we were able to get the optimal solution to the example integer program relatively quickly, it is not always the case that branch and bound quickly solves integer programs. In particular, it is possible that the *bounding* aspects of branch and bound are not invoked, and the branch and bound algorithm can then generate a huge number of subproblems. In the worst case, a problem with n binary variables (variables that have to take on the value 0 or 1) can have 2^n subproblems. This exponential growth is inherent in any algorithm for integer programming (unless P = NP) due to the range of problems that can be formulated within integer programming.

Despite the possibility of extreme computation time, there are a number of techniques that have been developed to increase the likelihood of finding optimal solutions quickly. After we discuss creativity in formulations, we will discuss some of these techniques.

3.2 Be Creative in Formulations

At first, it may seem that integer programming does not offer much over linear programming: both require linear objectives and constraints, and both have numerical variables. Can requiring some of the variables to take on integer values significantly expand the capability of the models? Absolutely . . . integer programming models go far beyond the power of linear programming models. The key is the creative use of integrality to model a wide range of common structures in models. Here we outline some of the major uses of integer variables.

3.2.1 Integer Quantities

The most obvious use of integer variables is when an integer quantity of a good is required. For instance, in a production model involving television sets, an integral number of television sets might be required. Or, in a personnel assignment problem, an integer number of workers might be assigned to a shift.

This use of integer variables is the most obvious, and the most over-used. For many applications, the added “accuracy” in requiring integer variables is far outweighed by the greater difficulty in finding the optimal solution. For instance, in the production example, if the number of televisions produced is in the hundreds (say the fractional optimal solution is 202.7) then having a plan with the rounded-off value (203 in this example) is likely appropriate in practice. The uncertainty of the data almost certainly means that no production plan is accurate to four figures! Similarly, if the personnel assignment problem is for a large enterprise over a year, and the linear programming model suggests 154.5 people are required, it is probably not worthwhile to invoke an integer programming model in order to handle the fractional parts.

However, there are times when integer quantities are required. A production system that can produce either two or three aircraft carriers and a personnel assignment problem for small teams of five or six people are examples. In these cases, the addition of the integrality constraint can mean the difference between useful models and irrelevant models.

3.2.2 Binary Decisions

Perhaps the most used type of integer variable is the *binary variable*: an integer variable restricted to take on the values 0 or 1. We will see a number of uses of these variables. Our first example is in modeling binary decisions.

Many practical decisions can be seen as “yes” or “no” decisions: should we construct a chemical refining facility in site j (as in the introduction), should we invest in project B, should we start producing new product Y? For many of these decisions,

a binary integer programming model is appropriate. In such a model, each decision is modeled with a binary variable: setting the variable equal to 1 corresponds to making the “yes” decision, while setting it to 0 corresponds to going with the “no” decision. Constraints are then formed to correspond to the effects of the decision.

As an example, suppose we need to choose among projects A, B, C and D. Each project has a capital requirement (\$1, \$2.5, \$4 and \$5 million respectively) and an expected return (say, \$3, \$6, \$13 and \$16 million). If we have \$7 million to invest, which projects should we take on in order to maximize our expected return?

We can formulate this problem with binary variables x_A, x_B, x_C and x_D representing the decision to take on the corresponding project. The effect of taking on a project is to use up some of the funds we have available to invest. Therefore, we have a constraint

$$x_A + 2.5x_B + 4x_C + 5x_D \leq 7.$$

Our objective is to maximize the expected profit:

$$\text{Maximize } 3x_A + 6x_B + 13x_C + 16x_D.$$

In this case, binary variables let us make the “yes–no” decision on whether to invest in each fund, with a constraint ensuring that our overall decisions are consistent with our budget. Without integer variables, the solution to our model would have fractional parts of projects, which may not be in keeping with the needs of the model.

3.2.3 Fixed-Charge Requirements

In many production applications, the cost of producing x of an item is roughly linear except for the special case of producing no items. In that case, there are additional savings since no equipment or other items need be procured for the production. This leads to a *fixed-charge* structure. The cost for producing x of an item is

- 0, if $x = 0$
- $c_1 + c_2x$, if $x > 0$ for constants c_1, c_2 .

This type of cost structure is impossible to embed in a linear program. With integer programming, however, we can introduce a new binary variable y . The value $y = 1$ is interpreted as having non-zero production, while $y = 0$ means no production. The objective function for these variables then becomes

$$c_1y + c_2x,$$

which is appropriately linear in the variables. It is necessary, however, to add constraints that link the x and y variables. Otherwise, the solution might be $y = 0$ and $x = 10$, which we do not want. If there is an upper bound M on how large x can be (perhaps derived from other constraints), then the constraint

$$x \leq My$$

correctly links the two variables. If $y = 0$ then x must equal 0; if $y = 1$ then x can take on any value. Technically, it is possible to have the values $x = 0$ and $y = 1$ with this formulation, but as long as this is modeling a fixed cost (rather than a fixed profit), this will not be an optimal (cost minimizing) solution.

This use of “M” values is common in integer programming, and the result is called a Big-M model. Big-M models are often difficult to solve, for reasons we will see.

We saw this fixed-charge modeling approach in our initial facility location example. There, the y variables corresponded to opening a refining facility (incurring a fixed cost). The x variables correspond to assigning a factory to the refining facility, and there was an upper bound on the volume of raw material a refinery could handle.

3.2.4 Logical Constraints

Binary variables can also be used to model complicated logical constraints, a capability not available in linear programming. In a facility location problem with binary variables y_1, y_2, y_3, y_4 and y_5 corresponding to the decisions to open warehouses at locations 1, 2, 3, 4 and 5 respectively, complicated relationships between the warehouses can be modeled with linear functions of the y variables. Here are a few examples:

- At most one of locations 1 and 2 can be opened: $y_1 + y_2 \leq 1$
- Location 3 can only be opened if location 1 is: $y_3 \leq y_1$
- Location 4 cannot be opened if locations 2 or 3 are: $y_4 + y_2 \leq 1, y_4 + y_3 \leq 1$
- If location 1 is open, either locations 2 or 5 must be: $y_2 + y_5 \geq y_1$.

Much more complicated logical constraints can be formulated with the addition of new binary variables. Consider a constraint of the form $3x_1 + 4x_2 \leq 10$ OR $4x_1 + 2x_2 \geq 12$. As written, this is not a linear constraint. However, if we let M be the largest either $|3x_1 + 4x_2|$ or $|4x_1 + 2x_2|$ can be, then we can define a new binary variable z which is 1 only if the first constraint is satisfied and 0 only if the second constraint is satisfied. Then we get the constraints

$$\begin{aligned} 3x_1 + 4x_2 &\leq 10 + (M - 10)(1 - z) \\ 4x_1 + 2x_2 &\geq 12 - (M + 12)z. \end{aligned}$$

When $z = 1$, we get

$$\begin{aligned} 3x_1 + 4x_2 &\leq 10 \\ 4x_1 + 2x_2 &\geq -M. \end{aligned}$$

When $z = 0$, we get

$$\begin{aligned} 3x_1 + 4x_2 &\leq M \\ 4x_1 + 2x_2 &\geq 12. \end{aligned}$$

This correctly models the original nonlinear constraint.

As we can see, logical requirements often lead to Big-M type formulations.

3.2.5 Sequencing Problems

Many problems in sequencing and scheduling require the modeling of the order in which items appear in the sequence. For instance, suppose we have a model in which there are items, where each item i has a processing time on a machine p_i . If the machine can only handle one item at a time and we let t_i be a (continuous) variable representing the start time of item i on the machine, then we can ensure that items i and j are not on the machine at the same time with the constraints

$$\begin{aligned} t_j &\geq t_i + p_i \text{ IF } t_j \geq t_i \\ t_i &\geq t_j + p_j \text{ IF } t_j < t_i. \end{aligned}$$

This can be handled with a new binary variable y_{ij} which is 1 if $t_i \leq t_j$ and 0 otherwise. This gives the constraints

$$\begin{aligned} t_j &\geq t_i + p_i - M(1-y) \\ t_i &\geq t_j + p_j - My \end{aligned}$$

for sufficiently large M . If y is 1 then the second constraint is automatically satisfied (so only the first is relevant) while the reverse happens for $y = 0$.

3.3 Find Formulations with Strong Relaxations

As the previous section made clear, integer programming formulations can be used for many problems of practical interest. In fact, for many problems, there are many alternative integer programming formulations. Finding a *good* formulation is key to the successful use of integer programming. The definition of a good formulation is primarily computational: a good formulation is one for which branch and bound (or another integer programming algorithm) will find and prove the optimal solution quickly. Despite this empirical aspect of the definition, there are some guidelines to help in the search for good formulations. The key to success is to find formulations whose linear relaxation is not too different from the underlying integer program.

We saw in our first example that solving linear relaxations was key to the basic integer programming algorithm. If the solution to the initial linear relaxation is integer, then no branching need be done and integer programming is no harder

than linear programming. Unfortunately, finding formulations with this property is very hard to do. But some formulations can be better than other formulations in this regard.

Let us modify our facility location problem by requiring that every factory be assigned to exactly one refinery (incidentally, the optimal solution to our original formulation happened to meet this requirement). Now, instead of having x_{ij} be the tonnes sent from factory i to refinery j , we define x_{ij} to be 1 if factory i is serviced by refinery j . Our formulation becomes

$$\begin{aligned} \text{minimize} \quad & 1,000 \cdot 52 \cdot 25x_{11} + 1,000 \cdot 52 \cdot 20x_{12} + 1,000 \cdot 52 \cdot 15x_{13} \\ & + 1,000 \cdot 52 \cdot 15x_{21} + 1,000 \cdot 52 \cdot 25x_{22} + 1,000 \cdot 52 \cdot 20x_{23} \\ & + 500 \cdot 52 \cdot 20x_{31} + 500 \cdot 52 \cdot 15x_{32} + 500 \cdot 52 \cdot 25x_{33} \\ & + 500 \cdot 52 \cdot 25x_{41} + 500 \cdot 52 \cdot 15x_{42} + 500 \cdot 52 \cdot 15x_{43} \\ & + 500,000y_1 + 500,000y_2 + 500,000y_3 \\ \text{subject to} \quad & x_{11} + x_{12} + x_{13} = 1 \\ & x_{21} + x_{22} + x_{23} = 1 \\ & x_{31} + x_{32} + x_{33} = 1 \\ & x_{41} + x_{42} + x_{43} = 1 \\ & 1,000x_{11} + 1,000x_{21} + 500x_{31} + 500x_{41} \leq 1,500y_1 \\ & 1,000x_{12} + 1,000x_{22} + 500x_{32} + 500x_{42} \leq 1,500y_2 \\ & 1,000x_{13} + 1,000x_{23} + 500x_{33} + 500x_{43} \leq 1,500y_3 \\ & x_{ij} \in \{0, 1\} \quad \text{for all } i \text{ and } j \\ & y_j \in \{0, 1\} \quad \text{for all } j. \end{aligned}$$

Let us call this formulation the *base formulation*. This is a correct formulation to our problem. There are alternative formulations, however. Suppose we add to the base formulation the set of constraints

$$x_{ij} \leq y_j \quad \text{for all } i \text{ and } j.$$

Call the resulting formulation the *expanded formulation*. Note that it too is an appropriate formulation for our problem. At the simplest level, it appears that we have simply made the formulation larger: there are more constraints so the linear programs solved within branch-and-bound will likely take longer to solve. Is there any advantage to the expanded formulation?

The key is to look at non-integer solutions to linear relaxations of two formulations: we know the two formulations have the same integer solutions (since they are formulations of the same problem), but they can differ in non-integer solutions. Consider the solution $x_{13} = 1, x_{21} = 1, x_{32} = 1, x_{42} = 1, y_1 = 2/3, y_2 = 2/3, y_3 = 2/3$. This solution is feasible to the linear relaxation of the base formulation but is not feasible to the linear relaxation of the expanded formulation. If the branch-and-bound algorithm works on the base formulation, it may have to consider this solution; with the expanded formulation, this solution can never be examined. If there are fewer fractional solutions to explore (technically, fractional extreme point solutions), branch and bound will typically terminate quicker.

Since we have added constraints to get the expanded formulation, there is no non-integer solution to the linear relaxation of the expanded formulation that is not also feasible for the linear relaxation of the base formulation. We say that the expanded formulation is *tighter* than the base formulation.

In general, tighter formulations are to be preferred for integer programming formulations even if the resulting formulations are larger. Of course, there are exceptions: if the size of the formulation is much larger, the gain from the tighter formulation may not be sufficient to offset the increased linear programming times. Such cases are definitely the exception, however: almost invariably, tighter formulations are better formulations. For this particular instance, the expanded formulation happens to provide an integer solution without branching.

There has been a tremendous amount of work done on finding tighter formulations for different integer programming models. For many types of problems, classes of constraints (or *cuts*) to be added are known. These constraints can be added in one of two ways: they can be included in the original formulation or they can be added as needed to remove fractional values. The latter case leads to a *branch and cut* approach, which is the subject of Sect. 3.6.

A cut relative to a formulation has to satisfy two properties: first, every feasible integer solution must also satisfy the cut; second, some fractional solution that is feasible to the linear relaxation of the formulation must not satisfy the cut. For instance, consider the single constraint

$$3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16,$$

where the x_i are binary variables. Then the constraint $x_3 + x_4 \leq 1$ is a cut (every integer solution satisfies it and, for instance $x = (0, 0, 0.75, 1)$ does not) but $x_1 + x_2 + x_3 + x_4 \leq 4$ is not a cut (no fractional solutions removed) nor is $x_1 + x_2 + x_3 \leq 2$ (which incorrectly removes $x = (1, 1, 1, 0)$).

Given a formulation, finding cuts to add to it to strengthen the formulation is not a routine task. It can take deep understanding, and a bit of luck, to find improving constraints.

One generally useful approach is called the Chvátal (or Gomory–Chvátal) procedure. Here is how the procedure works for “ \leq ” constraints where all the variables are non-negative integers:

1. Take one or more constraints, multiple each by a non-negative constant (the constant can be different for different constraints). Add the resulting constraints into a single “ \leq ” constraint.
2. Round down each coefficient on the left-hand side of the constraint.
3. Round down the right-hand side of the constraint.

The result is a constraint that does not cut off any feasible integer solutions. It may be a cut if the effect of rounding down the right-hand side of the constraint is more than the effect of rounding down the coefficients.

This is best seen through an example. Taking the constraint above, let us take the two constraints

$$\begin{aligned} 3x_1 + 5x_2 + 8x_3 + 10x_4 &\leq 16 \\ x_3 &\leq 1. \end{aligned}$$

If we multiply each constraint by 1/9 and add them we get

$$3/9x_1 + 5/9x_2 + 9/9x_3 + 10/9x_4 \leq 17/9.$$

Now, round down the left-hand coefficients (this is valid since the x variables are non-negative and it is a “ \leq ” constraint):

$$x_3 + x_4 \leq 17/9.$$

Finally, round down the right-hand side (this is valid since the x variables are integer) to get

$$x_3 + x_4 \leq 1$$

which turns out to be a cut. Notice that the three steps have differing effects on feasibility. The first step, since it is just taking a linear combination of constraints neither adds nor removes feasible values; the second step weakens the constraint, and may add additional fractional values; the third step strengthens the constraint, ideally removing fractional values.

This approach is particularly useful when the constants are chosen so that no rounding-down is done in the second step. For instance, consider the following set of constraints (where the x_i are binary variables):

$$\begin{aligned} x_1 + x_2 &\leq 1 \\ x_2 + x_3 &\leq 1 \\ x_1 + x_3 &\leq 1. \end{aligned}$$

These types of constraints often appear in formulations where there are lists of mutually exclusive variables. Here, we can multiply each constraint by 1/2 and add them to get

$$x_1 + x_2 + x_3 \leq 3/2.$$

There is no rounding down on the left-hand side, so we can move on to rounding down the right-hand side to get

$$x_1 + x_2 + x_3 \leq 1,$$

which, for instance, cuts off the solution $x = (1/2, 1/2, 1/2)$.

In cases where no rounding-down is needed on the left-hand side but there is rounding-down on the right-hand side, the result has to be cut (relative to the included constraints). Conversely, if no rounding down is done on the right-hand side, the result cannot be a cut.

In the formulation section, we mentioned that Big-M formulations often lead to poor formulations. This is because the linear relaxation of such a formulation often allows for many fractional values. For instance, consider the constraint (all variables are binary)

$$x_1 + x_2 + x_3 \leq 1,000y.$$

Such constraints often occur in facility location and related problems. This constraint correctly models a requirement that the x variables can be 1 only if y is also 1, but does so in a very weak way. Even if the x values of the linear relaxation are integer, y can take on a very small value (instead of the required 1). Here, even for $x = (1, 1, 1)$, y need only be 3/1,000 to make the constraint feasible. This typically leads to very bad branch-and-bound trees: the linear relaxation gives little guidance as to the *true* values of the variables.

Better would be the constraint

$$x_1 + x_2 + x_3 \leq 3y,$$

which forces y to take on larger values. This is the concept of making the M in Big-M as small as possible. Better still would be the three constraints

$$\begin{aligned} x_1 &\leq y \\ x_2 &\leq y \\ x_3 &\leq y, \end{aligned}$$

which forces y to be integer as soon as the x values are.

Finding improved formulations is a key concept to the successful use of integer programming. Such formulations typically revolve around the strength of the linear relaxation: does the relaxation well-represent the underlying integer program? Finding classes of cuts can improve formulations. Finding such classes can be difficult, but without good formulations, integer programming models are unlikely to be successful except for very small instances.

3.4 Avoid Symmetry

Symmetry often causes integer programming models to fail. Branch and bound can become an extremely inefficient algorithm when the model being solved displays many symmetries.

Consider again our facility location model. Suppose instead of having just one refinery at a site, we were permitted to have up to three refineries at a site. We could modify our model by having variables y_j , z_j and w_j for each site (representing the three refineries). In this formulation, the cost and other coefficients for y_j are the same as for z_j and w_j . The formulation is straightforward, but branch and bound does very poorly on the result.

The reason for this is symmetry: for every solution in the branch-and-bound tree with a given y , z and w , there is an equivalent solution with z taking on y 's values, w taking on z 's and y taking on w . This greatly increases the number of solutions the branch-and-bound algorithm must consider in order to find and prove the optimality of a solution.

It is important to remove as many symmetries in a formulation as possible. Depending on the problem and the symmetry, this removal can be done by adding constraints, fixing variables or modifying the formulation.

For our facility location problem, the easiest thing to do is to add the constraints

$$y_j \geq z_j \geq w_j \text{ for all } j.$$

Now, at a refinery site, z_j can be non-zero only if y_j is non-zero, and w_j non-zero only if both y_j and z_j are. This partially breaks the symmetry of this formulation, though other symmetries (particularly in the x variables) remain.

This formulation can be modified in another way by redefining the variables. Instead of using binary variables, let y_j be the number of refineries put in location j . This removes all of the symmetries at the cost of a weaker linear relaxation (since some of the strengthenings we have explored require binary variables).

Finally, to illustrate the use of variable fixing, consider the problem of coloring a graph with K colors: we are given a graph with node set V and edge set E and wish to determine if we can assign a value $v(i)$ to each node i such that $v(i) \in \{1, \dots, K\}$ and $v(i) \neq v(j)$ for all $(i, j) \in E$.

We can formulate this problem as an integer programming by defining a binary variable x_{ik} to be 1 if i is given color k and 0 otherwise. This leads to the constraints

$$\begin{aligned} \sum_k x_{ik} &= 1 && \text{for all } i \text{ (every node gets a color)} \\ x_{ik} + x_{jk} &= 1 && \text{for all } k, (i, j) \in E \text{ (no adjacent get same)} \\ x_{ik} &\in \{0, 1\} && \text{for all } i, k. \end{aligned}$$

The graph-coloring problem is equivalent to determining if the above set of constraints is feasible. This can be done by using branch-and-bound with an arbitrary objective value.

Unfortunately, this formulation is highly symmetric. For any coloring of graph, there is an equivalent coloring that arises by permuting the coloring (that is, permuting the set $\{1, \dots, k\}$ in this formulation). This makes branch and bound very ineffective for this formulation. Note also that the formulation is very weak, since setting $x_{ik} = 1/k$ for all i, k is a feasible solution to the linear relaxation no matter what E is.

We can strengthen this formulation by breaking the symmetry through variable fixing. Consider a clique (set of mutually adjacent vertices) of the graph. Each member of the clique has to get a different color. We can break the symmetry by finding a large (ideally maximum sized) clique in the graph and setting the colors of the clique arbitrarily, but fixed. So if the clique has size k_c , we would assign the colors $1, \dots, k_c$ to members of the clique (adding in constraints forcing the corresponding x values to be 1). This greatly reduces the symmetry, since now only permutations among the colors $k_c + 1, \dots, K$ are valid. This also removes the $x_{ik} = 1/k$ solution from consideration.

3.5 Consider Formulations with Many Constraints

Given the importance of the strength of the linear relaxation, the search for improved formulations often leads to sets of constraints that are too large to include in the formulation. For example, consider a single constraint with non-negative coefficients

$$a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_nx_n \leq b,$$

where the x_i are binary variables. Consider a subset S of the variables such that $\sum_{i \in S} a_i > b$. The constraint

$$\sum_{i \in S} x_i \leq |S| - 1$$

is valid (it isn't violated by any feasible integer solution) and cuts off fractional solutions as long as S is minimal. These constraints are called *cover constraints*. We would then like to include this set of constraints in our formulation.

Unfortunately, the number of such constraints can be very large. In general, it is exponential in n , making it impractical to include the constraints in the formulation. But the relaxation is much tighter with the constraints.

To handle this problem, we can choose to generate only those constraints that are needed. In our search for an optimal integer solution, many of the constraints aren't needed. If we can generate the constraints as we need them, we can get the strength of the improved relaxation without the huge number of constraints.

Suppose our instance is

$$\text{Maximize } 9x_1 + 14x_2 + 20x_3 + 32x_4$$

Subject to

$$3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16$$

$$x_i \in \{0, 1\}.$$

The optimal solution to the linear relaxation is $x^* = (1, 0.6, 0, 1)$ with objective 49.4. Now consider the set $S = (x_1, x_2, x_4)$. The constraint

$$x_1 + x_2 + x_4 \leq 2$$

is a cut that x^* violates. If we add that constraint to our problem, we get a tighter formulation. Solving this model gives solution $x = (1, 0, 0.375, 1)$ and objective 48.5. The constraint

$$x_3 + x_4 \leq 1$$

is a valid cover constraint that cuts off this solution. Adding this constraint and solving gives solution $x = (0, 1, 0, 1)$ with objective 46. This is the optimal solution to the original integer program, which we have found only by generating cover inequalities.

In this case, the cover inequalities were easy to see, but this process can be formalized. A reasonable heuristic for identifying violated cover inequalities would be

to order the variables by decreasing $a_i x_i^*$ then add the variables to the cover S until $\sum_{i \in S} a_i > b$. This heuristic is not guaranteed to find violated cover inequalities (for that, a knapsack optimization problem can be formulated and solved) but even this simple heuristic can create much stronger formulations without adding too many constraints.

This idea is formalized in the *branch-and-cut* approach to integer programming. In this approach, a formulation has two parts: the *explicit constraints* (denoted $Ax \leq b$) and the *implicit constraints* ($A'x \leq b'$). Denote the objective function as Maximize cx . Here we will assume that all x are integral variables, but this can be easily generalized:

- Step 1 Solve the linear program “Maximize cx ” subject to $Ax \leq b$ to get optimal relaxation solution x^* .
- Step 2 If x^* integer, then stop. x^* is optimal.
- Step 3 Try to find a constraint $a'x \leq b'$ from the implicit constraints such that $a'x^* > b$. If found, add $a'x \leq b'$ to the $Ax \leq b$ constraint set and go to step 1. Otherwise, do branch-and-bound on the current formulation.

In order to create a branch-and-cut model, there are two aspects: the definition of the implicit constraints, and the definition of the approach in Step 3 to find violated inequalities. The problem in Step 3 is referred to as the *separation problem* and is at the heart of the approach. For many sets of constraints, no good separation algorithm is known. Note, however, that the separation problem might be solved heuristically: it may miss opportunities for separation and therefore invoke branch-and-bound too often. Even in this case, it is often the case that the improved formulations are sufficiently tight to greatly decrease the time needed for branch-and-bound.

This basic algorithm can be improved by doing cut generation within the branch-and-bound tree. It may be that by fixing variables, different constraints become violated and those can be added to the subproblems.

3.6 Consider Formulations with Many Variables

Just as improved formulations can result from adding many constraints, adding many variables can lead to very good formulations. Let us begin with our graph coloring example. Recall that we are given a graph with vertices V and edges E and want to assign a value $v(i)$ to each node i such that $v(i) \neq v(j)$ for all $(i, j) \in E$. Our objective is to use the minimum number of different values (before, we had a fixed number of colors to use: in this section we will use the optimization version rather than the feasibility version of this problem).

Previously, we described a model using binary variables x_{ik} denoting whether node i gets color k or not. As an alternative model, let us concentrate on the set of nodes that gets the same color. Such a set must be an *independent set* (a set of mutually non-adjacent nodes) of the graph. Suppose we listed all independent sets of the graph: S_1, S_2, \dots, S_m . Then we can define binary variables y_1, y_2, \dots, y_m with the

interpretation that $y_j = 1$ means that independent set S_j is part of the coloring, and $y_j = 0$ means that independent set S_j is not part of the coloring. Now our formulation becomes

$$\begin{aligned} & \text{Minimize } \sum_j y_j \\ & \text{Subject to} \\ & \quad \sum_{j:i \in S_j} y_j = 1 \text{ for all } i \in V \\ & \quad y_j \in \{0, 1\} \text{ for all } j \in \{1 \dots m\}. \end{aligned}$$

The constraint states that every node must be in some independent set of the coloring.

This formulation is a much better formulation than our x_{ik} formulation. It does not have the symmetry problems of the previous formulation and results in a much tighter linear relaxation. Unfortunately, the formulation is impractical for most graphs because the number of independent sets is exponential in the number of nodes, leading to an impossibly large formulation.

Just as we could handle an exponential number of constraints by generating them as needed, we can also handle an exponential number of variables by *variable generation*: the creation of variables only as they are needed. In order to understand how to do this, we will have to understand some key concepts from linear programming.

Consider a linear program, where the variables are indexed by j and the constraints indexed by i :

$$\begin{aligned} & \text{Maximize } \sum_j c_j x_j \\ & \text{Subject to} \\ & \quad \sum_i a_{ij} x_{ij} \leq b_i \text{ for all } i \\ & \quad x_j \geq 0 \text{ for all } j. \end{aligned}$$

When this linear program is solved, the result is the optimal solution x^* . In addition, however, there is a value called the *dual value*, denoted π_i , associated with each constraint. This value gives the marginal change in the objective value as the right-hand side for the corresponding constraint is changed. So if the right-hand side of constraint i changes to $b_i + \Delta$, then the objective will change by $\pi_i \Delta$ (there are some technical details ignored here involving how large Δ can be for this to be a valid calculation: since we are only concerned with marginal calculations, we can ignore these details).

Now, suppose there is a new variable x_{n+1} , not included in the original formulation. Suppose it could be added to the formulation with corresponding objective coefficient c_{n+1} and coefficients $a_{i,n+1}$. Would adding the variable to the formulation result in an improved formulation? The answer is certainly “no” in the case when

$$c_{n+1} < \sum_i a_{i,n+1} \pi_i.$$

In this case, the value gained from the objective is insufficient to offset the cost charged marginally by the effect on the constraints. We need $c_{n+1} - \sum_i a_{i,n+1}\pi_i > 0$ in order to possibly improve on our solution.

This leads to the idea of variable generation. Suppose you have a formulation with a huge number of variables. Rather than solve this huge formulation, begin with a smaller number of variables. Solve the linear relaxation and get dual values π . Using π , determine if there is one (or more) variables whose inclusion might improve the solution. If not, then the linear relaxation is solved. Otherwise, add one or more such variables to the formulation and repeat.

Once the linear relaxation is solved, if the solution is integer, then it is optimal. Otherwise, branch and bound is invoked, with the variable generation continuing in the subproblems.

Key to this approach is the algorithm for generating the variables. For a huge number of variables it is not enough to check all of them: that would be too time consuming. Instead, some sort of optimization problem must be defined whose solution is an improving variable. We'll illustrate this for our graph coloring problem.

Suppose we begin with a limited set of independent sets and solve our relaxation over them. This leads to a dual value π_i for each node. For any other independent set S , if $\sum_{i \in S} \pi_i > 1$, then S corresponds to an improving variable. We can write this problem using binary variables z_i corresponding to whether i is in S or not:

$$\begin{aligned} & \text{Maximize } \sum_i \pi_i z_i \\ & \text{Subject to} \\ & \quad z_i + z_j \leq 1 \text{ for all } (i, j) \in E \\ & \quad z_i \in \{0, 1\} \text{ for all } i. \end{aligned}$$

This problem is called the *maximum weighted independent set* (MWIS) problem, and, while the problem is formally hard, effective methods have been found for solving it for problems of reasonable size.

This gives a variable generation approach to graph coloring: begin with a small number of independent sets, then solve the MWIS problem, adding in independent sets until no independent set improves the current solution. If the variables are integer, then we have the optimal coloring. Otherwise we need to branch.

Branching in this approach needs special care. We need to branch in such a way that our subproblem is not affected by our branching. Here, if we simply branch on the y_j variables (so have one branch with $y_j = 1$ and another with $y_j = 0$), we end up not being able to use the MWIS model as a subproblem. In the case where $y_j = 0$ we need to find an improving set, except S_j does not count as improving. This means we need to find the second most improving set. As more branching goes on, we may need to find the third most improving, the fourth most improving, and so on. To handle this, specialized branching routines are needed (involving identifying nodes that, on one side of the branch, must be the same color and, on the other side of the branch, cannot be the same color).

Variable generation together with appropriate branching rules and variable generation at the subproblems is a method known as *branch and price*. This approach has been very successful in attacking a variety of very difficult problems over the last few years.

To summarize, models with a huge number of variables can provide very tight formulations. To handle such models, it is necessary to have a variable generation routine to find improving variables, and it may be necessary to modify the branching method in order to keep the subproblems consistent with that routine. Unlike constraint generation approaches, heuristic variable generation routines are not enough to ensure optimality: at some point it is necessary to prove conclusively that the right variables are included. Furthermore, these variable generation routines must be applied at each node in the branch-and-bound tree if that node is to be crossed out from further analysis.

3.7 Modify Branch-and-Bound Parameters

Integer programs are solved with computer programs. There are a number of computer programs available to solve integer programs. These range from basic spreadsheet-oriented systems to open-source research codes to sophisticated commercial applications. To a greater or lesser extent, each of these codes offers parameters and choices that can have a significant affect on the solvability of integer programming models. For most of these parameters, the only way to determine the best choice for a particular model is experimentation: any choice that is uniformly dominated by another choice would not be included in the software.

Here are some common key choices and parameters, along with some comments on each.

3.7.1 *Description of Problem*

The first issue to be handled is to determine how to describe the integer program to the optimization routine(s). Integer programs can be described as spreadsheets, computer programs, matrix descriptors and higher-level languages. Each has advantages and disadvantages with regards to such issues as ease of use, solution power, flexibility and so on. For instance, implementing a branch-and-price approach is difficult if the underlying solver is a spreadsheet program. Using “callable libraries” that give access to the underlying optimization routines can be very powerful, but can be time consuming to develop.

Overall, the interface to the software will be defined by the software. It is generally useful to be able to access the software in multiple ways (callable libraries, high-level languages, command line interfaces) in order to have full flexibility in solving.

3.7.2 Linear Programming Solver

Integer programming relies heavily on the underlying linear programming solver. Thousands or tens of thousands of linear programs might be solved in the course of branch-and-bound. Clearly a faster linear programming code can result in faster integer programming solutions. Some possibilities that might be offered are primal simplex, dual simplex, or various interior point methods. The choice of solver depends on the problem size and structure (for instance, interior point methods are often best for very large, block-structured models) and can differ for the initial linear relaxation (when the solution must be found “from scratch”) and subproblem linear relaxations (when the algorithm can use previous solutions as a starting basis). The choice of algorithm can also be affected by whether constraint and/or variable generation are being used.

3.7.3 Choice of Branching Variable

In our description of branch-and-bound, we allowed branching on any fractional variable. When there are multiple fractional variables, the choice of variable can have a big effect on the computation time. As a general guideline, more “important” variables should be branched on first. In a facility location problem, the decisions on opening a facility are generally more important than the assignment of a customer to that facility, so those would be better choices for branching when a choice must be made.

3.7.4 Choice of Subproblem to Solve

Once multiple subproblems have been generated, it is necessary to choose which subproblem to solve next. Typical choices are depth-first search, breadth-first search, or best-bound search. Depth-first search continues fixing variables for a single problem until integrality or infeasibility results. This can lead quickly to an integer solution, but the solution might not be very good. Best-bound search works with subproblems whose linear relaxation is as large (for maximization) as possible, with the idea that subproblems with good linear relaxations may have good integer solutions.

3.7.5 Direction of Branching

When a subproblem and a branching variable have been chosen, there are multiple subproblems created corresponding to the values the variable can take on. The ordering of the values can affect how quickly good solutions can be found. Some choices

here are a fixed ordering or the use of estimates of the resulting linear relaxation value. With fixed ordering, it is generally good to first try the more restrictive of the choices (if there is a difference).

3.7.6 Tolerances

It is important to note that while integer programming problems are primarily combinatorial, the branch-and-bound approach uses numerical linear programming algorithms. These methods require a number of parameters giving allowable tolerances. For instance, if $x_j = 0.998$ should x_j be treated as the value 1 or should the algorithm branch on x_j ? While it is tempting to give overly big values (to allow for faster convergence) or small values (to be “more accurate”), either extreme can lead to problems. While for many problems the default values from a quality code are sufficient, these values can be the source of difficulties for some problems.

3.8 Tricks of the Trade

Faced with the contents of this chapter, all of which is about “tricks of the trade”, it is easy to throw one’s hands up and give up on integer programming! There are so many choices, so many pitfalls, and so much chance that the combinatorial explosion will make solving problems impossible. Despite this complexity, integer programming is used routinely to solve problems of practical interest. There are a few key steps to make your integer programming implementation go well.

- Use state-of-the-art software. It is tempting to use software because it is easy, or available, or cheap. For integer programming, however, not having the most current software embedding the latest techniques can doom your project to failure. Not all such software is commercial. The COIN-OR project is an open-source effort to create high-quality optimization codes.
- Use a modeling language. A modeling language, such as OPL, Mosel, AMPL, or other language can greatly reduce development time, and allows for easy experimentation of alternatives. Callable libraries can give more power to the user, but should be reserved for “final implementations”, once the model and solution approached are known.
- If an integer programming model does not solve in a reasonable amount of time, look at the formulation first, not the solution parameters. The default settings of current software are generally pretty good. The problem with most integer programming formulations is the formulation, not the choice of branching rule, for example.
- Solve some small instances and look at the solutions to the linear relaxations. Often constraints to add to improve a formulation are quite obvious from a few small examples.

- Decide whether you need “optimal” solutions. If you are consistently getting within 0.1 % of optimal, without proving optimality, perhaps you should declare success and go with the solutions you have, rather than trying to hunt down that final gap.
- Try radically different formulations. Often, there is another formulation with completely different variables, objective and constraints which will have a very different computational experience.

3.9 Conclusion

Integer programming models represent a powerful approach to solving hard problems. The bounds generated from linear relaxations are often sufficient to greatly cut down on the search tree for these problems. Key to successful integer programming is the creation of good formulations. A good formulation is one where the linear relaxation closely resembles the underlying integer program. Improved formulations can be developed in a number of ways, including finding formulations with tight relaxations, avoiding symmetry, and creating and solving formulations that have an exponential number of variables or constraints. It is through the judicious combination of these approaches, combined with fast integer programming computer codes, that the practical use of integer programming has greatly expanded in the last 20 years.

Sources of Additional Information

Integer programming has existed for more than 50 years and has developed a huge literature. This bibliography therefore makes no effort to be comprehensive, but rather provides initial pointers for further investigation.

General Integer Programming

There have been a number of excellent monographs on integer programming recently. The classic is [Nemhauser and Wolsey \(1998\)](#). A more recent book updating much of the material is [Wolsey \(1998\)](#). [Schrijver \(1998\)](#) is an outstanding reference book, covering the theoretical underpinnings of integer programming. An unusual resource on integer programming is a volume by [Jünger et al. \(2010\)](#) celebrating 50 years of integer programming. This volume contains a mix of classic papers dating back to Dantzig et al.’s paper on solving the traveling salesman problem, commentaries on those papers, and a selection of current surveys, including topics such as reformulations, symmetry, nonlinear integer programming and much more. The volume includes a DVD of presentations made at a 2008 conference held in Aussois, France.

Integer Programming Formulation

There are relatively few books on formulating problems. An exception is [Williams \(1999\)](#). In addition, most operations research textbooks offer examples and exercises on formulations, though many of the examples are not of realistic size. Some choices are [Winston \(2003\)](#), [Taha \(2010\)](#), and [Hillier and Lieberman \(2009\)](#).

Branch and Bound

Branch and bound can be traced back to the 1960s and the work of [Land and Doig \(1960\)](#). Most basic textbooks (see above) give an outline of the method (at the level given here).

Branch and Cut

Cutting plane approaches date back to the late 1950s and the work of [Gomory \(1958\)](#), whose cutting planes are applicable to any integer program. [Jünger et al. \(1995\)](#) provides a survey of the use of cutting plane algorithms for specialized problem classes.

As a computational technique, the work of [Crowder et al. \(1983\)](#) showed how cuts could greatly improve basic branch and bound.

For an example of the success of such approaches for solving extremely large optimization problems, see [Applegate et al. \(2011\)](#).

Branch and Price

[Barnhart et al. \(1998\)](#) give an excellent survey of this approach.

Implementations

There are a number of very good implementations that allow the optimization of realistic integer programs. Some of these are commercial, like IBM's ILOG CPLEX implementation, currently up to version 12.4.¹ [Bixby et al. \(1999\)](#) give a detailed description of the advances that this software has made.

¹ <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>

Another commercial product is FICO's Xpress-MP (<http://www.fico.com/en/~Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx>), with the textbook by Gueret et al. (2002) providing a very nice set of examples and applications.

A final major commercial product is the code by Gurobi (<http://www.gurobi.com>). While the initial version of this code was released in 2008, this product has quickly become competitive with its older rivals.

COIN-OR (<http://www.coin-or.org>) provides an open-source initiative for optimization, including integer programming. Other approaches are described by Ralphs and Ladanyi (1999) and Cordier et al. (1999).

References

- Applegate DL, Bixby RE, Chvatal V, Cook WJ (2011) The traveling salesman problem: a computational study. Princeton University Press
- Barnhart C, Johnson EL, Nemhauser GL, Savelsbergh MWP, Vance PH (1998) Branch-and-price: column generation for huge integer programs. Oper Res 46:316
- Bixby RE, Fenelon M, Gu Z, Rothberg E, Wunderling R (1999) MIP: theory and practice—closing the gap. In: Proceedings of the 19th IFIP TC7 conference on system modelling, Cambridge. Kluwer, Dordrecht, pp 19–50
- Common Optimization INterface for Operations Research (COIN-OR) (2004) <http://www.coin-or.org>
- Cordier C, Marchand H, Laundy R, Wolsey LA (1999) bc-opt: a branch-and-cut code for mixed integer programs. Math Program 86:335
- Crowder H, Johnson EL, Padberg MW (1983) Solving large scale zero-one linear programming problems. Oper Res 31:803–834
- Gomory RE (1958) Outline of an algorithm for integer solutions to linear programs. Bull Am Math Soc 64(5):275–278
- Gueret C, Prins C, Sevaux M (2002) Applications of optimization with Xpress-MP (trans: Heipcke S). Dash Optimization, Blisworth
- Hillier FS, Lieberman GJ (2009) Introduction to operations research, 9th edn. McGraw-Hill, New York
- Jünger M, Reinelt G, Thienel S (1995) Practical problem solving with cutting plane algorithms in combinatorial optimization. DIMACS series in discrete mathematics and theoretical computer science. AMS, Providence, p 111
- Jünger M, Liebling T, Naddef D, Nemhauser G, Pulleyblank W, Reinelt G, Rinaldi G, Wolsey L (2010) 40 years of integer programming 1958–2008. Springer, Heidelberg
- Land AH, Doig AG (1960) An automatic method for solving discrete programming problems. Econometrica 28:83–97
- Nemhauser GL, Wolsey LA (1998) Integer and combinatorial optimization. Wiley, New York

- Ralphs TK, Ladanyi L (1999) SYMPHONY: a parallel framework for branch and cut. White paper, Rice University
- Schrijver A (1998) Theory of linear and integer programming. Wiley, New York
- Taha HA (2010) Operations research: an introduction, 9th edn. Prentice-Hall, New York
- Williams HP (1999) Model building in mathematical programming. Wiley, New York
- Winston W (2003) Operations research: applications and algorithms, 4th edn. Thomson, New York
- Wolsey LA (1998) Integer programming. Wiley, New York

Chapter 4

Genetic Algorithms

Kumara Sastry, David E. Goldberg, and Graham Kendall

4.1 Introduction

Genetic algorithms (GAs) are search methods based on principles of natural selection and genetics (Fraser 1957; Bremermann 1958; Holland 1975). We start with a brief introduction of simple GAs and the associated terminologies. GAs encode the decision variables of a search problem into finite-length strings of alphabets of certain cardinality. The strings which are candidate solutions to the search problem are referred to as *chromosomes*, the alphabets are referred to as *genes* and the values of genes are called *alleles*. For example, in a problem such as the traveling salesman problem (TSP), a chromosome represents a route, and a gene may represent a city. In contrast to traditional optimization techniques, GAs work with coding of parameters, rather than the parameters themselves.

To evolve good solutions and to implement natural selection, we need a measure for distinguishing good solutions from bad solutions. The measure could be an *objective* function that is a mathematical model or a computer simulation, or it can be a *subjective* function where humans choose better solutions over worse ones. In essence, the fitness measure must determine a candidate solution's relative fitness, which will subsequently be used by the GA to guide the evolution of good solutions.

K. Sastry
University of Illinois, Urbana-Champaign, IL, USA
Current Affiliation: Intel Corp.

D.E. Goldberg
ThreeJoy Associates, Inc. and University of Illinois, Urbana-Champaign, IL, USA

G. Kendall (✉)
Automated Scheduling, Optimization and Planning Research Group, School of Computer Science,
University of Nottingham, Nottingham, UK

Automated Scheduling, Optimization and Planning Research Group, School of Computer Science,
University of Nottingham, Semenyih, Malaysia
e-mail: Graham.Kendall@nottingham.edu.my

Another important concept of GAs is the notion of population. Unlike traditional search methods, GAs rely on a population of candidate solutions. The population size, which is usually a user-specified parameter, is one of the important factors affecting the scalability and performance of GAs. For example, small population sizes might lead to premature convergence and yield substandard solutions. On the other hand, large population sizes lead to unnecessary expenditure of valuable computational time.

Once the problem is encoded in a chromosomal manner and a fitness measure for discriminating good solutions from bad ones has been chosen, we can start to *evolve* solutions to the search problem using the following steps:

1. **Initialization:** The initial population of candidate solutions is usually generated randomly across the search space. However, domain-specific knowledge or other information can be easily incorporated in the generation of the initial population.
2. **Evaluation:** Once the population is initialized, or an offspring population is created, the fitness values of the candidate solutions are evaluated.
3. **Selection:** Selection allocates more copies to solutions with better fitness values and thus imposes the survival-of-the-fittest mechanism on the candidate solutions. The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, ranking selection and tournament selection, some of which are described in the next section.
4. **Recombination:** Recombination combines bits and pieces of two or more parental solutions to create new, possibly better solutions (i.e. offspring). There are many ways of accomplishing this (some of which are discussed in the next section), and achieving competent performance depends on getting the recombination mechanism designed properly; but the primary idea to keep in mind is that the offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner (Goldberg 2002).
5. **Mutation:** While recombination operates on two or more parental chromosomes, mutation, locally but randomly, modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes that are made to an individual's trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution.
6. **Replacement:** The offspring population created by selection, recombination, and mutation replaces the original parental population. Many replacement techniques such as elitist replacement, generation-wise replacement and steady-state replacement methods are used in GAs.
7. Repeat steps 2–6 until one or more stopping criteria are met.

Goldberg (1983, 1999a, 2002) has likened GAs to mechanistic versions of certain modes of human innovation and has shown that, although these operators when analyzed individually are ineffective, when combined together they can work well. This aspect has been explained with the concepts of *fundamental intuition* and *innovation intuition*. The same study compares a combination of selection and mutation to *continual improvement* (a form of hill climbing), and the combination of selec-

tion and recombination to *innovation* (cross-fertilizing). These analogies have been used to develop a design decomposition methodology and so-called *competent* genetic algorithms — GAs that solve hard problems quickly, reliably, and accurately — both of which are discussed in subsequent sections.

This chapter is organized as follows. The next section provides details of the individual steps of a typical GA and introduces several popular genetic operators. Section 4.3 presents a principled methodology of designing competent GAs based on decomposition principles. Section 4.4 gives a brief overview of designing principled efficiency enhancement techniques to speed up genetic and evolutionary algorithms.

4.2 Basic GA Operators

In this section we describe some of the selection, recombination and mutation operators commonly used in GAs.

4.2.1 Selection Methods

Selection procedures can be broadly classified into two classes (Goldberg 1989): fitness proportionate selection and ordinal selection.

4.2.1.1 Fitness Proportionate Selection

This includes methods such as roulette-wheel selection and stochastic universal selection. In roulette-wheel selection, each individual in the population is assigned a roulette-wheel slot sized in proportion to its fitness. That is, in the biased roulette wheel, good solutions have larger slot sizes than the less fit solutions. The roulette wheel is spun to obtain a reproduction candidate. The roulette-wheel selection scheme can be implemented as follows:

1. Evaluate the fitness, f_i , of each individual in the population.
2. Compute the probability (slot size), p_i , of selecting each member of the population: $p_i = f_i / \sum_{j=1}^n f_j$, where n is the population size.
3. Calculate the cumulative probability, q_i , for each individual: $q_i = \sum_{j=1}^i p_j$.
4. Generate an uniform random number, $r \in (0, 1]$.
5. If $r < q_1$ then select the first chromosome, x_1 , else select the individual x_i such that $q_{i-1} < r \leq q_i$.
6. Repeat steps 4–5 n times to create n candidates in the mating pool.

To illustrate, consider a population with five individuals ($n = 5$), with the fitness values as shown in the table below. The total fitness, $\sum_{j=1}^n f_j = 28 + 18 + 14 + 9 +$

$26 = 95$. The probability of selecting an individual and the corresponding cumulative probabilities are also shown.

Chromosome #	1	2	3	4	5
Fitness, f	28	18	14	9	26
Probability, p_i	$28/95 = 0.295$	0.189	0.147	0.095	0.274
Cumulative probability, q_i	0.295	0.484	0.631	0.726	1.000

Now if we generate a random number r , say 0.585, then the third chromosome is selected as $q_2 = 0.484 < 0.585 \leq q_3 = 0.631$.

4.2.1.2 Ordinal Selection

Ordinal selection includes methods such as tournament selection ([Goldberg et al. 1989](#)) and truncation selection ([Mühlenbein and Schlierkamp-Voosen 1993](#)). In tournament selection, s chromosomes are chosen at random (either with or without replacement) and entered into a tournament against each other. The fittest individual in the group of s chromosomes wins the tournament and is selected as the parent. The most widely used value of s is 2. Using this selection scheme, n tournaments are required to choose n individuals. In truncation selection, the top $(1/s)$ th of the individuals get s copies each in the mating pool.

4.2.2 Recombination Operators

After selection, individuals from the mating pool are recombined (or crossed over) to create new, hopefully better offsprings. In the GA literature, many crossover methods have been designed ([Goldberg 1989](#); [Booker et al. 1997](#)) and some of them are described in this section. Many of the recombination operators used in the literature are problem specific and in this section we will introduce a few generic (problem independent) crossover operators. It should be noted that while for *hard* search problems many of the following operators are not scalable, they are very useful as a first option. Recently, however, researchers have achieved significant success in designing scalable recombination operators that adapt linkage, which is briefly discussed in Sect. 4.3.

In most recombination operators, two individuals are randomly selected and are recombined with a probability p_c , called crossover probability. That is, a uniform random number, r , is generated and if $r \leq p_c$, the two randomly selected individuals undergo recombination. Otherwise, that is if $r > p_c$, the two offspring are simply copies of their parents. The value of p_c can either be set experimentally, or can be set based on schema theorem principles ([Goldberg 1989, 2002](#)).

k-point crossover: One-point and two-point crossovers are among the simplest and most widely applied crossover methods. In one-point crossover, illustrated

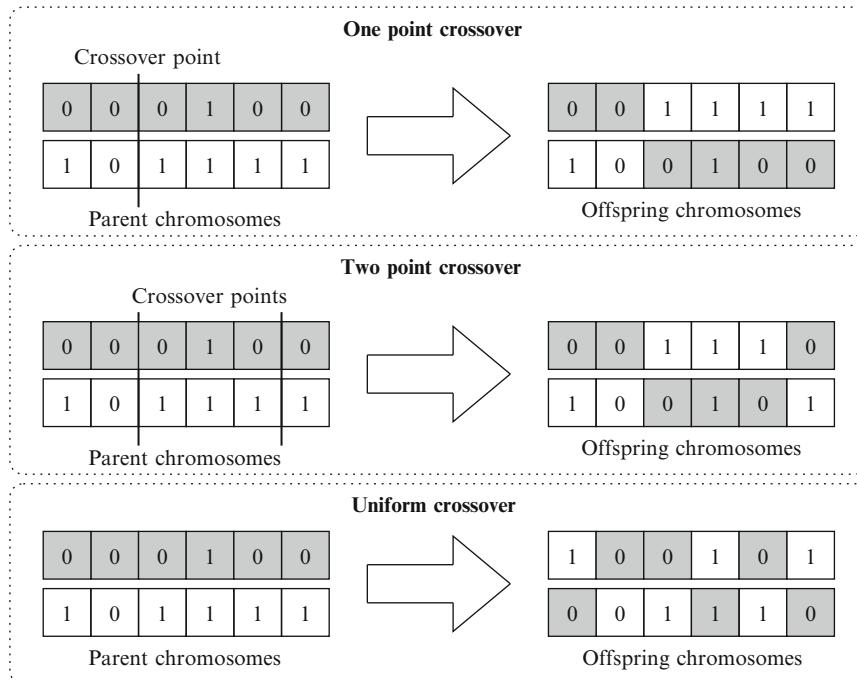


Fig. 4.1 Illustration of one-point, two-point, and uniform crossover methods

in Fig. 4.1, a crossover site is selected at random over the string length, and the alleles on one side of the site are exchanged between the individuals. In two-point crossover, two crossover sites are randomly selected. The alleles between the two sites are exchanged between the two randomly paired individuals. Two-point crossover is illustrated in Fig. 4.1. The concept of one-point crossover can be extended to k -point crossover, where k crossover points are used, rather than just one or two.

Uniform crossover: Another common recombination operator is uniform crossover (Syswerda 1989; Spears and De Jong 1994). In uniform crossover, illustrated in Fig. 4.1, every allele is exchanged between a pair of randomly-selected chromosomes with a certain probability, p_e , known as the swapping probability. Usually the swapping probability value is taken to be 0.5.

Uniform order-based crossover: The k -point and uniform crossover methods described above are not well suited for search problems with permutation codes such as those used in the TSP. They often create offsprings that represent invalid solutions for the search problem. Therefore, when solving search problems with permutation codes, a problem-specific repair mechanism is often required (and used) in conjunction with the above recombination methods to always create valid candidate solutions.

Another alternative is to use recombination methods developed specifically for permutation codes, which always generate valid candidate solutions. Several such crossover techniques are described in the following paragraphs starting with the uniform order-based crossover.

Parent P ₁	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr></table>	A	B	C	D	E	F	G
A	B	C	D	E	F	G		
Parent P ₂	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>E</td><td>B</td><td>D</td><td>C</td><td>F</td><td>G</td><td>A</td></tr></table>	E	B	D	C	F	G	A
E	B	D	C	F	G	A		
Template	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	1	0	0	1	0
0	1	1	0	0	1	0		
Child C ₁	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>E</td><td>B</td><td>C</td><td>D</td><td>G</td><td>F</td><td>A</td></tr></table>	E	B	C	D	G	F	A
E	B	C	D	G	F	A		
Child C ₂	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>D</td><td>C</td><td>E</td><td>G</td><td>F</td></tr></table>	A	B	D	C	E	G	F
A	B	D	C	E	G	F		

Fig. 4.2 Illustration of uniform-order crossover

Parent P ₁	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr></table>	A	B	C	D	E	F	G
A	B	C	D	E	F	G		
Parent P ₂	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>C</td><td>B</td><td>G</td><td>E</td><td>F</td><td>D</td><td>A</td></tr></table>	C	B	G	E	F	D	A
C	B	G	E	F	D	A		
Child C ₁	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>?</td><td>?</td><td>C</td><td>D</td><td>E</td><td>?</td><td>?</td></tr></table>	?	?	C	D	E	?	?
?	?	C	D	E	?	?		
Child C ₂	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>?</td><td>?</td><td>G</td><td>E</td><td>F</td><td>?</td><td>?</td></tr></table>	?	?	G	E	F	?	?
?	?	G	E	F	?	?		
Child C ₁	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>F</td><td>A</td><td>C</td><td>D</td><td>E</td><td>B</td><td>G</td></tr></table>	F	A	C	D	E	B	G
F	A	C	D	E	B	G		
Child C ₂	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>C</td><td>D</td><td>G</td><td>E</td><td>F</td><td>A</td><td>B</td></tr></table>	C	D	G	E	F	A	B
C	D	G	E	F	A	B		

Fig. 4.3 Illustration of order-based crossover

In uniform order-based crossover, two parents (say P₁ and P₂) are randomly selected and a random binary template is generated (see Fig. 4.2). Some of the genes for offspring C₁ are filled by taking the genes from parent P₁ where there is a one in the template. At this point we have C₁ partially filled, but it has some “gaps”. The genes of parent P₁ in the positions corresponding to zeros in the template are taken and sorted in the same order as they appear in parent P₂. The sorted list is used to fill the gaps in C₁. Offspring C₂ is created using a similar process (see Fig. 4.2).

Order-based crossover: The order-based crossover (Davis 1985) is a variation of the uniform order-based crossover in which two parents are randomly selected and two random crossover sites are generated (see Fig. 4.3). The genes between the cut points are copied to the children. Starting from the second crossover site, copy the genes that are not already present in the offspring from the alternative parent (the parent other than the one whose genes are copied by the offspring

in the initial phase) in the order they appear. For example, as shown in Fig. 4.3, for offspring C_1 , since alleles C, D and E are copied from the parent P_1 , we get alleles B, G, F and A from the parent P_2 . Starting from the second crossover site, which is the sixth gene, we copy alleles B and G as the sixth and seventh genes respectively. We then wrap around and copy alleles F and A as the first and second genes.

Partially matched crossover (PMX): As well as always generating valid offspring, the PMX operator ([Goldberg and Lingle 1984](#)) also preserves orderings within the chromosome. In PMX, two parents are randomly selected and two random crossover sites are generated. Alleles within the two crossover sites of a parent are exchanged with the alleles corresponding to those mapped by the other parent. For example, as illustrated in Fig. 4.4 (reproduced with permission from [Goldberg 1989](#)), looking at parent P_1 , the first gene within the two crossover sites, 5, maps to 2 in P_2 . Therefore, genes 5 and 2 are swapped in P_1 . Similarly we swap 6 and 3, and 10 and 7 to create the offspring C_1 . After all exchanges it can be seen that we have achieved a duplication of the ordering of one of the genes in between the crossover point within the opposite chromosome, and vice versa.

Cycle crossover (CX): We describe cycle crossover ([Oliver et al. 1987](#)) with the help of a simple illustration (reproduced with permission from [Goldberg 1989](#)). Consider two randomly selected parents P_1 and P_2 as shown in Fig. 4.5 that are solutions to a TSP problem. The offspring C_1 receives the first variable (representing city 9) from P_1 . We then choose the variable that maps onto the same position in P_2 . Since city 9 is chosen from P_1 which maps to city 1 in P_2 , we choose city 1 and place it into C_1 in the same position as it appears in P_1 (fourth gene), as shown in Fig. 4.5. City 1 in P_1 now maps to city 4 in P_2 , so we place city 4 in C_1 in the same position it occupies in P_1 (sixth gene). We continue this process once more and copy city 6 to the 9th gene of C_1 from P_1 . At this point, since city 6 in P_1 maps to city 9 in P_2 , we should take city 9 and place it in C_1 , but this has already been done, so we have completed a cycle; which is where this operator gets its name. The missing cities in offspring C_1 are filled from P_2 . Offspring C_2 is created in the same way by starting with the first city of parent P_2 (see Fig. 4.5).

4.2.3 Mutation Operators

If we use a crossover operator, such as one-point crossover, we may get better and better chromosomes but the problem is, if the two parents (or worse—the entire population) have the same allele at a given gene then one-point crossover will not

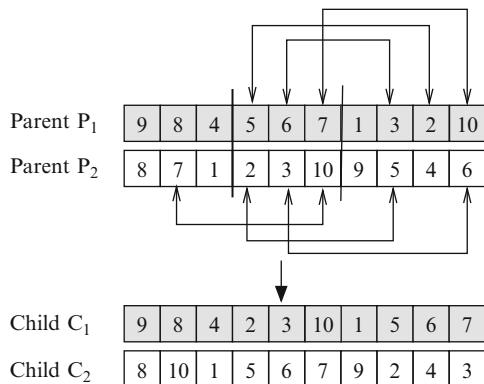


Fig. 4.4 Illustration of partially matched crossover

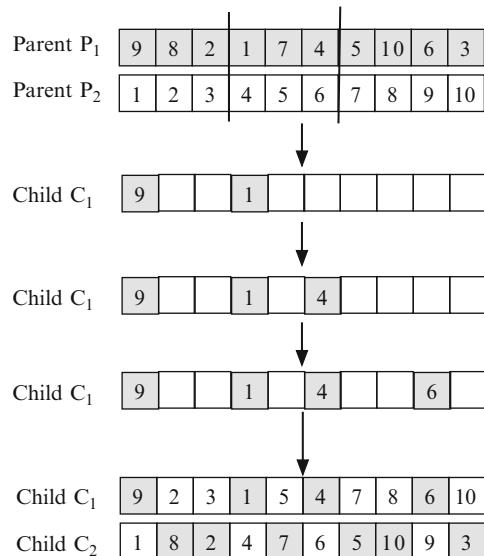


Fig. 4.5 Illustration of cycle crossover

change that. In other words, that gene will have the same allele forever. Mutation is designed to overcome this problem in order to add diversity to the population and ensure that it is possible to explore the entire search space.

In evolution strategies, mutation is the primary variation/search operator. For an introduction to evolution strategies see, for example, Bäck (1996). Unlike evolution strategies, mutation is often the secondary operator performed with a low probability in GAs. One of the most common mutations is the bit-flip mutation. In bit-flip mutation, each bit in a binary string is changed (a 0 is converted to 1, and vice versa) with a certain probability, p_m , known as the mutation probability. As mentioned earlier, mutation performs a random walk in the vicinity of the individual. Other mutation operators such as those that swap genes and problem-specific ones can also be developed and are often used in the literature.

4.2.4 Replacement

Once the new offspring solutions are created using crossover and mutation, we need to introduce them into the parental population. There are many ways we can approach this. Bear in mind that the parent chromosomes have already been selected according to their fitness, so we are hoping that the children (which includes parents which did not undergo crossover) are among the fittest in the population and so we would hope that the population will gradually, on average, increase their fitness. Some of the most common techniques are outlined below.

Delete-all: This technique deletes all the members of the current population and replaces them with the same number of chromosomes that have just been created. This is probably the most common technique and will be the technique of choice for most people due to its relative ease of implementation. It is also parameter free, which is not the case for those listed below.

Steady-state: This technique deletes n old members and replaces them with n new members. The number to delete and replace, n , at any one time is a parameter to this deletion technique. Another consideration for this technique is deciding which members to delete from the current population. Do you delete the worst individuals, pick them at random or delete the chromosomes that you used as parents? Again, this is a parameter to this technique.

Steady-state-no-duplicates: This is the same as the steady-state technique but the algorithm checks that no duplicate chromosomes are added to the population. This adds to the computational overhead but can mean that more of the search space is explored.

4.3 Competent GAs

While using innovation for explaining working mechanisms of GAs is very useful, as a design metaphor it poses difficulty as the processes of innovation are themselves not well understood. However, if we want GAs to successfully solve increasingly difficult problems across a wide spectrum of areas, we need a principled, but mechanistic way of designing GAs. The last few decades have witnessed great strides, not only toward the development of so-called *competent* genetic algorithms—GAs that solve hard problems, quickly, reliably and accurately (Goldberg 1999a). From a computational standpoint, the existence of competent GAs suggests that many difficult problems can be solved in a scalable fashion. Furthermore, it significantly reduces the burden on a user to decide on a good coding or a good genetic operator that accompanies many GA applications. If the GA can adapt to the problem, there is less reason for the user to have to adapt the problem, coding or operators to the GA.

In this section we briefly review some of the key lessons of competent GA design. Specifically, we restrict the discussion to selectorecombinative GAs and focus on the

cross-fertilization type of innovation. Using Holland's notion of a building block (Holland 1975), the first author proposed decomposing the problem of designing a competent selectorecombinative GA (Goldberg et al. 1992). This design decomposition has been explained in detail elsewhere (Goldberg 2002), but is briefly reviewed in what follows.

Know that GAs process building blocks (BBs): The primary idea of selectorecombinative GA theory is that GAs work through a mechanism of *decomposition* and *reassembly*. Holland (1975) identified well-adapted sets of features that were components of effective solutions (BBs). The basic idea is that GAs (1) implicitly identify BBs or sub-assemblies of good solutions, and (2) recombine different sub-assemblies to form very high performance solutions.

Understand BB hard problems: From the standpoint of cross-fertilizing innovation, problems that are hard have BBs that are hard to acquire. This may be because the BBs are complex, hard to find, or because different BBs are hard to separate, or because low-order BBs may be *misleading* or *deceptive* (Goldberg 1987, 2002).

Understand BB growth and timing: Another key idea is that BBs or notions exist in a kind of competitive *market economy of ideas*, and steps must be taken to ensure that the best ones (1) grow and take over a dominant market share of the population, and (2) the growth rate is neither too fast nor too slow.

The growth in market share can be easily satisfied by appropriately setting the crossover probability, p_c , and the selection pressure, s (Goldberg and Sastry 2001)

$$p_c \leq \frac{1 - s^{-1}}{\epsilon}, \quad (4.1)$$

where ϵ is the probability of BB disruption.

Two other approaches have been used in understanding time. It is not appropriate in a basic text like this to describe them in detail, but we give a few example references for the interested reader:

- Takeover time models, where the dynamics of the best individual is modeled (Goldberg and Deb 1991; Bäck 1994; Sakamoto and Goldberg 1997; Cantú-Paz 1999; Rudolph 2000).
- Selection-intensity models, where the dynamics of the average fitness of the population is modeled (Mühlenbein and Schlierkamp-Voosen 1993; Bäck 1995; Thierens and Goldberg 1994; Miller and Goldberg 1995, 1996a).

The time models suggest that for a problem of size ℓ , with all BBs of equal importance or salience, the convergence time of GAs is given by Miller and Goldberg (1995)

$$t_c = \frac{\pi}{2I} \sqrt{\ell}, \quad (4.2)$$

where I is the selection intensity (Bulmer 1985), which is a parameter dependent on the selection method and selection pressure.

On the other hand, if the BBs of a problem have different salience, then the convergence time scales up differently. For example, when BBs of a problem are exponentially scaled, with a particular BB being exponentially better than the others, then the convergence time of a GA is linear with the problems size (Thierens et al. 1998):

$$t_c = \frac{-\log 2}{\log(1 - I/\sqrt{3})} \ell. \quad (4.3)$$

To summarize, the convergence time of GAs scale up as $O(\sqrt{\ell}) - O(\ell)$ (see Chap. 1 for an explanation of the O notation).

Understand BB supply and decision making: One role of the population is to ensure an adequate *supply* of the raw building blocks in a population. Randomly generated populations of increasing size will, with higher probability, contain larger numbers of more complex BBs (Holland 1975; Goldberg et al. 2001). For a problem with m building blocks, each consisting of k alphabets of cardinality χ , the population size required to ensure the presence of at least one copy of all the raw building blocks is given by Goldberg et al. (2001)

$$n = \chi^k \log m + k\chi^k \log \chi. \quad (4.4)$$

Just ensuring the raw supply is not enough, decision making among different, competing notions (BBs) is *statistical* in nature, and as we increase the population size, we increase the likelihood of making the best possible decisions (Goldberg et al. 1992; Harik et al. 1999). For an additively decomposable problem with m building blocks of size k each, the population size required to not only ensure supply, but also ensure correct decision making is approximately given by Harik et al. (1999)

$$n = -\frac{\sqrt{\pi}}{2} \frac{\sigma_{BB}}{d} 2^k \sqrt{m} \log \alpha, \quad (4.5)$$

where d/σ_{BB} is the signal-to-noise ratio (Goldberg et al. 1992), and α is the probability of incorrectly deciding among competing building blocks. In essence, the population-sizing model consists of the following components:

- *Competition complexity*, quantified by the total number of competing BBs, 2^k
- *Subcomponent complexity*, quantified by the number of BBs, m
- *Ease of decision making*, quantified by the signal-to-noise ratio, d/σ_{bb}
- *Probabilistic safety factor*, quantified by the coefficient $-\log \alpha$.

On the other hand, if the BBs are exponentially scaled, the population size scales as (Thierens et al. 1998; Goldberg 2002)

$$n = -c_o \frac{\sigma_{BB}}{d} 2^k m \log \alpha, \quad (4.6)$$

where c_o is a constant dependent on the drift effects (Goldberg and Segrest 1987; Asoh and Mühlenbein 1994).

To summarize, the population size required by GAs scales up as $O(2^k \sqrt{m}) - O(2^k m)$.

Identify BBs and exchange them: Perhaps the most important lesson of current research in GAs is that the *identification and exchange of BBs* is the critical path to innovative success. First-generation GAs, usually fail in their ability to promote this exchange reliably. The primary design challenge to achieving competence is the need to identify and promote effective BB exchange. Theoretical studies using a *facetwise* modeling approach (Goldberg et al. 1993b; Thierens 1999; Sastry and Goldberg 2003) have shown that while fixed recombination operators such as uniform crossover, due to inadequacies of effective identification and exchange of BBs, demonstrate polynomial scalability on simple problems, they scale up exponentially with problem size on boundedly difficult problems. The mixing models also yield a *control map* delineating the region of good performance for a GA. Such a control map can be a useful tool in visualizing GA sweet-spots and provide insights into parameter settings (Goldberg 1999a). This is in contrast to recombination operators that can automatically and adaptively identify and exchange BBs, which scale up polynomially (subquadratically–quadratically) with problem size.

Efforts in principled design of effective BB identification and exchange mechanisms have led to the development of competent GAs. Competent GAs are a class of GAs that solve hard problems quickly, reliably and accurately. Hard problems, are loosely defined as those problems that have large sub-solutions that cannot be decomposed into simpler sub-solutions, or have badly scaled sub-solutions, or have numerous local optima, or are subject to a high stochastic noise. While designing a competent GA, the objective is to develop a GA that can solve problems with bounded difficulty and exhibit a polynomial (usually subquadratic) scale-up with the problem size.

Interestingly, the mechanics of competent GAs vary widely, but the principles of innovative success are invariant. Competent GA design began with the development of the *messy genetic algorithm* (Goldberg et al. 1989), culminating in 1993 with the *fast messy GA* (Goldberg et al. 1993a). Since those early scalable results, a number of competent GAs have been constructed using different mechanism styles. We categorize these approaches and provide some references for the interested reader, but a detailed treatment is beyond the scope of this chapter:

- *Perturbation techniques* such as the messy GA (mGA) (Goldberg et al. 1989), fast messy GA (fmGA) (Goldberg et al. 1993a), gene expression messy GA (GEMGA) (Kargupta 1996), linkage identification by nonlinearity check/linkage identification by detection GA (LINC/LIMD GA) (Munetomo and Goldberg 1999; Heckendorn and Wright 2004), and dependency structure matrix driven genetic algorithm (DSMGA) (Yu 2006).
- *Linkage adaptation techniques* such as linkage learning GA (LLGA) (Chen 2004; Harik and Goldberg 1997).
- *Probabilistic model building techniques* (Pelikan et al. 2006) such as population-based incremental learning (PBIL) (Baluja 1994), the univariate model building algorithm (UMDA) (Mühlenbein and Paass 1996), the compact GA (CGA) (Harik et al. 1998), extended compact GA (eCGA) (Harik 1999), the Bayesian

optimization algorithm (BOA) (Pelikan et al. 2000), the iterated distribution estimation algorithm (IDEA) (Bosman and Thierens 1999) and the hierarchical Bayesian optimization algorithm (hBOA) (Pelikan 2005).

4.4 Efficiency Enhancement of Genetic Algorithms

The previous section presented a brief account of competent GAs. These GA designs have shown promising results and have successfully solved hard problems requiring only a subquadratic number of function evaluations. In other words, competent GAs usually solve an ℓ -variable search problem, requiring only $O(\ell^2)$ number of function evaluations. While competent GAs take problems that were intractable with first-generation GAs and render them tractable, for large-scale problems, the task of computing even a subquadratic number of function evaluations can be daunting. If the fitness function is a complex simulation, model or computation, then a single evaluation might take hours, even days. For such problems, even a subquadratic number of function evaluations is very high. For example, consider a 20-bit search problem and assume that a fitness evaluation takes 1 hour. It requires about a month to solve the problem. This places a premium on a variety of *efficiency enhancement techniques*. Also, it is often the case that a GA needs to be integrated with problem-specific methods in order to make the approach effective for a particular problem. The literature contains a large number of papers which discuss enhancements of GAs. Once again, a detailed discussion is well beyond the scope of the chapter, but we provide four broad categories of GA enhancement and examples of appropriate references for the interested reader:

Parallelization, where GAs are run on multiple processors and the computational resource is distributed among these processors (Cantú-Paz 2000). Evolutionary algorithms are by *nature* parallel, and many different parallelization approaches such as a simple master–slave, coarse-grained fine-grained or hierarchical architectures can be readily used. Regardless of how parallelization is done, the key idea is to distribute the computational load on several processors thereby speeding-up the overall GA run. Moreover, there exists a principled design theory for developing an efficient parallel GA and optimizing the key facts of parallel architecture, connectivity, and deme size (Cantú-Paz 2000).

For example, when the function evaluation time, T_f , is much greater than the communication time, T_c , which is very often the case, then a simple master–slave parallel GA—where the fitness evaluations are distributed over several processors and the rest of the GA operations are performed on a single processor—can yield linear speed-up when the number of processors is less than or equal to $\sqrt[3]{\frac{T_f}{T_c}} n$, and an optimal speed-up when the number of processors equals $\sqrt{\frac{T_f}{T_c}} n$, where n is the population size.

Hybridization can be an extremely effective way of improving the performance of GAs. The most common form of hybridization is to couple GAs with local search techniques and to incorporate domain-specific knowledge into the

search process. A common form of hybridization is to incorporate a local search operator into the GA by applying the operator to each member of the population after each generation. This hybridization is often carried out in order to produce stronger results than the individual approaches can achieve on their own. However, this improvement in solution quality usually comes at the expense of increased computational time (e.g. [Burke et al. 2001](#)). Such approaches are often called memetic algorithms in the literature. This term was first used by [Moscato \(1989\)](#) and has since been employed widely. For more details about memetic algorithms, see [Krasnogor and Smith \(2005\)](#), [Krasnogor et al. \(2004\)](#), [Moscato and Cotta \(2003\)](#) and [Moscato \(1999\)](#). Of course, the hybridization of GAs can take other forms. Examples include:

- Initializing a GA population ([Burke et al. 1998](#); [Fleurent and Ferland 1993](#); [Watson et al. 1999](#));
- Repairing infeasible solutions into legal ones ([Ibaraki 1997](#));
- Developing specialized heuristic recombination operators ([Burke et al. 1995](#));
- Incorporating a case-based memory (experience of past attempts) into the GA process ([Louis and McDonnell 2004](#));
- Heuristically decomposing large problems into smaller sub-problems before employing a memetic algorithm ([Burke and Newell 1999](#)).

Hybrid GA and memetic approaches have demonstrated significant success in difficult real-world application areas. A small number of examples are included below (many more examples can be found in the literature):

- University timetabling: examination timetabling ([Burke et al. 1996, 1998](#); [Burke and Newell 1999](#)) and course timetabling ([Paechter et al. 1995, 1996](#));
- Machine scheduling ([Cheng and Gen 1997](#)). Electrical power systems: unit commitment problems ([Valenzuela and Smith 2002](#));
- Electricity transmission network maintenance scheduling ([Burke and Smith 1999](#)); thermal generator maintenance scheduling ([Burke and Smith 2000](#));
- Sports scheduling ([Costa 1995](#));
- Nurse rostering ([Burke et al. 2001](#));
- Warehouse scheduling ([Watson et al. 1999](#)).

While GA practitioners have often understood that real-world or commercial applications often require hybridization, there has been limited effort devoted to developing a theoretical underpinning of genetic algorithm hybridization. However, the following list contains examples of work which has aimed to answer critical issues such as

- The optimal division of labor between global and local searches (or the right mix of exploration and exploitation) ([Goldberg and Voessner 1999](#); [Sinha 2003](#));
- The effect of local search on sampling ([Hart and Belew 1996](#));
- Hybrid GA modeling issues ([Whitley 1995](#)).

The papers cited in this section are only a tiny proportion of the literature on hybrid GAs but they should provide a starting point for the interested reader. However, although there is a significant body of literature on the subject, there are many research directions still to be explored. Indeed, considering the option of hybridizing a GA with other approaches is one of the suggestions we give in the *Tricks of the Trade* section at the end of the chapter.

Time continuation, where capabilities of both mutation and recombination are optimally utilized to obtain a solution of as high quality as possible with a given limited computational resource ([Goldberg 1999b](#); [Sastry and Goldberg 2004](#)). Time utilization (or continuation) exploits the tradeoff between the search for solutions with large populations and a single convergence epoch or using a small population with multiple convergence epochs.

Early theoretical investigations indicate that when the BBs are of equal (or nearly equal) salience and both recombination and mutation operators have the linkage information, then a small population with multiple convergence epochs is more efficient. However, if the fitness function is noisy or has overlapping BBs, then a large population with single convergence epoch is more efficient ([Sastry and Goldberg 2004](#)). On the other hand, if the BBs of the problem are of non-uniform salience, which essentially require serial processing, then a small population with multiple convergence epochs is more efficient ([Goldberg 1999b](#)). Much work needs to be done to develop a principled design theory for efficiency enhancement via time continuation and to design competent continuation operators to reinitialize population between epochs.

Evaluation relaxation, where an accurate but computationally expensive fitness evaluation is replaced with a less accurate, but computationally inexpensive fitness estimate. The low-cost, less-accurate fitness estimate can either be (1) *exogenous*, as in the case of surrogate (or approximate) fitness functions ([Jin 2005](#)), where external means can be used to develop the fitness estimate, or (2) *endogenous*, as in the case of *fitness inheritance* ([Smith et al. 1995](#)) where the fitness estimate is computed internally based on parental fitnesses.

Evaluation relaxation in GAs dates back to the early, largely empirical, work of [Grefenstette and Fitzpatrick \(1985\)](#) in image registration ([Fitzpatrick et al. 1984](#)) where significant speed-ups were obtained by reduced random sampling of the pixels of an image. Approximate models have since been used extensively to solve complex optimization problems in many engineering applications such as aerospace and structural engineering ([Barthelemy and Haftka 1993](#); [Dennis and Torczon 1997](#)).

While early evaluation relaxation studies were largely empirical in nature, design theories have since been developed to understand the effect of approximate surrogate functions on population sizing and convergence time and to optimize speed-ups in approximate fitness functions with known variance ([Miller and Goldberg 1996b](#)), in integrated fitness functions ([Albert 2001](#)), in simple functions of known variance or known bias ([Sastry 2001](#)) and also in fitness inheritance ([Sastry et al. 2001, 2004](#); [Pelikan and Sastry 2004](#)).

Speed-up obtained by employing an efficiency-enhancement technique (EET) is measured in terms of a ratio of the computation effort required by a GA when the EET is used to that required by GA in the absence of the EET. That is, $\eta = T_{\text{base}}/T_{\text{efficiency-enhanced}}$. Speed-up obtained by employing even a single EET can potentially be significant. Furthermore, assuming that the performance of one of the above methods does not affect the performance of others, if we employ more than one EET, the overall speed-up is the product of individual speed-ups. That is, if the speed-ups obtained by employing parallelization, hybridization, time continuation and evaluation relaxation be η_p , η_h , η_t , and η_e respectively. If one uses all these EETs, then the overall speed-up obtained is

$$\eta_{\text{total}} = \eta_p \eta_h \eta_t \eta_e.$$

Even if the speed-up obtained by a single EET is modest, a combination of two or more EETs can yield a significant speed-up. For example, if we use a parallel GA that yields linear speed-up with 10 processors, and each of the other three EETs makes GAs 25% more efficient, then together they yield a speed-up of $10 * 1.25^3 = 19.5$. That is evaluation relaxation, time continuation and hybridization would give slightly more than 9.5 processors' worth of additional computation power. GAs designed using the decomposition principles and principled efficiency enhancement outlined in this chapter have opened doors for efficiently solving routine billion-variable optimization in the increasingly large, complex problems of science ([Sastry et al. 2007](#)).

Tricks of the Trade

In this section we present some suggestions for the reader who is new to the area of GAs and wants to know how best to get started. Fortunately, the ideas behind GAs are intuitive and the basic algorithm is not complex. Here are some basic tips.

- Start by using an “off the shelf” GA. It is pointless developing a complex GA, if your problem can be solved using a simple and standard implementation.
- There are many excellent software packages that allow you to implement a GA very quickly. Many of the introductory texts are supplied with a GA implementation and GA-LIB is probably seen as the software of choice for many people (see below).
- Consider carefully your representation. In the early days, the majority of implementations used a bit representation which was easy to implement. Crossover and mutation were simple. However, many other representations are now used, some utilizing complex data structures. You should carry out some research to determine what is the best representation for your particular problem.
- A basic GA will allow you to implement the algorithm and the only thing you have to supply is an evaluation function. If you can achieve this, then this is the fastest way to get a prototype system up and running. However, you may want

to include some problem-specific data in your algorithm. For example, you may want to include your own crossover operators (in order to guide the search) or you may want to produce the initial population using a constructive heuristic (to give the GA a good starting point).

- In recent times, many researchers have hybridized GAs with other search methods (see Sect. 4.4). Perhaps the most common method is to include a local searcher after the crossover and mutation operators (sometimes known as a memetic algorithm). This local searcher might be something as simple as a hill climber, which acts on each chromosome to ensure it is at a local optimum before the evolutionary process starts again.
- There are many parameters required to run a GA (which can be seen as one of the shortcomings). At a minimum you have the population size, the mutation probability and the crossover probability. The problem with having so many parameters to set is that it can take a lot of experimentation to find a set of values which solves your particular problem to the required quality. A broad rule of thumb, to start with, is to use a mutation probability of 0.05 (De Jong 1975), a crossover rate of 0.6 (De Jong 1975) and a population size of about 50. An alternative method is to set these parameters based on facetwise models (see Sect. 4.3). These three parameters are just an example of the many choices you are going to have to make to get your GA implementation working. To provide just a small sample: Which crossover operator should you use? Which mutation operator? Should the crossover/mutation rates be dynamic and change as the run progresses? Should you use a local search operator? If so, which one, and how long should that be allowed to run for? What selection technique should you use? What replacement strategy should you use? Fortunately, many researchers have investigated many of these issues and the following section *Additional Sources* provides many suitable references.

Sources of Additional Information

At the time of writing, a Google search for Genetic Algorithms returned over seven million hits, so it is impossible to give a comprehensive list of all potentially useful sources. Below, we have suggested a small selection of sources that you might want to look at. We have split them into three areas, web sites, books and journal articles. We apologise for any that we have missed but it is impossible to list every available source.

Web Sites

Due to the nature of the internet, these URLs may not always be available and may move. All those listed were last accessed on 11 November 2012. You should also be

wary of citing URLs as they often lack a peer review mechanism and the page could be changed or disappear altogether.

- http://en.wikipedia.org/wiki/Genetic_algorithm/
- <http://geneticalgorithms.ai-depot.com/>
- <http://illigal.org/>
- <http://lancet.mit.edu/ga/>
- <http://lancet.mit.edu/~mbwall/presentations/IntroToGAs/>
- <http://mathworld.wolfram.com/GeneticAlgorithm.html>
- <http://www.obitko.com/tutorials/genetic-algorithms/>
- <http://www.youtube.com/watch?v=ejxfTy4II6I/>

Books

- Coley, D. A. 1999. *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific.
- Davis, L. D. (ed) 1987. *Genetic Algorithms and Simulated Annealing*. Pitman.
- Davis, L. D. (ed) 1991. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.
- Dawkins, R. 1976. *The Selfish Gene*. Oxford University Press.
- De Jong, K. 2002. *Evolutionary Computation: A Unified Approach*. MIT Press.
- Eiben, A. E., Smith, J. E. 2010. *Introduction to Evolutionary Computing*. Natural Computing Series, Springer.
- Falkenauer, E. 1998. *Genetic Algorithms and Grouping Problems*. Wiley.
- Fogel, D. B. 1998. *Evolutionary Computation The Fossil Record*. IEEE Press.
- Fogel, D. B. 2000. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. 2nd edn, IEEE Press.
- Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.
- Haupt, R. L., Haupt, S. E. 2004. *Practical Genetic Algorithms*. Wiley-Interscience.
- Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press (a second edition was published in 1992).
- Michalewicz, M. 1996. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer.
- Mitchell, M. 1998. *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*. MIT Press.
- Munoz, A. R., Rodriguez, I. G. 2012. *Handbook of Genetic Algorithms: New Research*. Mathematics Research Developments. Nova Science.
- Reeves, C. R., Rowe, J. E. 2002. *Genetic Algorithms—Principles and Perspectives: A Guide to GA Theory*. Operations Research/Computer Science Interfaces Series. Springer.
- Sivanandam, S. N. 2008. *Introduction to Genetic Algorithms*. Springer.
- Vose, M. 1999. *The Simple Genetic Algorithm: Foundations and Theory (Complex Adaptive Systems)*. MIT Press.

Journal Articles

In this section, as well as providing some general references to the evolutionary computation literature, we have also provided a few suggestions for review articles in specific applications as these might be useful introductions which you may not otherwise come across.

- Carter, J. N. 2003. Chapter 3, Introduction to using genetic algorithms. *Developments in Petroleum Science*, 51, 51–76.
- Chatterjee, S., Laudato, M., Lynch, L. A. 1996. Genetic algorithms and their statistical applications: an introduction. *Computational Statistics and Data Analysis*, 22, 633–651.
- Cheng, R., Gen, M., Tsujimura, Y. 1999. A tutorial survey of job-shop scheduling problems using genetic algorithms. Part II: Hybrid genetic search strategies. *Computers and Industrial Engineering*, 36, 343–364.
- Forrest, S. 1993. Genetic algorithms: principles of natural selection applied to computation. *Science*, 261, 872–878.
- Larrañaga, P., Kuijpers, C. M. H., Murga, R. H., Inza, I., Dizdarevic, S. 1999. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13, 129–170.
- Salomon, R. 1996. Re-evaluating genetic algorithm performance under coordinate rotation of benchmark functions. A survey of some theoretical and practical aspects of genetic algorithms. *Biosystems*, 39, 263–278.
- Srinivas, M., Patnaik, L. M. 1994. Genetic algorithms: a survey. *Computer*, 27, 17–26.
- Zang, H., Zhang, S., Hapeshi, K. 2010. A review of nature-inspired algorithms. *Journal of Bionic Engineering*, 7, S232–S237.

References

- Albert LA (2001) Efficient genetic algorithms using discretization scheduling. Master's thesis, University of Illinois at Urbana-Champaign (also IlliGAL report no 2002005)
- Asoh H, Mühlenbein H (1994) On the mean convergence time of evolutionary algorithms without selection and mutation. PPSN 3, pp 98–107
- Bäck T (1994) Selective pressure in evolutionary algorithms: a characterization of selection mechanisms. In: Proceedings of the 1st IEEE conference on evolutionary computation, Orlando, pp 57–62
- Bäck T (1995) Generalized convergence models for tournament—and (μ, λ) —selection. In: Proceedings of 6th international conference on genetic algorithms, Pittsburgh, pp 2–8
- Bäck T (1996) Evolutionary algorithms in theory and practice. Oxford University Press, New York

- Baluja S (1994) Population-based incremental learning: a method of integrating genetic search based function optimization and competitive learning. Technical report CMU-CS-94-163, Carnegie Mellon University
- Barthelemy J-FM, Haftka RT (1993) Approximation concepts for optimum structural design—a review. *Struct Optim* 5:129–144
- Booker LB, Fogel DB, Whitley D, Angeline PJ (1997) Recombination. In: Bäck T, Fogel DB, Michalewicz Z (eds) *The handbook of evolutionary computation*, chap E3.3. IOP Publishing/Oxford University Press, London/Oxford, pp C3.3:1–C3.3:27
- Bosman PAN, Thierens D (1999) Linkage information processing in distribution estimation algorithms. In: *Proceedings of the GECCO*, Orlando, pp 60–67 (also Technical report no UU-CS-1999-10)
- Bremermann HJ (1958) The evolution of intelligence. The nervous system as a model of its environment. Technical report no 1, Department of Mathematics, University of Washington
- Bulmer MG (1985) *The mathematical theory of quantitative genetics*. Oxford University Press, Oxford
- Burke EK, Newell JP (1999) A multi-stage evolutionary algorithm for the timetabling problem. *IEEE Trans Evol Comput* 3:63–74
- Burke EK, Smith AJ (1999) A memetic algorithm to schedule planned maintenance for the national grid. *ACM J Exp Algor* 4. doi:10.1145/347792.347801
- Burke EK, Smith AJ (2000) Hybrid evolutionary techniques for the maintenance scheduling problem. *IEEE Trans Power Syst* 15:122–128
- Burke EK, Elliman DG, Weare RF (1995) Specialised recombinative operators for timetabling problems. In: Fogarty T (ed) *Evolutionary computing AISB workshop 1995*. LNCS 993. Springer, Berlin, pp 75–85
- Burke EK, Newall JP, Weare RF (1996) A memetic algorithm for university exam timetabling. In: Burke EK, Ross P (eds) *The practice and theory of automated timetabling I*. LNCS 1153. Springer, Berlin, pp 241–250
- Burke EK, Newall JP, Weare RF (1998) Initialisation strategies and diversity in evolutionary timetabling. *Evol Comput J* 6:81–103
- Burke EK, Cowling PI, De Causmaecker P, Vanden Berghe G (2001) A mimetic approach to the nurse rostering problem. *Appl Intell* 15:199–214
- Cantu-Paz E (1999) Migration policies and takeover times in parallel genetic algorithms. In: *Proceedings of the GECCO*, Orlando, p 775 (also IlliGAL report no 99008)
- Cantu-Paz E (2000) *Efficient and accurate parallel genetic algorithms*. Kluwer, Boston
- Chen J-H (2004) Theory and applications of efficient multi-objective evolutionary algorithms. Doctoral dissertation, Feng Chia University, Taiwan
- Cheng RW, Gen M (1997) Parallel machine scheduling problems using memetic algorithms. *Comput Indust Eng* 33:761–764
- Costa D (1995) An evolutionary tabu search algorithm and the NHL scheduling problem. *INFOR* 33:161–178

- Davis L (1985) Applying algorithms to epistatic domains. In: Proceedings of the international joint conference on artificial intelligence, Los Angeles, pp 162–164
- De Jong KA (1975) An analysis of the behavior of a class of genetic adaptive systems. Doctoral dissertation, University of Michigan (University microfilm no 76-9381)
- Dennis JE, Torczon V (1997) Managing approximation models in optimization. In: Alexandrov NM, Hussaini MY (eds) Multidisciplinary design optimization: state-of-the-art. SIAM, Philadelphia, pp 330–347
- Fitzpatrick JM, Grefenstette JJ, Van Gucht D (1984) Image registration by genetic search. In: Proceedings of the IEEE southeast conference. IEEE Press, Piscataway, Louisville, KY, pp 460–464
- Fleurent C, Ferland J (1993) Genetic hybrids for the quadratic assignment problem. DIMACS series in mathematics and theoretical computer science. This DIMACS workshop on Quadratic Assignment and Related Problems was held at DIMACS 16:173–188
- Fraser AS (1957) Simulation of genetic systems by automatic digital computers. II. Effects of linkage on rates under selection. *Aust J Biol Sci* 10:492–499
- Goldberg DE (1983) Computer-aided pipeline operation using genetic algorithms and rule learning. Doctoral dissertation, University of Michigan
- Goldberg DE (1987) Simple genetic algorithms and the minimal deceptive problem. In: Davis L (ed) Genetic algorithms and simulated annealing, chap 6. Morgan Kaufmann, Los Altos, pp 74–88
- Goldberg DE (1989) Genetic algorithms in search optimization and machine learning. Addison-Wesley, Reading
- Goldberg DE (1999a) The race, the hurdle, and the sweet spot: lessons from genetic algorithms for the automation of design innovation and creativity. In: Bentley P (ed) Evolutionary design by computers, chap 4. Morgan Kaufmann, San Mateo, pp 105–118
- Goldberg DE (1999b) Using time efficiently: genetic-evolutionary algorithms and the continuation problem. In: Proceedings of the GECCO, Orlando, pp 212–219 (also IlliGAL report no 99002)
- Goldberg DE (2002) Design of innovation: lessons from and for competent genetic algorithms. Kluwer, Boston
- Goldberg DE, Deb K (1991) A comparative analysis of selection schemes used in genetic algorithms. Foundations of genetic algorithms. Morgan Kaufmann, pp 69–93
- Goldberg DE, Lingle R (1984) Alleles, loci, and the TSP. In: Proceedings of the 1st international conference on genetic algorithms, Pittsburgh, pp 154–159
- Goldberg DE, Sastry K (2001) A practical schema theorem for genetic algorithm design and tuning. In: Proceedings of the GECCO, San Francisco, pp 328–335 (also IlliGAL report no 2001017)
- Goldberg DE, Segrest P (1987) Finite Markov chain analysis of genetic algorithms. In: Proceedings of the 2nd international conference on genetic algorithms, Cambridge, MA, USA, pp 1–8

- Goldberg DE, Voessner S (1999) Optimizing global-local search hybrids. In: Proceedings of the GECCO, Orlando, pp 220–228 (also IlliGAL report no 99001)
- Goldberg DE, Korb B, Deb K (1989) Messy genetic algorithms: motivation, analysis, and first results. *Complex Syst* 3:493–530 (also IlliGAL report no 89003)
- Goldberg DE, Deb K, Clark JH (1992) Genetic algorithms, noise, and the sizing of populations. *Complex Syst* 6:333–362 (also IlliGAL report no 91010)
- Goldberg DE, Deb K, Kargupta H, Harik G (1993a) Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In: Proceedings of the international conference on genetic algorithms, Urbana, pp 56–64 (also IlliGAL report no 93004)
- Goldberg DE, Thierens D, Deb K (1993b) Toward a better understanding of mixing in genetic algorithms. *J Soc Instrum Contr Eng* 32:10–16 (also IlliGAL report no 92009)
- Goldberg DE, Sastry K, Latoza T (2001) On the supply of building blocks. In: Proceedings of the GECCO, San Francisco, pp 336–342 (also IlliGAL report no 2001015)
- Grefenstette JJ, Fitzpatrick JM (1985) Genetic search with approximate function evaluations. In: Proceedings of the international conference on genetic algorithms and their applications, Pittsburgh, pp 112–120
- Harik G (1999) Linkage learning via probabilistic modeling in the ECGA (IlliGAL report no 99010). University of Illinois at Urbana-Champaign
- Harik G, Goldberg DE (1997) Learning linkage. Foundations of genetic algorithms 4, pp 247–262 (also IlliGAL report no 96006)
- Harik G, Lobo F, Goldberg DE (1998) The compact genetic algorithm. In: Proceedings of the IEEE international conference on evolutionary computation, Piscataway, pp 523–528 (also IlliGAL report no 97006)
- Harik G, Cantú-Paz E, Goldberg DE, Miller BL (1999) The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evol Comput* 7:231–253 (also IlliGAL report no 96004)
- Hart WE, Belew RK (1996) Optimization with genetic algorithm hybrids using local search. In: Belew RK, Mitchell M (eds) *Adaptive individuals in evolving populations*. Addison-Wesley, Reading, pp 483–494
- Heckendorn RB, Wright AH (2004) Efficient linkage discovery by limited probing. *Evol Comput* 12:517–545
- Holland JH (1975) *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor
- Ibaraki T (1997) Combinations with other optimixation problems. In: Bäck T, Fogel DB, Michalewicz Z (eds) *Handbook of evolutionary computation*. Institute of Physics Publishing and Oxford University Press, Bristol/New York, pp D3:1–D3:2
- Jin Y (2005) A comprehensive survey of fitness approximation in evolutionary computation. *Soft Comput J* 9:3–12

- Kargupta H (1996) The gene expression messy genetic algorithm. In: Proceedings of the international conference on evolutionary computation, Nagoya, pp 814–819
- Krasnogor N, Smith JE (2005) A tutorial for competent memetic algorithms: models, taxonomy and design issues. *IEEE Trans Evol Comput* 9:474–488
- Krasnogor N, Hart W, Smith JE (eds) (2004) Recent advances in memetic algorithms. Studies in fuzziness and soft computing, vol 166. Springer, Berlin
- Louis SJ, McDonnell J (2004) Learning with case injected genetic algorithms. *IEEE Trans Evol Comput* 8:316–328
- Miller BL, Goldberg DE (1995) Genetic algorithms, tournament selection, and the effects of noise. *Complex Syst* 9:193–212 (also IlliGAL report no 95006)
- Miller BL, Goldberg DE (1996a) Genetic algorithms, selection schemes, and the varying effects of noise. *Evol Comput* 4:113–131 (also IlliGAL report no 95009)
- Miller BL, Goldberg DE (1996b) Optimal sampling for genetic algorithms. Intelligent engineering systems through artificial neural networks, ASME Press, New York 6:291–297
- Moscato P (1989) On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. Technical report C3P 826, California Institute of Technology
- Moscato P (1999) Part 4: Memetic algorithms. In: Corne D, Dorigo M, Glover F (eds) New ideas in optimization. McGraw-Hill, New York, pp 217–294
- Moscato P, Cotta C (2003) A gentle introduction to memetic algorithms. In: Glover F, Kochenberger G (eds) Handbook of metaheuristics, chap 5. Kluwer, Norwell
- Mühlenbein H, Paass G (1996) From recombination of genes to the estimation of distributions I. Binary parameters. *PPSN* 4, pp 178–187
- Mühlenbein H, Schlierkamp-Voosen D (1993) Predictive models for the breeder genetic algorithm: I. Continuous parameter optimization. *Evol Comput* 1:25–49
- Munetomo M, Goldberg DE (1999) Linkage identification by non-monotonicity detection for overlapping functions. *Evol Comput* 7:377–398
- Oliver JM, Smith DJ, Holland JRC (1987) A study of permutation crossover operators on the travelling salesman problem. In: Proceedings of the 2nd international conference on genetic algorithms, Cambridge, pp 224–230
- Paechter B, Cumming A, Luchian H (1995) The use of local search suggestion lists for improving the solution of timetable problems with evolutionary algorithms. In: Fogarty T (ed) Evolutionary computing: AISB workshop 1995. LNCS 993. Springer, Berlin, pp 86–93
- Paechter B, Cumming A, Norman MG, Luchian H (1996) Extensions to a memetic timetabling system. In: Burke EK, Ross P (eds) The practice and theory of automated timetabling I. LNCS 1153. Springer, Berlin, pp 251–265
- Pelikan M (2005) Hierarchical Bayesian optimization algorithm: toward a new generation of evolutionary algorithm. Springer, Berlin
- Pelikan M, Sastry K (2004) Fitness inheritance in the Bayesian optimization algorithm. In: Proceedings of the GECCO 2, Seattle, pp 48–59 (also IlliGAL report no 2004009)

- Pelikan M, Goldberg DE, Cantú-Paz E (2000) Linkage learning, estimation distribution, and Bayesian networks. *Evol Comput* 8:314–341 (also IlliGAL report no 98013)
- Pelikan M, Sastry K, Cantú-Paz E (eds) (2006) Scalable optimization via probabilistic modeling: algorithms to applications. Springer, Berlin
- Rudolph G (2000) Takeover times and probabilities of non-generational selection rules. In: Proceedings of the GECCO, Las Vegas, pp 903–910
- Sakamoto Y, Goldberg DE (1997) Takeover time in a noisy environment. In: Proceedings of the 7th international conference on genetic algorithms, East Lansing, pp 160–165
- Sastry K (2001) Evaluation-relaxation schemes for genetic and evolutionary algorithms. Master's thesis, University of Illinois at Urbana-Champaign (also IlliGAL report no 2002004)
- Sastry K, Goldberg DE (2003) Scalability of selectorecombinative genetic algorithms for problems with tight linkage. In: Proceedings of the GECCO, Chicago, pp 1332–1344 (also IlliGAL report no 2002013)
- Sastry K, Goldberg DE (2004) Let's get ready to rumble: crossover versus mutation head to head. In: Proceedings of the GECCO 2, Seattle, pp 126–137 (also IlliGAL report no 2004005)
- Sastry K, Goldberg DE, Pelikan M (2001) Don't evaluate, inherit. In: Proceedings of the GECCO, San Francisco, pp 551–558 (also IlliGAL report no 2001013)
- Sastry K, Pelikan M, Goldberg DE (2004) Efficiency enhancement of genetic algorithms building-block-wise fitness estimation. In: Proceedings of the IEEE international congress on evolutionary computation. Portland, OR, USA, pp 720–727
- Sastry K, Goldberg DE, Llorà X (2007) Towards billion bit optimization via efficient estimation of distribution algorithms. In: Proceedings of the GECCO, London, pp 577–584 (also IlliGAL report no 2007007)
- Sinha A (2003) Designing efficient genetic and evolutionary hybrids. Master's thesis, University of Illinois at Urbana-Champaign (also IlliGAL report no 2003020)
- Smith R, Dike B, Stegmann S (1995) Fitness inheritance in genetic algorithms. In: Proceedings of the ACM symposium on applied computing. ACM, New York, pp 345–350
- Spears WM, De Jong KA (1994) On the virtues of parameterized uniform crossover. In: Proceedings of the 4th international conference on genetic algorithms, San Diego, pp 230–236
- Syswerda G (1989) Uniform crossover in genetic algorithms. In: Proceedings of the 3rd international conference on genetic algorithms, San Mateo, pp 2–9
- Thierens D (1999) Scalability problems of simple genetic algorithms. *Evol Comput* 7:331–352
- Thierens D, Goldberg DE (1994) Convergence models of genetic algorithm selection schemes. PPSN 3, Springer, Berlin/New York, pp 116–121
- Thierens D, Goldberg DE, Pereira AG (1998) Domino convergence, drift, and the temporal-salience structure of problems. In: Proceedings of the IEEE international congress on evolutionary computation, pp 535–540

- Valenzuela J, Smith AE (2002) A seeded memetic algorithm for large unit commitment problems. *J Heuristics* 8:173–196
- Watson JP, Rana S, Whitley LD, Howe AE (1999) The impact of approximate evaluation on the performance of search algorithms for warehouse scheduling. *J Scheduling* 2:79–98
- Whitley D (1995) Modeling hybrid genetic algorithms. In: Winter G, Périaux J, Galán M, Cuesta P (eds) *Genetic algorithms in engineering and computer science*. Wiley, Chichester, pp 191–201
- Yu T-L (2006) A matrix approach for finding extrema: problems with modularity, hierarchy, and overlap. Doctoral dissertation, University of Illinois at Urbana-Champaign (also IlliGAL report no 2007012)

Chapter 5

Scatter Search

Manuel Laguna

5.1 Introduction

Scatter search (SS) is an evolutionary approach for optimization. It has been applied to problems with continuous and discrete variables and with one or multiple objectives. The success of SS as an optimization technique is well documented in a constantly growing number of journal articles, book chapters ([Glover et al. 2003a, 2003b, 2004; Laguna 2002](#)) and a book ([Laguna and Martí 2003](#)). This chapter contains some of the material that can be found in the aforementioned publications but also contains some new ideas that are based on research that has been performed and reported in recent years. The chapter focuses on the basic SS framework, which is responsible for most of the outcomes reported in the literature. SS consists of five methods:

1. Diversification generation
2. Improvement
3. Reference set update
4. Subset generation
5. Solution combination.

The *diversification generation* method is used to generate a set of diverse solutions that are the basis for initializing the search. The most effective diversification methods are those capable of creating a set of solutions that balances diversification and quality. It has been shown that SS produces better results when the diversification generation method is not purely random and constructs solutions by reference to both a diversification measure and the objective function.

The *improvement* method transforms solutions with the goal of improving quality (typically measured by the objective-function value) or feasibility (typically

M. Laguna (✉)

Leeds School of Business, University of Colorado, Boulder, CO, USA

e-mail: laguna@Colorado.EDU

measured by some degree of constraint violation). The input to the improvement method is a single solution that may or may not be feasible. The output is a solution that may or may not be better (in terms of quality or feasibility) than the original solution. The typical improvement method is a local search with the usual rule of stopping as soon as no improvement is detected in the neighborhood of the current solution. There is the possibility of basing the improvement method on procedures that use a neighborhood search but that are able to escape local optimality. Tabu search, simulated annealing and variable neighborhood search qualify as candidates for such a design. This may seem as an attractive option as a general approach for an improvement method, however, these procedures do not have a natural stopping criterion. The end result is that choices need to be made to control the amount of computer time that is spent improving solutions (by running a metaheuristic-based procedure) versus the time spent outside the improvement method (e.g. combining solutions). In general, local search procedures seem to work well and most SS implementations do not include mechanisms to escape local optimality within the process of improving a solution.

The *reference set update* method refers to the process of building and maintaining a reference set of solutions that are used in the main iterative loop of any SS implementation. While there are several implementation options, this element of SS is fairly independent from the context of the problem. The first goal of the reference update method is to build the initial reference set of solutions from the population of solutions generated with the diversification method. Subsequent calls to the reference update method serve the purpose of maintaining the reference set. The typical design of this method builds the first reference set by blending high-quality solutions and diverse solutions. When choosing a diverse solution, reference needs to be made to a distance metric that typically depends on the solution representation. That is, if the problem context is such that continuous variables are used to represent solutions, then diversification may be measured with Euclidean distances. Other solution representations (e.g. binary variables or permutations) result in different ways of calculating distances and in turn diversification. The updating of the reference set during the SS iterations is customarily done on the basis of solution quality.

The *subset generation* method produces subsets of reference solutions which become the input to the combination method. The typical implementation of this method consists of generating all possible pairs of solutions. The SS framework considers also the generation of larger subsets of reference solutions; however, most SS implementations have been limited to operate on pairs of solutions. Clearly, no context information is needed to implement the subset generation method.

The *solution combination* method uses the output from the subset generation method to create new solutions. New trial solutions are the results of combining, typically two but possibly more, reference solutions. The combination of reference solutions is usually designed to exploit problem context information and solution representation. Linear combinations of solutions represented by continuous variables have been used often since suggested by [Glover \(1998\)](#) in connection with the solution of nonlinear programming problems. Several proposals for combining solutions represented by permutations have also been applied ([Martí et al. 2005](#)).

```

1. Diversification generation and improvement methods
2. while(stopping criteria not satisfied) {
3.   Reference set update method
4.   while(new reference solutions) {
5.     Subset generation method
6.     Combination method
7.     Improvement method
8.     Reference set update method
9.   }
10.  Rebuild reference set
11. }
```

Fig. 5.1 Scatter search framework

The strategy known as path relinking, originally proposed within the tabu search methodology (Glover and Laguna 1997), has also played a relevant role in designing combination methods for SS implementations.

The basic SS framework is outlined in Fig. 5.1. The search starts with the application of the diversification and improvement methods (step 1 in Fig. 5.1). The typical outcome consists of a set of about 100 solutions that is referred to as the population (denoted by P). In most implementations, the diversification generation method is applied first followed by the improvement method. If the application of the improvement method results in the shrinking of the population (due to more than one solution converging to the same local optimum) then the diversification method is applied again until the total number of improved solutions reaches the desired target. Other implementations construct and improve solutions, one by one, until reaching the desired population size.

The main SS loop is shown in lines 2–11 of Fig. 5.1. The input to the first execution of the reference set update method (step 3) is the population of solutions generated in step 1 and the output is a set of solutions known as the reference set (or *RefSet*). Typically, ten solutions are chosen from a population of 100. The first five solutions are chosen to be the best solutions (in terms of the objective-function value) in the population. The other five are chosen to be the most diverse with respect to the solutions in the reference set. If the diverse solutions are chosen sequentially, then the sixth solution is the most diverse with respect to the five best solutions that were chosen first. The seventh solution is the most diverse with respect to the first six and so on until the tenth one is added to the reference set.

The inner while-loop (lines 4–9) is executed as long as at least one reference solution is new in the *RefSet*. A solution is considered new if it has not been subjected to the subset generation (step 5) and combination (step 6) methods. If the reference set contains at least one new solution, the subset generation method builds a list of all the reference solution subsets that will become the input to the combination method. The subset generation method creates new subsets only. A subset is new if it contains at least one new reference solution. This avoids the application of the combination method to the same subset more than once, which is particularly wasteful

when the combination method is completely deterministic. Combination methods that contain random elements may be able to produce new trial solutions even when applied more than once to the same subset of reference solutions. However, this is generally discouraged in favor of introducing new solutions in the reference set by replacing some of the old ones in the rebuilding step (line 10).

The combination method (step 6) is applied to the subsets of reference solutions generated in the previous step. Most combination methods are designed to produce more than one trial solution from the combination of the solutions in a subset. These trial solutions are given to the improvement method (step 7) and the output forms a pool of improved trial solutions that will be considered for admission in the reference set (step 8).

If no new solutions are added to the reference set after the execution of the reference set update method, then the process exits the inner while-loop. The rebuilding step in line 10 is optional. That is, it is possible to implement a SS procedure that terminates the first time that the reference set does not change. However, most implementations extend the search beyond this point by executing a *RefSet* rebuilding step. The rebuilding of the reference set entails the elimination of some current reference solutions and the addition of diverse solutions. In most implementations, all solutions except the best are replaced in this step. The diverse solutions to be added may be either population solutions that have not been used or new solutions constructed with the generation diversification method. Note that only ten solutions out of 100 are used from the population to build the initial reference set and therefore the remaining 90 could be used for rebuilding purposes.

The process (i.e. the main while-loop in lines 2–11) continues as long as the stopping criteria are not satisfied. Possible stopping criteria include number of rebuilding steps or elapsed time. When SS is applied in the context of optimizing expensive black boxes, a limit on the number of calls to the objective-function evaluator (i.e. the black box) may also be used as a criterion for stopping. We now expand our description and provide examples of each of the SS methods.

5.2 Diversification Generation Method

The most effective form of diversification generation is one in which the solutions are not only diverse but their collective quality is better than the outcome of a purely random process. Campos et al. (2001) conducted an experimental study to determine the effectiveness of ten different diversification generation methods. They used two normalized measures to assess the quality and the diversity of the populations generated with each method. The methods varied from purely deterministic to purely random. Most methods were based on semi-greedy constructions, popularized by the first phase of GRASP (Feo and Resende 1995).

Figure 5.2 shows a summary of the results from this experiment. In this figure, Δ_C and Δ_d are values in the range from 0 to 1 that respectively represent the quality and diversity of the population generated by each method. Maximum quality

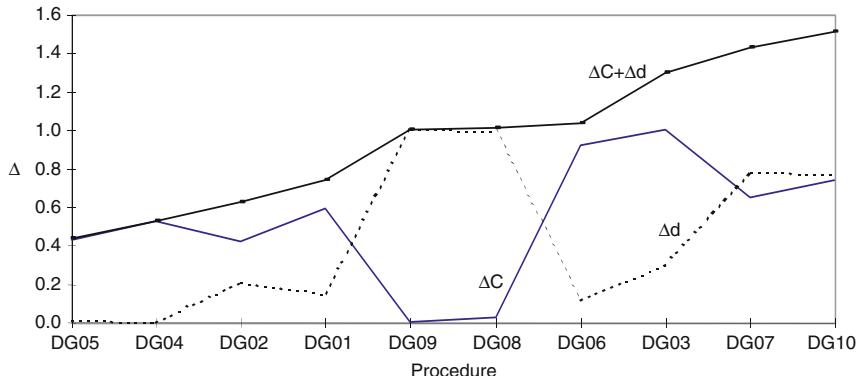


Fig. 5.2 Population diversity versus quality for ten diversification generation methods

or diversity is given by a value of 1 and therefore the maximum score for a diversification method is 2. DG08 is a purely random diversification generation method that achieves the highest level of diversity but also the lowest quality level. DG09 is a purely deterministic method that is designed to achieve maximum diversity without considering the objective-function value during the generation of solutions. The most effective methods, measured by the results achieved at the end of the search, are DG07 and DG10. These methods show an almost perfect balance between diversification and quality of the populations that they generate, as shown by their Δ_C and Δ_d values in Fig. 5.2. DG07 is a semi-greedy construction procedure that uses six different randomized and adaptive functions to select elements to add to a partial solution. DG10 also uses an adaptive function as well as frequency memory to bias the selection of elements during the construction process.

We focus on semi-greedy construction procedures as the basis for developing generation diversification methods. We assume that the problem to be solved consists of minimizing $f(x)$, where x is a solution to the problem. We do not restrict the representation of the problem to any particular form. That is, x may be a vector of real, integer or binary numbers, a permutation, or a set of edges in a graph. We assume, however, that the problem being solved is such that solutions can be constructed by selecting elements one at a time. Selecting an element may refer to choosing a value for a variable, adding an edge to a graph or a task to a sequence. Figure 5.3 outlines the semi-greedy procedure to construct solutions.

Because we assume that the objective of the problem is to minimize a function f , adding elements to a partial solution results in a non-negative change in the objective-function value. Therefore, the increase of the objective function value caused by all the candidate elements is evaluated as the initialization step for the construction process (see step 1 in Fig. 5.3). Following Resende and Ribeiro (2003), let $c(e)$ be the objective-function increase caused by adding element e to the partial solution. Also let c^{\max} and c^{\min} be the maximum and minimum c values, respectively. A purely greedy construction would select, at each iteration, the element e such that $c(e) = c^{\min}$.

1. Evaluate incremental f values for candidate elements
2. **while** (construction is partial) {
3. Build the restricted candidate list (RCL)
4. Randomly choose an element from RCL
5. Add the selected element to the partial solution
6. Re-evaluate incremental f values
7. }

Fig. 5.3 Semi-greedy construction

The procedure in Fig. 5.3 is semi-greedy because the next element to be added is randomly selected from a restricted candidate list (RCL). Candidate elements are assumed to be ordered in such a way that the top candidate has an incremental objective-function value of c^{\min} and the last candidate's incremental objective-function value is c^{\max} . The RCL may be formed in two different ways:

- *Cardinality-based*: the top k candidate elements
- *Value-based*: all elements e such $c(e) \leq c^{\min} + \alpha(c^{\max} - c^{\min})$.

Both of these alternatives require a parameter value that controls the level of randomization. For instance, if $\alpha = 1$, the value-based procedure becomes totally random because all candidate elements have the same probability of being chosen. If $\alpha = 0$, the value-based procedure is totally greedy and results in the same construction every time that the procedure is executed.

A variation of the semi-greedy approach consists of adding a frequency memory structure that is typical to tabu search implementations (Glover and Laguna 1997). The main idea is to keep track of the number of times an element has been assigned a certain value (or occupied a certain position). The frequency counts are then used to modify the c values to discourage (in the probabilistic sense) an element to take on a value that has often been assigned to it in previous constructions. Let $q(e, v)$ be the number of times element e has been given the value v (or has been assigned to position v) in previous constructions, then the modified evaluation that measures the attractiveness of choosing element e is given by

$$c'(e) = c(e) + \beta \left(\frac{q(e, v)}{q^{\max}} \right).$$

The value of β should create a balance between the increase in the objective-function value and the importance given to moving the element away from values (or positions) that it has taken in past constructions. In this equation, q^{\max} refers to the maximum frequency count and is used to normalize the individual frequency counts in order to facilitate the tuning of the β value. Note that we refer to an element taken on a particular value v only as a way of simplifying our discussion. However, this should be reinterpreted in each context. For instance, in nonlinear optimization with continuous variables, v could be interpreted as a particular range of values from which a variable should be moving away due to a frequency count that indicates that

1. Current solution is the trial solution being improved
2. **while** (current solution improves) {
3. Identify the best neighbor of the current solution
4. **if** (neighbor solution is better than current solution)
5. Make neighbor the current solution
6. }

Fig. 5.4 Local search

such a variable has often taken values in the given range. Similarly, v could represent a position in a permutation or a value that indicates whether or not an element (such as an arc in a graph) has been part of the final construction.

When using frequency memory, the selection of elements in each step of the construction does not require the RCL or random components. The first construction results in the pure greedy solution because all frequency counts are zero and $c'(e) = c(e)$. However, as the frequency counts grow, the constructions start diverting from the pure greedy solution. It is also possible, however, to include random components and use the $c'(e)$ to set probabilities proportional to the c' values, where the probability of choosing e increases as $c'(e)$ decreases.

5.3 Improvement Method

As discussed in the introduction to this chapter, most improvement methods within SS implementations consist of local search procedures. Problem context is relevant to the design of an effective local search. The design process starts with the definition of a neighborhood that depends on the moves that may be applied to a solution to transform it into another. Figure 5.4 shows the outline of a local search.

Identifying the *best* neighbor solution may have different meanings according to the problem context and the current solution. For instance, for an unconstrained optimization problem where all the solutions in the solution space generated by a particular solution representation are feasible, the best neighbor of the current solution may be the one that locally minimizes the objective-function value. If the optimization problem contains constraints, then the best neighbor may be either a feasible one that minimizes the objective function, if at least one feasible neighbor exists, or the one that minimizes a measure of infeasibility, if no feasible solutions can be found in the current neighborhood. The local search outlined in Fig. 5.4 results in either a better solution (either a feasible solution with a better objective-function value or an infeasible solution with a smaller infeasibility value) or the unchanged trial solution that was submitted to the improvement method. In either case, the method guarantees finding a locally optimal solution with respect to the defined neighborhood, because it stops only when the current solution cannot be improved (line 2 in Fig. 5.4).

```

1. Current solution is the trial solution being improved
2. while (current solution improves) {
3.   Set  $k$  to 1
4.   while  $k \leq k_{\max}$  {
5.     Identify the best neighbor of the current solution in
6.     neighborhood  $k$ 
7.     if (neighbor is better than current solution)
8.       Make neighbor the current solution
9.       Set  $k$  to 1
10.    else
11.      Make  $k = k + 1$ 
12.   }
13. }

```

Fig. 5.5 Variable neighborhood descent

Variable neighborhood descent (VND) represents an attractive alternative for an improvement method within SS. VND ([Hansen and Mladenović 2003](#)) operates on a finite set of neighborhood structures that are systematically changed until no improvement is obtained. Let k indicate the current neighborhood being explored and k_{\max} the maximum number of neighborhoods. The VND steps are outlined in Fig. 5.5.

VND is the simplest form of the family of variable neighborhood search (VNS) procedures. The basic VNS employs deterministic and stochastic rules for changing the neighborhood that results in a randomized descent method with a first improvement strategy. Extensions transform the basic VNS into a descent–ascent method with first or best improvement strategies.

The appeal of VND as improvement method for SS is its deterministic orientation and its natural termination criterion. As mentioned in the introduction, allowing the improvement method to operate as a metaheuristic capable of escaping local optimality poses the problem of balancing the time spent improving solutions versus the time spent performing SS iterations. In other words, if the improvement method is a metaheuristic in its own right, then there is a risk of diminishing the role of the SS framework to a simple mechanism that provides starting points. [Campos et al. \(2005\)](#), for instance, test the use of a tabu search with a simple short-term memory that terminates after a number of iterations without improvement. This number is set to a relatively small value (e.g. less than or equal to 10) to balance the amount of computational effort spent in the improvement method and the other SS methods.

While the most effective improvement methods are designed for specific problems, it is also possible to develop generic or pseudo-generic local searches. Consider, for instance, the improvement method developed by [Campos et al. \(2005\)](#). In this work, SS is applied to a class of permutation problems. The improvement method is given limited information about the problem instance. In particular, the improvement method is provided only with a flag that indicates whether the objective function tends to be influenced more by the absolute positions of the elements in the permutation (e.g. as in the linear ordering problem) than by their relative

positions (e.g. as in the traveling salesman problem). The actual form of the objective function is not known and therefore the improvement method operates as a black-box optimizer, requiring a context-independent neighborhood search. For permutation problems, a generic local search could consist of either all swaps of two elements or the insertion of one element in every possible position in the permutation. A swap of elements i and j in a permutation with n elements may be represented as follows:

$$(1, \dots, i-1, i, i+1, \dots, j-1, j, j+1, \dots, n) \\ \rightarrow (1, \dots, i-1, j, i+1, \dots, j-1, i, j+1, \dots, n).$$

An insertion of element i before element j produces the following changes in the permutation:

$$(1, \dots, i-1, i, i+1, \dots, j-1, j, j+1, \dots, n) \\ \rightarrow (1, \dots, i-1, i+1, \dots, j-1, i, j, j+1, \dots, n).$$

Note that both neighborhoods are large, containing $O(n^2)$ moves, where n is the number of elements in the permutation. To avoid exploring such a large neighborhood (and in the SS spirit of using strategy instead of relying on randomness) frequency information is used to create a list of promising insertion positions for an element in the permutation. A promising position is one where historically (given by a frequency count) the objective function has improved when the element was placed there. The structure and updating of the frequency count depend on the general class of permutation problem. Thus, for an absolute-position permutation problem a frequency count $\text{freq}(i, p_j)$ may indicate the number of times that the objective function improved when moving (via a swap or an insert) element i to position p_j . Likewise, for a relative-position problem a frequency count $\text{freq}(i, j)$ may indicate the number of times that the objective function improved when moving (via a swap or an insert) element i immediately before element j . When exploring a neighborhood, the moves are limited to those where the frequency counts indicate that there may be merit in placing an element under consideration.

5.4 Reference Set Update

The execution of the diversification generation and improvement methods in line 1 of Fig. 5.1 results in a population P of solutions. The reference set update method is executed in two different parts of the SS procedure. The first time that the method is called, its goal is to produce the initial RefSet , consisting of a mix of b (typically ten) high-quality and diverse solutions drawn from P . The mix of high-quality and diverse solutions could be considered a tunable parameter; however, most implementations populate the initial reference set with half of the solutions chosen by quality and half chosen by diversity.

Choosing solutions by quality is straightforward. From the population P , the best (according to the objective-function value) $b/2$ solutions are chosen. These solutions are added to RefSet and deleted from P . To choose the remaining half, an appropriate measure of distance $d(r, p)$ is needed, where r is a reference solution and p is a solution in P . The distance measure depends on the solution representation, but must satisfy the usual conditions for a metric, that is

$$\begin{aligned} d(r, p) &= 0 \text{ if and only if } r = p \\ d(r, p) &= d(p, r) > 0 \text{ if } r \neq p \\ d(r, q) + d(q, p) &\geq d(r, p) \text{ triangle inequality.} \end{aligned}$$

For instance, the Euclidean distance is commonly used when solutions are represented by continuous variables:

$$d(r, p) = \left(\sum_{i=1}^n (r_i - p_i)^2 \right)^{\frac{1}{2}}.$$

Likewise, the [Hamming \(1950\)](#) distance is appropriate for two strings of equal length. The distance is given by the number of positions for which the corresponding symbols are different. In other words, the distance is the number of changes required to transform one string into the other:

$$d(r, p) = \sum_{i=1}^n v_i \quad v_i = \begin{cases} 0 & r_i = p_i \\ 1 & r_i \neq p_i. \end{cases}$$

This distance has been typically used for problems whose solution representation is given by binary vectors (e.g. the max-cut and knapsack problems) and permutation vectors (e.g. the traveling salesman, quadratic assignment and linear ordering problems).

The distance measure is used to calculate the minimum distance $d_{\min}(p)$ of a population solution and all the reference solutions r :

$$d_{\min}(p) = \min_{r \in \text{RefSet}} d(r, p).$$

Then, the next population solution p to be added to RefSet and deleted from P is the one with the maximum d_{\min} value. That is, we want to choose the population solution p^* that has the maximum minimum distance between itself and all the solutions currently in RefSet :

$$p^* = \left\{ p : \max_{p \in P} d_{\min}(p) \right\}.$$

The process is repeated $b/2$ times to complete the construction of the initial RefSet . Note that after the first calculation of the d_{\min} values, they can be updated as follows. Let p^* be the population solution most recently added to the RefSet . Then, the d_{\min} value for a population solution p is given by

$$d_{\min}(p) = \min(d_{\min}(p), d(p, p^*)).$$

Alternatively, the diverse solutions for the initial reference set may be chosen by solving the maximum diversity problem (MDP). The MDP consists of finding, from a given set of elements and corresponding distances between elements, the most diverse subset of a given size. The diversity of the chosen subset is given by the sum of the distances between every pair of elements. The special version of the MDP that must be solved includes a set of elements that have already been chosen (i.e. the high-quality solutions). Mathematically, the problem may be formulated as follows:

$$\begin{aligned} & \text{Maximize } \sum_{(p,p') \in P: p \neq p'} d(p, p') x_p x_{p'} \\ & \text{Subject to } \sum_{p \in P} x_p = b \\ & \quad x_p = 1 \quad \forall p \in \text{RefSet} \\ & \quad x_p = \{0, 1\} \quad \forall p \in P. \end{aligned}$$

The formulation assumes that a subset of high-quality solutions in P have already been chosen and added to RefSet . The binary variables indicate whether a population solution p is chosen ($x_p = 1$) or not ($x_p = 0$). The second set of constraints in the formulation force the high-quality solutions to be included in the set of b solutions that will become the initial RefSet . This nonlinear programming model has been translated into an integer program for the purpose of solving it as well as for showing that the MDP is NP-hard. [Martí et al. \(2009\)](#) embed the MDP in a SS procedure for the max-cut problem. Instead of solving the MDP exactly, they employ the GRASP_C2 procedure developed by [Duarte and Martí \(2007\)](#). The procedure was modified to account for the high-quality solutions that are chosen before the subset of diverse solutions is added to the RefSet . The procedure is executed for 100 iterations and the most diverse RefSet is chosen to initiate the SS.

The reference set update method is also called in step 8 of Fig. 5.1. This step is performed after a set of one or more trial solutions has been generated by the sequential calls to the subset generation, combination and improvement methods (see steps 5–7 in Fig. 5.1). The most common update consists of the selection of the best (according to the objective function value) solutions from the union of the reference set and the set of trial solutions generated by steps 5–7 in Fig. 5.1. Other updates have been suggested in order to preserve a certain amount of diversity in the RefSet . These advanced updating mechanisms are beyond the scope of this tutorial chapter but the interested reader is referred to [Laguna and Martí \(2003, Chap. 5\)](#) for a detailed description and to [Laguna and Martí \(2005\)](#) for experimental results.

5.5 Subset Generation

This method is in charge of providing the input to the combination method. This input consists of a list of subsets of reference solutions. The most common subset generation consists of creating a list of all pairs (i.e. all 2-subsets) of reference solutions for which at least one of the solutions is new. A reference solution is new if it hasn't been used by the combination method. The first time that this method is called (step 5 in Fig. 5.1), all the reference solutions are new, given that the method

is operating on the initial *RefSet*. Therefore, the execution of the subset generation method results in the list of all 2-subsets of reference solutions, consisting of a total of $(b^2 - b)/2$ pairs. Because the subset generation method is not based on a sample but rather on the universe of all possible pairs, the size of the *RefSet* in SS implementation must be moderate (e.g. less than 20). As mentioned in the introduction, a typical value for b is 10, resulting in 45 pairs the first time that the subset generation method is executed.

When the inner while-loop (steps 5–8 in Fig. 5.1) is executing, the number of new reference solutions at the time that the subset generation method is called depends on the strategies implemented in the reference set update method. Nonetheless, the number of new solutions decreases with the number of iterations within the inner while-loop. Suppose that b is set to 10 and that, after the first iteration of the inner while-loop, six solutions are replaced in the reference set. This means that the reference set that will serve as the input to the subset generation method will consist of four old solutions and six new solutions. Then, the output of the subset generation method will be 39 2-subsets. In general, if the reference set contains n new solutions and m old ones, the number of 2-subsets that the subset generation method produces is given by

$$nm + \frac{n^2 - n}{2}.$$

The SS methodology also considers the generation of subsets with more than two elements for the purpose of combining reference solutions. As described in [Laguna and Martí \(2003\)](#), the procedure uses a strategy to expand pairs into subsets of larger size while controlling the total number of subsets to be generated. In other words, the mechanism does not attempt to create all 2-subsets, then all 3-subsets, and so on until reaching the $b - 1$ -subsets and finally the entire *RefSet*. This approach would not be practical because there are 1,013 subsets in a reference set of size $b = 10$. Even for a smaller reference set, combining all possible subsets would not be effective because many subsets will be very similar. For example, a subset of size four containing solutions 1–4 is almost the same as all the subsets with four solutions for which the first three solutions are solutions 1–3. And even if the combination of subset {1, 2, 3, 5} would generate a different solution than the combination of subset {1, 2, 3, 6}, these new trial solutions would likely reside in the same basin of attraction and therefore converge to the same local optimum after the application of the improvement method. Instead, the approach selects representative subsets of different sizes by creating subset types:

- *Subset Type 1*: all 2-element subsets.
- *Subset Type 2*: 3-element subsets derived from the 2-element subsets by augmenting each 2-element subset to include the best solution not in this subset.
- *Subset Type 3*: 4-element subsets derived from the 3-element subsets by augmenting each 3-element subset to include the best solutions not in this subset.
- *Subset Type 4*: the subsets consisting of the best i elements, for $i = 5$ to b .

[Campos et al. \(2001\)](#) designed an experiment with the goal of assessing the contribution of combining subset types 1–4 in the context of the linear ordering problem. The experiment undertook to identify how often, across a set of benchmark

problems, the best solutions came from combinations of reference solution subsets of various sizes. The experimental results showed that most of the contribution (measured as the percentage of time that the best solutions came from a particular subset type) could be attributed to subset type 1. It was acknowledged, however, that the results could change if the subset types were generated in a different sequence. Nonetheless, the experiments indicate that the basic SS that employs only subsets of type 1 is quite effective and explains why most implementations do not use subset types of higher dimensions.

5.6 Solution Combination

The implementation of this method depends on the solution representation. Problem context can also be exploited by this method; however, it is also possible to create context-independent combination mechanisms. When implementing a context-independent procedure, it is beneficial to employ more than one combination method and track their individual performance. Consider, for instance, the context-independent SS implementations for permutation problems developed by [Campos et al. \(2005\)](#) and [Martí et al. \(2005\)](#). The following combination methods for permutation vectors with n elements were proposed:

1. An implementation of a classical GA crossover operator. The method randomly selects a position k to be the crossing point from the range $[1, n/2]$. The first k elements are copied from one reference point while the remaining elements are randomly selected from both reference points. For each position i ($i = k + 1, \dots, n$) the method randomly selects one reference point and copies the first element that is still not included in the new trial solution.
2. A special case of 1, where the crossing point k is always fixed to one.
3. An implementation of what is known in the GA literature as the *partially matched crossover*. The method randomly chooses two crossover points in one reference solution and copies the partial permutation between them into the new trial solution. The remaining elements are copied from the other reference solution preserving their relative ordering ([Michalewicz 1994](#)).
4. A special case of what the GA literature refers to as a *mutation operator*, and it is applied to a single solution. The method selects two random positions in a chosen reference solution and inverts the partial permutation between them. The inverted partial permutation is copied into the new trial solution. The remaining elements are directly copied from the reference solution preserving their relative order.
5. A combination method that operates on a single reference solution. The method scrambles a sublist of elements randomly selected in the reference solution. The remaining elements are directly copied from the reference solution into the new trial solution.
6. A special case of combination method 5 where the sublist always starts in position 1 and the length is randomly selected in the range $[2, n/2]$.

7. A method that scans (from left to right) both reference permutations, and uses the rule that each reference permutation votes for its first element that is still not included in the combined permutation (referred to as the incipient element). The voting determines the next element to enter the first still-unassigned position of the combined permutation. This is a min–max rule in the sense that if any element of the reference permutation is chosen other than the incipient element, then it would increase the deviation between the reference and the combined permutations. Similarly, if the incipient element were placed later in the combined permutation than its next available position, this deviation would also increase. So the rule attempts to minimize the maximum deviation of the combined solution from the reference solution under consideration, subject to the fact that the other reference solution is also competing to contribute. A bias factor that gives more weight to the vote of the reference permutation with higher quality is also implemented for tie breaking. This rule is used when more than one element receives the same votes. Then the element with highest weighted vote is selected, where the weight of a vote is directly proportional to the objective-function value of the corresponding reference solution. Additional details about this combination method can be found in [Campos et al. \(2001\)](#).
8. A variant of combination method 7. As in the previous method, the two reference solutions vote for their incipient element to be included in the first still-unassigned position of the combined permutation. If both solutions vote for the same element, the element is assigned. But in this case, if the reference solutions vote for different elements and these elements occupy the same position in both reference permutations, then the element from the permutation with the better objective function is chosen. Finally, if the elements are different and occupy different positions, then the one in the lower position is selected.
9. Given two reference solutions r_1 and r_2 , this method probabilistically selects the first element from one of these solutions. The selection is biased by the objective-function value corresponding to r_1 and r_2 . Let e be the last element added to the new trial solution. Then, r_1 votes for the first unassigned element that is positioned after e in the permutation r_1 . Similarly, r_2 votes for the first unassigned element that is positioned after e in r_2 . If both reference solutions vote for the same element, the element is assigned to the next position in the new trial solution. If the elements are different then the selection is proportionally weighted by the objective-function values of r_1 and r_2 .
10. A deterministic version of combination method 9. The first element is chosen from the reference solution with the better objective-function value. Then reference solutions vote for the first unassigned successor of the last element assigned to the new trial solution. If both solutions vote for the same element, then the element is assigned to the new trial solution. Otherwise, the “winner” element is determined with a score, which is updated separately for each reference solution in the combination. The score values attempt to keep the proportion of times that a reference solution “wins” close to its relative importance, where the importance is measured by the value of the objective function. The scores are calculated to minimize the deviation between the “winning rate” and the “relative importance”. For example, if two reference solutions r_1 and r_2 have

objective-function values of $\text{value}(r_1) = 40$ and $\text{value}(r_2) = 60$, then r_1 should contribute with 40 % of the elements in the new trial solution and r_2 with the remaining 60 % in a maximization problem. The scores are updated so that after all the assignments are made the relative contribution from each reference solution approximates the target proportion. More details about this combination method can be found in [Glover \(1994\)](#).

The form of these combination methods show that SS provides great flexibility in terms of generating new trial solutions. That is, the methodology accepts fully deterministic combination methods or those containing random elements that are typically used in genetic algorithms (and labeled crossover or mutation operators). At each execution of step 6 of Fig. 5.1, a combination method is randomly selected. Initially, all combination methods have the same probability of being selected but, as the search progresses and after a specified number of iterations, the probability values increase for those combination methods that have been more successful. Success is measured by the quality of the solutions that the methods are able to produce. Suppose that the *RefSet* is ordered in such a way that the first solution is the best and the b th solution is the worst (according to the objective-function value). Then a *score* for each combination method is kept throughout the search. Initially, all the scores are zero. If a combination method generates a solution that is admitted as the j th reference solution, then a value of $b - j + 1$ is added to the score of this combination method. The probability of selecting a combination method is proportional to its score and therefore combination methods that generate high-quality solutions accumulate higher scores and increase their probability of being chosen.

The same procedure is used by [Gortazar et al. \(2010\)](#) in the context of binary problems. They develop seven combination methods that are probabilistically selected according to the success score. One of these combination methods is based on the path relinking strategy. As described in [Martí et al. \(2006\)](#), the strategy of creating trajectories of moves passing through high-quality solutions was first proposed in connection with tabu search by [Glover \(1998\)](#). The approach was then elaborated in greater detail as a means of integrating intensification and diversification strategies, and given the name path relinking (PR), in the context of tabu search ([Glover and Laguna 1997](#)). PR generally operates by starting from an *initiating* solution, selected from a subset of high-quality solutions, and generating a path in the neighborhood space that leads toward the other solutions in the subset, which is called the *guiding* solution. This is accomplished by selecting moves that introduce attributes contained in the guiding solutions. PR variants consider the use of more than one guiding solution as well as reversing the roles of initiating and guiding solutions resulting in the approach known as *simultaneous relinking*.

Path relinking can be considered an extended form of the combination method. Instead of directly producing a new solution when combining two or more original solutions, PR generates paths between and beyond the selected solutions in the neighborhood space. The character of such paths is easily specified by reference to solution attributes that are added, dropped or otherwise modified by the moves executed.

[Gortazar et al. \(2010\)](#) apply path relinking to combine two binary vectors as follows. The procedure gradually transforms the initiating solution into the guiding solution by changing the value of the variables in the initiating solution with their value in the guiding solution. If the value is the same in both solutions, then no change is made and the procedure moves to the next variable. The procedure examines the variables in lexicographical order and in at most n steps (where n is the number of variables) it reaches the guiding solution. The procedure uses a first-improving strategy, meaning that if during the relinking process it finds an intermediate solution that is better than either the initiating or guiding solutions, then the procedure stops. If no better solution is found, the solution that is most distant from the initiating and guiding solutions is the combined solution resulting from the application of this method. In addition, the procedure reverses the roles of the initiating and guiding solutions and chooses the best solution found during both processes to be the outcome of the combination method.

5.7 Multiobjective Optimization

Evolutionary procedures have enjoyed a fair amount of success in multiobjective optimization, as documented by [Coello et al. \(2002\)](#). SS is starting to be applied to this area. [Molina et al. \(2007\)](#) developed a tabu/SS hybrid for approximating the efficient frontier of multiobjective nonlinear problems with continuous variables. The method consists of two major phases:

1. Generation of an initial set of efficient points through various tabu searches;
2. Combination of solutions and updating of the efficient frontier approximation via a SS.

As we have discussed above for single-objective problems, the *RefSet* contains a mixture of high-quality and diverse solutions, where quality is measured with reference to the single objective function and diversity is measured by an appropriate metric in the solution space. The role of the *RefSet* must be modified to deal with the special characteristics of multiobjective optimization. In particular, solution quality is measured considering multiple objective functions and solution diversity is measured in the objective-function space. The notion of diversity is related to the ability of finding solutions that cover the efficient frontier. Hence, measuring diversity in the objective-function space is an effective means to produce the desired results.

One of the key search mechanisms is the combination of solutions that are currently considered efficient and therefore belong to the best approximation of the efficient frontier. The *RefSet* is a subset of efficient solutions of a size larger than the number of objective functions in the problem and is initially constructed as follows:

- Select the best solution in the current approximation of the efficient frontier for each of the objective functions and add them to *RefSet*. (Note that it is possible, but unlikely, to select fewer solutions than the number of objective function if a solution happens to be best for more than one objective function.)

- Select additional solutions (up to a total of b) from the approximation of the efficient frontier in order to maximize the distance between them and those solutions already in the *RefSet*, where distance is measured in the objective-function space.

The construction of the initial *RefSet* reveals that in the multiobjective implementation of SS, the population P is the best approximation of the efficient frontier. This is an expanded role for P (when compared to single-objective optimization) because it not only supports the diversification of the *RefSet* but also acts as a repository of efficient solutions. A list of solutions that have been selected as reference points is kept to prevent the selection of those solutions in future iterations. Therefore, every solution that is added to the *RefSet* is also added to a *TabuRefSet*. The size of the *TabuRefSet* increases as the search progresses because this memory function is an explicit record of past reference solutions. The motivation for creating and maintaining the *TabuRefSet* is that the final approximation of the efficient frontier must have adequate density. To achieve this, the procedure must encourage a uniform generation of points in the efficient frontier and avoid gaps that may be the result of generating too many points in one region while neglecting other regions.

Combination methods operate in the solution space and therefore are similar to their single-objective counterpart. Improvement methods, however, must consider more than one objective and are typically based on concepts from compromise programming. A global criterion is used to guide the search and the goal is to minimize a function that measures the distance to an ideal point. Given that the ideal point consists of the optimal (or best known) values of the individual objective functions, compromise programming assumes that it is logical for the decision maker to prefer a point that is closer to the ideal point over one that is farther away.

The same framework has been adapted for multiobjective combinatorial optimization. In particular, [Caballero et al. \(2011\)](#) tackled partitioning problems for cluster analysis that requires the simultaneous optimization of more than one objective function. They considered two main classes of multiobjective partitioning problems: (1) partitioning of objects using one partitioning criterion but multiple dissimilarity matrices and (2) partitioning of objects using one dissimilarity matrix but more than one partitioning criteria. The adaptation consists of formulating an appropriate solution representation and employing the representation to develop combination and improvement methods. Other methods (particularly those that operate in the objective-function space) were applied with minor or no changes.

5.8 Tricks of the Trade

Tricks associated with implementing SS are extensively addressed in three tutorial Chaps. (2–4) in [Laguna and Martí \(2003\)](#). We summarize a few here:

1. Effective diversification generation methods employ controlled randomization and frequency-based memory to generate a set of diverse solutions. The use of frequency-based memory is typical in implementations of tabu search.

2. The diversification generation method is used at the beginning of the search to generate a set of diverse solutions. In most SS applications, the size of this population of solutions is generally set at $\max(100, 5 * b)$, where b is the size of the reference set.
3. While some solution generation is done without considering the objective function, in other words, some diversification generation methods focus on diversification and not on the quality of the resulting solutions, it is generally more effective to design a procedure that balances diversification and solution quality (such as those based on GRASP constructions).
4. Improvement methods must be capable of handling starting solutions that are either feasible or infeasible. When encountering an infeasible solution, an improvement method should search for a feasible solution first and then launch a search for improvement.
5. Whenever possible, the improvement method should be a known local search procedure. For example, when using SS for nonlinear optimization, the improvement method could be the well-known Nelder and Mead simplex procedure.
6. The reference set update method to try first is the so-called *static update*. Trial solutions that are constructed as combination of reference solutions are placed in a solution *pool*. After the application of both the combination method and the improvement method, the *pool* is full and the reference set is updated. The new reference set consists of the best b solutions (where b is the size of the reference set) from the solutions in the current reference set and the solutions in the *pool*, i.e. the updated reference set contains the best b solutions in the union of the reference set and the *pool*.
7. The subset generation method should be limited to generating all solution pairs first before trying more sophisticated designs to include more than two solutions in each subset. Even when the combination method includes stochastic elements, it is generally more effective to create only solution subsets that include at least one new reference solution (i.e. a solution that has been added to the reference set in the previous iteration).
8. The number of solutions created from the combination of two or more reference solutions should depend on the quality of the solutions being combined. For instance, the maximum number of solutions generated by the combination method should happen when the two best solutions in the reference set are being combined. Likewise, only one solution should be the result of combining the two reference solutions with the worst objective-function value.
9. If SS is implemented to exploit problem context, the combination method should take full advantage of the information associated with the problem being solved. For instance, a combination method for the linear ordering problem should take into consideration that the largest contribution to the objective-function value comes from the items that are placed in the first positions of the permutation.
10. Employing multiple combination methods has been shown to be an effective strategy. The combination methods are applied probabilistically, starting with an equal probability of selecting any of the available methods. The probability changes, with the success of each method, where success is typically defined

as creating trial solutions that are admitted to the reference set because of their quality. As the search progresses, the more effective (i.e. successful) methods are chosen more often. This strategy is particularly useful when SS is used as a black-box optimizer, where no context information is used to create combination methods that are known to be effective for certain classes of problems.

5.9 Conclusions

The goal of this chapter is to introduce the SS framework at a level that would make it possible for the reader to implement a basic but robust procedure. A number of extensions are possible and some of them have already been explored and reported in the literature. It is not possible within the limited scope of this chapter to detail completely many of the aspects of SS that warrant further investigation. Additional implementation considerations, including those associated with intensification and diversification processes, and the design of accompanying methods to improve solutions produced by combination strategies are found in several of the references listed below.

5.10 Promising Areas for Future Research

SS is at the core of OptQuest, a popular commercial software package for global optimization. OptQuest is implemented as a black-box optimizer that focuses on searching for high-quality solutions to problems with expensive objective-function evaluations (such as those characterized by a computer simulation). SS has shown merit in applications where the optimization horizon (represented by a number of objective function evaluations) is severely limited. This is a promising SS research avenue, considering the importance of optimizing black boxes in general and simulations in particular.

One way of creating effective heuristic search procedures with limited objective-function evaluations is through the use of rough set theory. Rough sets have been successfully adapted as a mechanism to combine solutions and to perform local optimization in the context of multiobjective optimization. They have been hybridized with search methods such as differential evolution ([Hernández-Díaz et al. 2006](#)), particle swarm optimization ([Santana-Quintero et al. 2006](#)) and metamodels based on radial basis functions ([Santana-Quintero et al. 2007](#)). Multiobjective optimization results are encouraging, showing the effectiveness of the hybrid implementations in obtaining dense approximations of the Pareto fronts. Additional details on the use of rough sets in multiobjective optimization can be found in [Hernández-Díaz et al. \(2008\)](#). These studies point to the merit of exploring the addition of mechanisms based on rough sets to SS implementations for multiobjective optimization.

While the merging of SS and rough sets for multiobjective optimization remains to be explored, [Laguna et al. \(2010\)](#) employed a rough-set theory procedure as a combination method within a SS for nonlinear optimization with a single multimodal objective function. The goal of this work was to find high-quality solutions to difficult multimodal functions while limiting the number of objective-function evaluations. The search process was divided into two stages, starting from a coarsely discretized solution space and ending at the original solution space represented by continuous variables. The authors adapted the standard SS methodology to provide the data that rough-set theory needs to identify promising areas in the solution space. The implementation departed from the traditional SS framework in that a sampling procedure was used to create subsets of solutions to which the rough-set combination method was applied. Similar mechanisms are worth exploring in order to create innovative ways of combining solutions for both single-objective and multiobjective optimization problems that are tackled with SS.

The experiments with 92 problems instances from the literature presented by [Laguna et al. \(2010\)](#) showed the merit of the SS/rough-set combination when compared to an existing method based on particle swarm optimization. A potential extension to this work is in the area of simulation optimization. As mentioned above, the evaluation of the objective function in this context typically requires a significant amount of computational effort and is noisy. The use of rough sets to reduce the number of evaluations required to identify promising regions in the solution space may be of great advantage, particularly in situations where the number of objective-function evaluations (i.e. calls to the simulation module) is limited to no more than 100 times the number of decision variables in the problem. Coupling rough sets, metamodels and ranking and selection may result in a highly effective approach for dealing with expensive and noisy objective functions. Also, given that rough sets have been shown to be effective in multiobjective optimization (see [Hernández-Díaz et al. 2006, 2008](#)), another promising research avenue is the development of a SS/rough-sets for multiobjective simulation optimization, an area that has generated a fair amount of interest in engineering and science ([Lee et al. 1996; Mebarki and Castagna 2000; Yang and Chou 2005; Pasandideh and Niaki 2006; Rosen et al. 2007, 2008; Teng et al. 2007; Zhang 2008; Willis and Jones 2008](#)).

Sources of Additional Information

The best source of information to get started with SS is the book by [Laguna and Martí \(2003\)](#). This book contains three tutorials, including searches in continuous spaces (nonlinear unconstrained optimization), constrained binary spaces (knapsack problems) and combinatorial optimization (linear ordering problem). Advanced strategies are also addressed and computer code is provided that can be easily modified to create basic and even advanced implementation of SS for other optimization problems.

Several SS tutorials have appeared in the literature and can be found at <http://leeds-faculty.colorado.edu/laguna>. Some of these chapters and tutorials include implementations that use path relinking as a mechanism to combine solutions within a SS framework.

The Opticom Website (<http://heur.uv.es/opticom/>) contains the description of several optimization procedures based on metaheuristics, including SS. Implementations of SS as black-box optimizer and for particular problem classes can be found in this website.

References

- Caballero R, Laguna M, Martí R, Molina J (2011) Scatter tabu search for multiobjective clustering problems. *J Oper Res Soc* 62:2034–2046
- Campos V, Glover F, Laguna M, Martí R (2001) An experimental evaluation of a scatter search for the linear ordering problem. *J Glob Optim* 21:397–414
- Campos V, Laguna M, Martí R (2005) Context-independent scatter and tabu search for permutation problems. *INFORMS J Comput* 17:111–122
- Coello CA, Van Veldhuizen DA, Lamont GB (2002) Evolutionary algorithms for solving multi-objective problems. Kluwer/Plenum, New York
- Duarte A, Martí R (2007) Tabu search and GRASP for the maximum diversity problem. *Eur J Oper Res* 178:71–84
- Feo TA, Resende MGC (1995) Greedy randomized adaptive search procedures. *J Glob Optim* 6:109–133
- Glover F (1994) Tabu search for nonlinear and parametric optimization with links to genetic algorithms. *Discret Appl Math* 49:231–255
- Glover F (1998) A template for scatter search and path relinking. In: Hao JK, Lutton E, Ronald E, Schoenauer M, Snyers D (eds) Artificial evolution. Lecture notes in computer science, vol. 1363. Springer, Berlin, pp 1–5
- Glover F, Laguna M (1997) Tabu search. Kluwer, Boston
- Glover F, Laguna M, Martí R (2003a) Scatter search. In: Ghosh A, Tsutsui S (eds) Advances in evolutionary computation: theory and applications. Springer, New York, pp 519–537
- Glover F, Laguna M, Martí R (2003b) Scatter search and path relinking: advances and applications. In: Glover F, Kochenberger G (eds) Handbook of metaheuristics. Kluwer, Boston, pp 1–35
- Glover F, Laguna M, Martí R (2004) New ideas and applications of scatter search and path relinking. In: Onwubolu GC, Babu BV (eds) New optimization techniques in engineering. Springer, Berlin, pp 367–383
- Gortazar F, Duarte A, Laguna M, Martí R (2010) Context-independent scatter search for binary problems. *Comput Oper Res* 37:1977–1986
- Hamming RW (1950) Error detecting and error correcting codes. *Bell Syst Tech J* 26:147–160

- Hansen P, Mladenović N (2003) Variable neighborhood search. In: Glover F, Kochenberger G (eds) *Handbook of metaheuristics*. Kluwer, Boston, pp 145–184
- Hernández-Díaz AG, Santana-Quintero LV, Coello Coello CA, Caballero R, Molina J (2006) A new proposal for multiobjective optimization using differential evolution and rough set theory. In: *Proceedings of the genetic and evolutionary computation conference*. ACM, New York, Seattle, Washington, pp 675–683
- Hernández-Díaz AG, Santana-Quintero LV, Coello Coello CA, Caballero R, Molina J (2008) Improving multi-objective evolutionary algorithms by using rough sets. In: Ligeza A, Reich S, Schaefer R, Cotta C (eds) *Knowledge-driven computing: knowledge engineering and intelligent computations. Studies in computational intelligence*, vol 102. Springer-Verlag Berlin Heidelberg, pp 81–98
- Laguna M (2002) Scatter search. In: Pardalos PM, Resende MGC (eds) *Handbook of applied optimization*. Oxford University Press, New York, pp 183–193
- Laguna M, Martí R (2003) Scatter search: methodology and implementations in C. Kluwer, Boston
- Laguna M, Martí R (2005) Experimental testing of advanced scatter search designs for global optimization of multimodal functions. *J Glob Optim* 33:235–255
- Laguna M, Molina J, Pérez F, Caballero R, Hernández-Díaz A (2010) The challenge of optimizing expensive black boxes: a scatter search/rough set theory approach. *J Oper Res Soc* 61:53–67
- Lee Y-H, Shin H-M, Yang B-H (1996) An approach for multiple criteria simulation optimization with application to turning operation. *Comput Indust Eng* 30:375–386
- Martí R, Laguna M, Campos V (2005) Scatter search vs. genetic algorithms: an experimental evaluation with permutation problems. In: Rego C, Alidaee B (eds) *Metaheuristic optimization via adaptive memory and evolution: tabu search and scatter search*. Kluwer, Norwell, pp 263–282
- Martí R, Laguna M, Glover F (2006) Principles of scatter search. *Eur J Oper Res* 169:359–372
- Martí R, Duarte A, Laguna M (2009) Advanced scatter search for the max-cut problem. *INFORMS J Comput* 21:26–38
- Mebarki N, Castagna P (2000) An approach based on Hotelling's test for multicriteria stochastic simulation-optimization. *Simul Pract Theor* 8:341–355
- Michalewicz Z (1994) *Genetic algorithms + data structures = evolution programs*. Springer, Berlin
- Molina J, Laguna M, Martí R, Caballero R (2007) SSPMO: a scatter search procedure for non-linear multiobjective optimization. *INFORMS J Comput* 19:91–100
- Pasandideh SHR, Niaki STA (2006) Multi-response simulation optimization using genetic algorithm within desirability function framework. *Appl Math Comput* 175:366–382
- Resende MGC, Ribeiro CC (2003) Greedy randomized adaptive search procedures. In: Glover F, Kochenberger G (eds) *Handbook of Metaheuristics*. Kluwer Academic Publishers, pp 219–249

- Rosen SL, Harmonosky CH, Traband MT (2007) A simulation optimization method that considers uncertainty and multiple performance measures. *Eur J Oper Res* 181:315–330
- Rosen SL, Harmonosky CH, Traband MT (2008) Optimization of systems with multiple performance measures via simulation: survey and recommendations. *Comput Ind Eng* 54:327–339
- Santana-Quintero LV, Ramírez-Santiago N, Coello Coello CA, Molina J, Hernández-Díaz AG (2006) In: Runarsson ThP, Beyer H-G, Burke E, Merelo-Guervós JJ, Whitley LD, Yao X (eds) Parallel problem solving from nature – PPSN IX. Lecture notes in computer science, vol 4193. Springer, Berlin/New York, pp. 483–492
- Santana-Quintero LV, Serrano-Hernández VA, Coello Coello CA, Hernández-Díaz AG, Molina J (2007) Use of radial basis functions and rough sets for evolutionary multiobjective optimization. In: IEEE symposium on computational intelligence in multicriteria decision making (MCDM 07), Seattle, Washington, pp 107–114
- Teng S, Lee JH, Chew EP (2007) Multi-objective ordinal optimization for simulation optimization problems. *Automatica* 43:1884–1895
- Willis KO, Jones DF (2008) Multi-objective simulation optimization through search heuristics and relational database analysis. *Decis Support Syst* 46:277–286
- Yang T, Chou P (2005) Solving a multiresponse simulation-optimization problem with discrete variables using a multi-attribute decision-making method. *Math Comput Simul* 68:9–21
- Zhang H (2008) Multi-objective simulation optimization for earthmoving operations. *Automat Constr* 18:79–86

Chapter 6

Genetic Programming

Riccardo Poli and John Koza

6.1 Introduction

The goal of getting computers to automatically solve problems is central to artificial intelligence, machine learning, and the broad area encompassed by what Turing called *machine intelligence* (Turing 1948, 1950).

In his 1983 talk entitled *AI: Where It Has Been and Where It Is Going*, machine learning pioneer Arthur Samuel stated the main goal of the fields of machine learning and artificial intelligence:

[T]he aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.

Genetic programming (GP) is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done. GP is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, GP iteratively transforms a population of computer programs into a new generation of programs by applying analogues of naturally occurring genetic operations. This process is illustrated in Fig. 6.1.

The genetic operations include crossover (sexual recombination), mutation and reproduction. It may also include other analogues of natural operations such as gene duplication, gene deletion and developmental processes which transform an embryo into a fully developed structure. GP is an extension of the genetic algorithm

R. Poli (✉)

School of Computer Science and Electronic Engineering, University of Essex,
Colchester, Essex, UK
e-mail: rpoli@essex.ac.uk

J. Koza

Stanford University, Stanford, CA, USA
e-mail: john@johnkoza.com

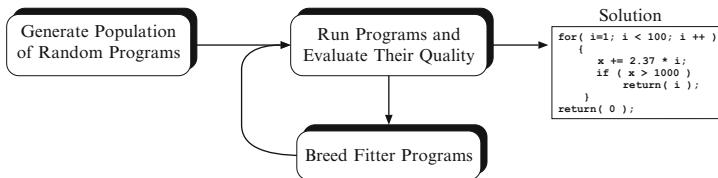


Fig. 6.1 Main loop of genetic programming

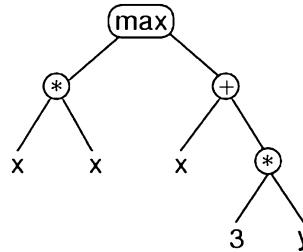


Fig. 6.2 Basic tree-like program representation used in genetic programming

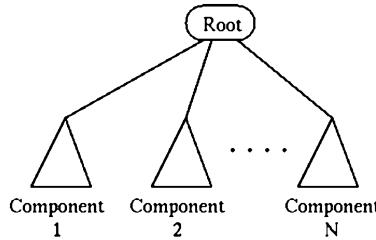


Fig. 6.3 Multi-tree program representation

(Holland 1975) in which the *structures* in the population are not fixed-length character strings that encode candidate solutions to a problem, but *programs* that, when executed, *are* the candidate solutions to the problem.

Programs are expressed in GP as *syntax trees* rather than as lines of code. For example, the simple expression $\max(x*x, x+3*y)$ is represented as shown in Fig. 6.2. The tree includes *nodes* (which we will also call *points*) and *links*. The nodes indicate the instructions to execute. The links indicate the arguments for each instruction. In the following the internal nodes in a tree will be called *functions*, while the tree's leaves will be called *terminals*.

In more advanced forms of GP, programs can be composed of multiple components (e.g. subroutines). Often in this case the representation used in GP is a set of trees (one for each component) grouped together under a special node called *root*, as illustrated in Fig. 6.3. We will call these (sub)trees *branches*. The number and type of the branches in a program, together with certain other features of the structure of the branches, form the *architecture* of the program.

GP trees and their corresponding expressions can equivalently be represented in *prefix notation* (e.g. as Lisp S-expressions). In prefix notation, functions always

precede their arguments. For example, $\max(x*x, x+3*y)$ becomes $(\max (* x x) (+ x (* 3 y)))$. In this notation, it is easy to see the correspondence between expressions and their syntax trees. Simple recursive procedures can convert prefix-notation expressions into infix-notation expressions and vice versa. Therefore, in the following, we will use trees and their corresponding prefix-notation expressions interchangeably.

6.2 Preparatory Steps of Genetic Programming

Genetic programming starts from a high-level statement of the requirements of a problem and attempts to produce a computer program that solves the problem.

The human user communicates the high-level statement of the problem to the GP algorithm by performing certain well-defined preparatory steps.

The five major preparatory steps for the basic version of genetic programming require the human user to specify:

1. The set of terminals (e.g. the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program,
2. The set of primitive functions for each branch of the to-be-evolved program,
3. The fitness measure (for explicitly or implicitly measuring the quality of individuals in the population),
4. Certain parameters for controlling the run, and
5. The termination criterion and method for designating the result of the run.

The first two preparatory steps specify the ingredients that are available to create the computer programs. A run of GP is a competitive search among a diverse population of programs composed of the available functions and terminals.

The identification of the function set and terminal set for a particular problem (or category of problems) is usually a straightforward process. For some problems, the function set may consist of merely the arithmetic functions of addition, subtraction, multiplication and division as well as a conditional branching operator. The terminal set may consist of the program's external inputs (independent variables) and numerical constants.

For many other problems, the ingredients include specialized functions and terminals. For example, if the goal is to get GP to automatically program a robot to mop the entire floor of an obstacle-laden room, the human user must tell GP what the robot is capable of doing. For example, the robot may be capable of executing functions such as moving, turning and swishing the mop.

If the goal is the automatic creation of a controller, the function set may consist of integrators, differentiators, leads, lags, gains, adders, subtractors and the like, and the terminal set may consist of signals such as the reference signal and plant output.

If the goal is the automatic synthesis of an analog electrical circuit, the function set may enable GP to construct circuits from components such as transistors, capacitors and resistors. Once the human user has identified the primitive ingredients for a problem of circuit synthesis, the same function set can be used to automatically

synthesize an amplifier, computational circuit, active filter, voltage reference circuit, or any other circuit composed of these ingredients.

The third preparatory step concerns the fitness measure for the problem. The fitness measure specifies what needs to be done. The fitness measure is the primary mechanism for communicating the high-level statement of the problem's requirements to the GP system. For example, if the goal is to get GP to automatically synthesize an amplifier, the fitness function is the mechanism for telling GP to synthesize a circuit that amplifies an incoming signal (as opposed to, say, a circuit that suppresses the low frequencies of an incoming signal or that computes the square root of the incoming signal). The first two preparatory steps define the search space whereas the fitness measure implicitly specifies the search's desired goal.

The fourth and fifth preparatory steps are administrative. The fourth preparatory step entails specifying the control parameters for the run. The most important control parameter is the population size. Other control parameters include the probabilities of performing the genetic operations, the maximum size for programs, and other details of the run.

The fifth preparatory step consists of specifying the termination criterion and the method of designating the result of the run. The termination criterion may include a maximum number of generations to be run as well as a problem-specific success predicate. The single best-so-far individual is then harvested and designated as the result of the run.

6.3 Executional Steps of GP

After the user has performed the preparatory steps for a problem, the run of genetic programming can be launched. Once the run is launched, a series of well-defined, problem-independent steps is executed.

GP typically starts with a population of randomly generated computer programs composed of the available programmatic ingredients (as provided by the human user in the first and second preparatory steps).

GP iteratively transforms a population of computer programs into a new generation of the population by applying analogues of naturally occurring genetic operations. These operations are applied to individual(s) selected from the population. The individuals are probabilistically selected to participate in the genetic operations based on their fitness (as measured by the fitness measure provided by the human user in the third preparatory step). The iterative transformation of the population is executed inside the main generational loop of the run of GP.

The executional steps of GP are as follows:

1. Randomly create an initial population (generation 0) of individual computer programs composed of the available functions and terminals.
2. Iteratively perform the following sub-steps (called a *generation*) on the population until the termination criterion is satisfied:

- (a) Execute each program in the population and ascertain its fitness (explicitly or implicitly) using the problem's fitness measure.
 - (b) Select one or two individual program(s) from the population with a probability based on fitness (with reselection allowed) to participate in the genetic operations in (c).
 - (c) Create new individual program(s) for the population by applying the following genetic operations with specified probabilities:
 - *Reproduction*: Copy the selected individual program into the new population.
 - *Crossover*: Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.
 - *Mutation*: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.
 - *Architecture-altering operations*: If this feature is enabled, choose an architecture-altering operation from the available repertoire of such operations and create one new offspring program for the new population by applying the chosen architecture-altering operation to one selected program.
3. After the termination criterion is satisfied, the single best program in the population produced during the run (the best-so-far individual) is harvested and designated as the result of the run. If the run is successful, the result will be a solution (or approximate solution) to the problem.

Figure 6.4 is a flowchart of GP showing the genetic operations of crossover, reproduction, and mutation as well as the architecture-altering operations. This flowchart shows a two-offspring version of the crossover operation.

The preparatory steps specify what the user must provide in advance to the GP system. Once the run is launched, the executional steps as shown in the flowchart (Fig. 6.4) are executed. GP is problem independent in the sense that the flowchart specifying the basic sequence of executional steps is not modified for each new run or each new problem.

There is usually no discretionary human intervention or interaction during a run of GP (although a human user may exercise judgment as to whether to terminate a run).

GP starts with an initial population of computer programs composed of functions and terminals appropriate to the problem. The individual programs in the initial population are typically generated by recursively generating a rooted point-labeled program tree composed of random choices of the primitive functions and terminals (provided by the user as part of the first and second preparatory steps). The initial individuals are usually generated subject to a pre-established maximum size (specified by the user as a minor parameter as part of the fourth preparatory step). For example, in the “Full” initialization method nodes are taken from the function set until a maximum tree depth is reached. Beyond that depth only terminals can be chosen. Figure 6.5 shows several snapshots of this process. A variant of this, the “Grow” initialization method, allows the selection of nodes from the whole primitive set until the depth limit is reached. Thereafter, it behaves like the “Full” method.

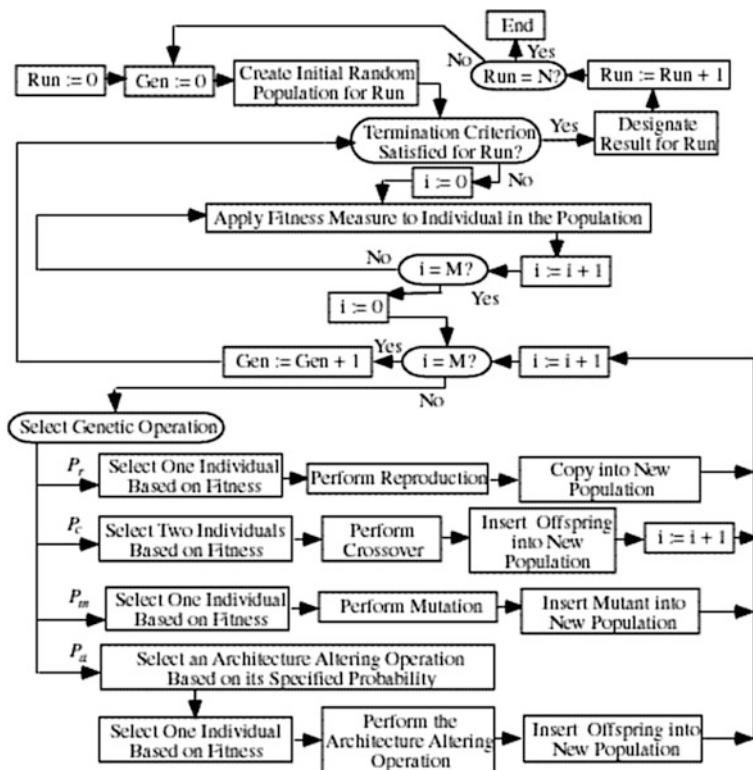


Fig. 6.4 Flowchart of genetic programming

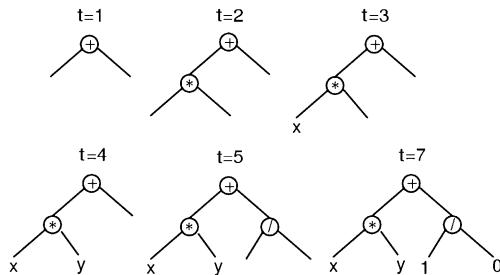


Fig. 6.5 Creation of a seven-point tree using the "Full" initialization method (t = time)

In general, after the initialization phase, the programs in the population are of different size (number of functions and terminals) and of different shape (the particular graphical arrangement of functions and terminals in the program tree).

Each individual program in the population is either measured or compared in terms of how well it performs the task at hand (using the fitness measure provided in the third preparatory step). For many problems, this measurement yields a single explicit numerical value, called fitness. Normally, fitness evalua-

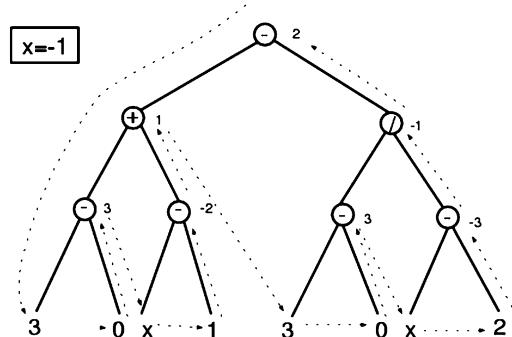


Fig. 6.6 Example interpretation of a syntax tree (the terminal x is a variable holding the value -1)

```

procedure: eval
arguments:
expr /* An expression in prefix notation */
results:
value /* A number */
begin
  if expr is a list /* Non-terminal */
    proc = expr(1)
    value = proc(eval(expr(2)),eval(expr(3)),...)
  else /* Terminal */
    if expr is a variable or a constant then
      value = expr
    else /* 0-arity function */
      value = expr()
    endif
  endif
end

```

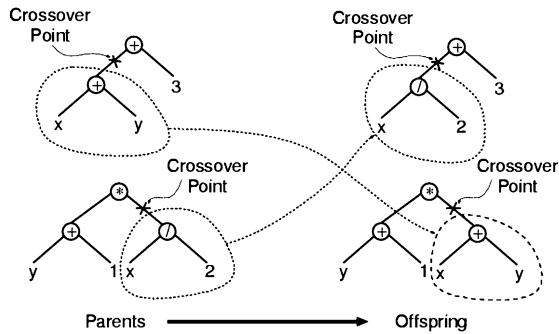
Fig. 6.7 Typical interpreter for genetic programming

tion requires executing the programs in the population, often multiple times, *within* the GP system. A variety of execution strategies exist. The most common are virtual-machine-code compilation and interpretation. We will look at the latter.

Interpreting a program tree means executing the nodes in the tree in an order that guarantees that nodes are not executed before the value of their arguments (if any) is known. This is usually done by traversing the tree in a recursive way starting from the root node, and postponing the evaluation of each node until the value of its children (arguments) is known. This process is illustrated in Fig. 6.6, where the numbers to the right of internal nodes represent the results of evaluating the subtrees rooted at such nodes. In this example, the independent variable x evaluates to -1 . Figure 6.7 gives a pseudo-code implementation of the interpretation procedure. The code assumes that programs are represented as prefix-notation expressions and that such expressions can be treated as lists of components (where a construct like $expr(i)$ can be used to read or set component i of expression $expr$).

Irrespective of the execution strategy adopted, the fitness of a program may be measured in many different ways, including, for example, in terms of the amount of error between its output and the desired output, the amount of time (fuel, money, etc.) required to bring a system to a desired target state, the accuracy of the program

Fig. 6.8 Example of two-child crossover between syntax trees



in recognizing patterns or classifying objects into classes, the payoff that a game-playing program produces, or the compliance of a complex structure (such as an antenna, circuit, or controller) with user-specified design criteria. The execution of the program sometimes returns one or more explicit values. Alternatively, the execution of a program may consist only of side effects on the state of a world (e.g. a robot's actions). Alternatively, the execution of a program may yield both return values and side effects.

The fitness measure is, for many practical problems, multi-objective in the sense that it combines two or more different elements. In practice, the different elements of the fitness measure are in competition with one another to some degree.

For many problems, each program in the population is executed over a representative sample of different *fitness cases*. These fitness cases may represent different values of the program's input(s), different initial conditions of a system, or different environments. Sometimes the fitness cases are constructed probabilistically.

The creation of the initial random population is, in effect, a blind random search of the search space of the problem. It provides a baseline for judging future search efforts. Typically, the individual programs in generation 0 all have exceedingly poor fitness. Nonetheless, some individuals in the population are (usually) fitter than others. The differences in fitness are then exploited by genetic programming. GP applies Darwinian selection and the genetic operations to create a new population of offspring programs from the current population.

The genetic operations include crossover (sexual recombination), mutation, reproduction, and the architecture-altering operations (when they are enabled). Given copies of two parent trees, typically, *crossover* involves randomly selecting a crossover point in each parent tree and swapping the subtrees rooted at the crossover points, as exemplified in Fig. 6.8. Often crossover points are not selected with uniform probability. A frequent strategy is, for example, to select internal nodes (functions) 90 % of the time, and any node for the remaining 10 % of the times. Traditional *mutation* consists of randomly selecting a mutation point in a tree and substituting the subtree rooted there with a randomly generated subtree, as illustrated in Fig. 6.9. *Reproduction* involves simply copying certain individuals into the new population. Architecture-altering operations are discussed later in this chapter.

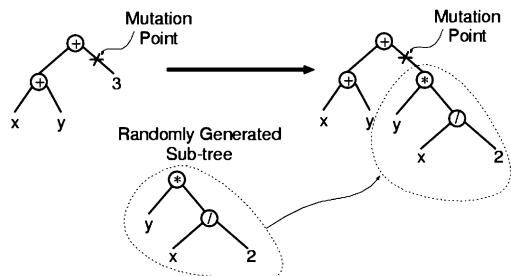


Fig. 6.9 Example of subtree mutation

The genetic operations described above are applied to individual(s) that are probabilistically selected from the population based on fitness. In this probabilistic selection process, better individuals are favored over inferior individuals. However, the best individual in the population is not necessarily selected and the worst individual in the population is not necessarily passed over.

After the genetic operations are performed on the current population, the population of offspring (i.e. the new generation) replaces the current population (i.e. the now-old generation). This iterative process of measuring fitness and performing the genetic operations is repeated over many generations.

The run of GP terminates when the termination criterion (as provided by the fifth preparatory step) is satisfied. The outcome of the run is specified by the method of result designation. The best individual ever encountered during the run (i.e. the best-so-far individual) is typically designated as the result of the run.

All programs in the initial random population (generation 0) of a run of GP are syntactically valid, executable programs. The genetic operations that are performed during the run (i.e. crossover, mutation, reproduction and the architecture-altering operations) are designed to produce offspring that are syntactically valid, executable programs. Thus, every individual created during a run of GP (including, in particular, the best-of-run individual) is a syntactically valid, executable program.

There are numerous alternative implementations of GP that vary from the preceding brief description. We will discuss some options in Sect. 6.5.

6.4 Example of a Run of GP

To provide concreteness, this section contains an illustrative run of GP in which the goal is to automatically create a computer program whose output is equal to the values of the quadratic polynomial $x^2 + x + 1$ in the range from -1 to $+1$. That is, the goal is to automatically create a computer program that matches certain numerical data. This process is sometimes called *system identification* or *symbolic regression*.

We begin with the five preparatory steps.

The purpose of the first two preparatory steps is to specify the ingredients of the to-be-evolved program.

Because the problem is to find a mathematical function of one independent variable, the terminal set (inputs to the to-be-evolved program) includes the independent variable, x . The terminal set also includes numerical constants. That is, the terminal set is $T = \{x, \mathfrak{R}\}$, where \mathfrak{R} denotes constant numerical terminals in some reasonable range (say from -5.0 to $+5.0$).

The preceding statement of the problem is somewhat flexible in that it does not specify what functions may be employed in the to-be-evolved program. One possible choice for the function set consists of the four ordinary arithmetic functions of addition, subtraction, multiplication, and division. This choice is reasonable because mathematical expressions typically include these functions. Thus, the function set for this problem is $F = \{+, -, *, \%\}$, where the two-argument $+$, $-$, $*$ and $\%$ functions add, subtract, multiply and divide, respectively. To avoid run-time errors, the division function $\%$ is protected: it returns a value of 1 when division by 0 is attempted (including 0 divided by 0), but otherwise returns the quotient of its two arguments.

Each individual in the population is a composition of functions from the specified function set and terminals from the specified terminal set.

The third preparatory step involves constructing the fitness measure. The purpose of the fitness measure is to specify what the human wants. The high-level goal of this problem is to find a program whose output is equal to the values of the quadratic polynomial $x^2 + x + 1$. Therefore, the fitness assigned to a particular individual in the population for this problem must reflect how closely the output of an individual program comes to the target polynomial $x^2 + x + 1$. The fitness measure could be defined as the value of the integral (taken over values of the independent variable x between -1.0 and $+1.0$) of the absolute value of the differences (errors) between the value of the individual mathematical expression and the target quadratic polynomial $x^2 + x + 1$. A smaller value of fitness (error) is better. A fitness (error) of zero would indicate a perfect fit.

For most problems of symbolic regression or system identification it is not practical or possible to analytically compute the value of the integral of the absolute error. Thus, in practice, the integral is numerically approximated using dozens or hundreds of different values of the independent variable x in the range between -1.0 and $+1.0$.

The population size in this small illustrative example will be just four. In actual practice, the population size for a run of GP consists of thousands or millions of individuals. In actual practice, the crossover operation is commonly performed on about 90 % of the individuals in the population, the reproduction operation is performed on about 8 % of the population, the mutation operation is performed on about 1 % of the population, and the architecture-altering operations are performed on perhaps 1 % of the population. Because this illustrative example involves an abnormally small population of only four individuals, the crossover operation will be performed on two individuals and the mutation and reproduction operations will each be performed on one individual. For simplicity, the architecture-altering operations are not used for this problem.

A reasonable termination criterion for this problem is that the run will continue from generation to generation until the fitness of some individual gets below 0.01.

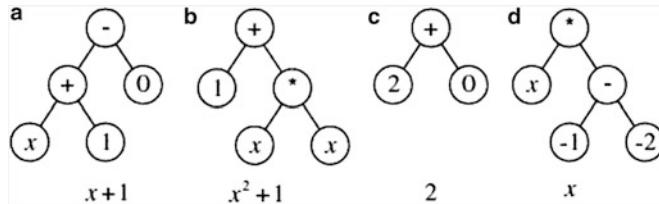


Fig. 6.10 Initial population of four randomly created individuals of generation 0

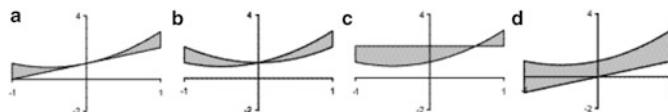


Fig. 6.11 The fitness of each of the four randomly created individuals of generation 0 is equal to the area between two curves

In this contrived example, the run will (atypically) yield an algebraically perfect solution (for which the fitness measure attains the ideal value of zero) after merely one generation.

Now that we have performed the five preparatory steps, the run of GP can be launched. That is, the executional steps shown in the flowchart of Fig. 6.4 are now performed.

GP starts by randomly creating a population of four individual computer programs. The four programs are shown in Fig. 6.10 in the form of trees.

The first randomly constructed program tree (Fig. 6.10a) is equivalent to the mathematical expression $x + 1$. A program tree is executed in a depth-first way, from left to right, in the style of the LISP programming language. Specifically, the addition function ($+$) is executed with the variable x and the constant value 1 as its two arguments. Then, the two-argument subtraction function ($-$) is executed. Its first argument is the value returned by the just-executed addition function. Its second argument is the constant value 0 . The overall result of executing the entire program tree is thus $x + 1$.

The first program (Fig. 6.10a) was constructed, using the “Grow” method, by first choosing the subtraction function for the root (top point) of the program tree. The random construction process continued in a depth-first fashion (from left to right) and chose the addition function to be the first argument of the subtraction function. The random construction process then chose the terminal x to be the first argument of the addition function (thereby terminating the growth of this path in the program tree). The random construction process then chose the constant terminal 1 as the second argument of the addition function (thereby terminating the growth along this path). Finally, the random construction process chose the constant terminal 0 as the second argument of the subtraction function (thereby terminating the entire construction process).

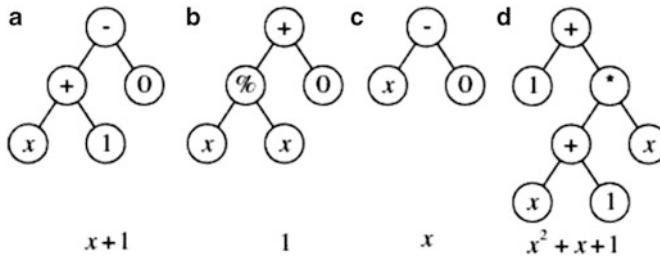


Fig. 6.12 Population of generation 1 (after one reproduction, one mutation, and one two-offspring crossover operation)

The second program (Fig. 6.10b) adds the constant terminal 1 to the result of multiplying x by x and is equivalent to $x^2 + 1$. The third program (Fig. 6.10c) adds the constant terminal 2 to the constant terminal 0 and is equivalent to the constant value 2. The fourth program (Fig. 6.10d) is equivalent to x .

Randomly created computer programs will, of course, typically be very poor at solving the problem at hand. However, even in a population of randomly created programs, some programs are better than others. The four random individuals from generation 0 in Fig. 6.10 produce outputs that deviate from the output produced by the target quadratic function $x^2 + x + 1$ by different amounts. In this particular problem, fitness can be graphically illustrated as the area between two curves. That is, fitness is equal to the area between the parabola $x^2 + x + 1$ and the curve representing the candidate individual. Figure 6.11 shows (as shaded areas) the integral of the absolute value of the errors between each of the four individuals in Fig. 6.10 and the target quadratic function $x^2 + x + 1$. The integral of absolute error for the straight line $x + 1$ (the first individual) is 0.67 (Fig. 6.11a). The integral of absolute error for the parabola $x^2 + 1$ (the second individual) is 1.0 (Fig. 6.11b). The integrals of the absolute errors for the remaining two individuals are 1.67 (Fig. 6.11c) and 2.67 (Fig. 6.11d), respectively.

As can be seen in Fig. 6.11, the straight line $x + 1$ (Fig. 6.11a) is closer to the parabola $x^2 + x + 1$ in the range from -1 to $+1$ than any of its three cohorts in the population. This straight line is, of course, not equivalent to the parabola $x^2 + x + 1$. This best-of-generation individual from generation 0 is not even a quadratic function. It is merely the best candidate that happened to emerge from the blind random search of generation 0. In the valley of the blind, the one-eyed man is king.

After the fitness of each individual in the population is ascertained, GP then probabilistically selects relatively fitter programs from the population. The genetic operations are applied to the selected individuals to create offspring programs. The most commonly employed methods for selecting individuals to participate in the genetic operations are tournament selection and fitness-proportionate selection. In both methods, the emphasis is on selecting relatively fit individuals. An important feature common to both methods is that the selection is not greedy. Individuals that are known to be inferior will be selected to a certain degree. The best individual in

the population is not guaranteed to be selected. Moreover, the worst individual in the population will not necessarily be excluded. Anything can happen and nothing is guaranteed.

We first perform the reproduction operation. Because the first individual is the most fit individual in the population (Fig. 6.10a), it is very likely to be selected to participate in a genetic operation. Let us suppose that this particular individual is, in fact, selected for reproduction. If so, 0 is copied, without alteration, into the next generation (generation 1). It is shown in Fig. 6.12a as part of the population of the new generation.

We next perform the mutation operation. Because selection is probabilistic, it is possible that the third-best individual in the population (Fig. 6.10c) is selected. One of the three nodes of this individual is then randomly picked as the site for the mutation. In this example, the constant terminal 2 is picked as the mutation site. This program is then randomly mutated by deleting the entire subtree rooted at the picked point (in this case, just the constant terminal 2) and inserting a subtree that is randomly grown in the same way that the individuals of the initial random population were originally created. In this particular instance, the randomly grown subtree computes the quotient of x and x using the protected division operation %. The resulting individual is shown in Fig. 6.12b. This particular mutation changes the original individual from one having a constant value of 2 into one having a constant value of 1. This particular mutation improves fitness from 1.67 to 1.00.

Finally, we perform the crossover operation. Because the first and second individuals in generation 0 are both relatively fit, they are likely to be selected to participate in crossover. The selection (and reselection) of relatively fitter individuals and the exclusion and extinction of unfit individuals is a characteristic feature of Darwinian selection. The first and second programs are mated sexually to produce two offspring (using the two-offspring version of the crossover operation). One point of the first parent (Fig. 6.10a), namely the + function, is randomly picked as the crossover point for the first parent. One point of the second parent (Fig. 6.10b), namely its leftmost terminal x , is randomly picked as the crossover point for the second parent. The crossover operation is then performed on the two parents. The two offspring are shown in Fig. 6.12c, d. One of the offspring (Fig. 6.12c) is equivalent to x and is not noteworthy. However, the other offspring (Fig. 6.12d) is equivalent to $x^2 + x + 1$ and has a fitness (integral of absolute errors) of zero. Because the fitness of this individual is below 0.01, the termination criterion for the run is satisfied and the run is automatically terminated. This best-so-far individual (Fig. 6.12d) is designated as the result of the run. This individual is an algebraically correct solution to the problem.

Note that the best-of-run individual (Fig. 6.12d) incorporates a good trait (the quadratic term x^2) from the second parent (Fig. 6.10b) with two other good traits (the linear term x and constant term of 1) from the first parent (Fig. 6.10a). The crossover operation produced a solution to this problem by recombining good traits from these two relatively fit parents into a superior (indeed, perfect) offspring.

In summary, GP has, in this example, automatically created a computer program whose output is equal to the values of the quadratic polynomial $x^2 + x + 1$ in the range from -1 to $+1$.

6.5 Further Features of GP

Various advanced features of GP are not covered by the foregoing illustrative problem and the foregoing discussion of the preparatory and executional steps of GP. In this section we will look at a few alternatives (a more complete and detailed survey is available in [Poli et al. 2008](#)).

6.5.1 Automatically Defined Functions and Libraries

Human programmers organize sequences of repeated steps into reusable components such as subroutines, functions and classes. They then repeatedly invoke these components, typically with different inputs. Reuse eliminates the need to “reinvent the wheel” every time a particular sequence of steps is needed. Reuse also makes it possible to exploit a problem’s modularities, symmetries and regularities (thereby potentially accelerating the problem-solving process). This can be taken further, as programmers typically organize these components into hierarchies in which top-level components call lower-level ones, which call still lower levels, etc. Automatically defined functions (ADFs) provide a mechanism by which the evolutionary process can evolve these kinds of potentially reusable components. We will review the basic concepts here, but ADFs are discussed in great detail in [Koza \(1994\)](#).

When ADFs are used, a program consists of multiple components. These typically consist of one or more function-defining branches (i.e. ADFs), as well as one or more main result-producing branches (RPBs). The RPB is the “main” program that is executed when the individual is evaluated. It can, however, call the ADFs, which can in turn potentially call each other. A single ADF may be called multiple times by the same RPB, or by a combination of the RPB and other ADFs, allowing the logic that evolution has assembled in that ADF to be re-used in different contexts.

Typically, recursion is prevented by imposing an order on the ADFs within an individual and by restricting calls so that ADF_i can only call ADF_j if $i > j$. Also, in the presence of ADFs, recombination operators are typically constrained to respect the larger structure. That is, during crossover, a subtree from ADF_i can only be swapped with a subtree from another individual’s ADF_i .

The program’s RPB and its ADFs typically have different function and terminal sets. For example, the terminal set for ADFs usually include arguments, such as `arg0` and `arg1`. Typically the user must decide in advance the primitive sets, the

number of ADFs and any call restrictions to prevent recursion. These choices can be evolved using the architecture-altering operations described in Sect. 6.5.2.

There have also been proposals for the automated creation of libraries of functions within GP. For example, [Angeline and Pollack \(1992\)](#) and [Rosca and Ballard \(1996\)](#) studied the creation and use of dynamic libraries of subtrees taken from parts of the GP trees in the population.

Naturally, while including ADFs and automatically created libraries makes it possible for modular re-use to emerge, there is no guarantee that they will be used that way. For example, it may be that the RPB never calls an ADF or only calls it once. It is also possible for an ADF to not actually encapsulate any significant logic.

6.5.2 Architecture-Altering Operations

The *architecture* of a program can be defined as the total number of trees, the type of each tree, the number of arguments (if any) possessed by each tree, and, finally, if there is more than one tree, the nature of the hierarchical references (if any) allowed among the trees (e.g. whether ADF_1 can call ADF_2) ([Koza 1994](#)).

There are three ways to determine the architecture of the computer programs that will be evolved. Firstly, the user may specify in advance the architecture of the overall program, i.e. perform an *architecture-defining preparatory step* in addition to the five steps itemized in Sect. 6.2. Secondly, a run of GP may employ the *evolutionary design of the architecture* ([Koza 1994](#)), thereby enabling the architecture of the overall program to emerge from a competitive process during the run. Finally, a run may employ a set of *architecture-altering operations* ([Koza 1994, 1995; Koza et al. 1999a](#)) which, for example, can create, remove or modify ADFs. Note that architecture changes are often designed not to initially change the semantics of the program and, so, the altered program often has exactly the same fitness as its parent. Nevertheless, the new architecture may make it easier to evolve better programs later.

6.5.3 Constraining Structures

Most GP systems require that all subtrees return data of the same type. This ensures that the output of any subtree can be used as one of the inputs to any node. The basic subtree crossover operator shuffles tree components entirely randomly. Type compatibility ensures that crossover cannot lead to incompatible connections between nodes. This is also required to stop mutation from producing illegal programs.

There are cases, however, where this approach is not ideal. For example, there might be constraints on the structure of the acceptable solutions or a problem domain might be naturally represented with multiple types. To apply GP in these cases one needs to be able to use primitives with different type signatures. Below we will

look at three approaches to constraining the syntax of the evolved expression trees in GP: simple structure enforcement, strongly typed GP and grammar-based constraints.

If a particular structure is believed or known to be important then one can modify the GP system to require that all individuals have that structure (Koza 1992). Enforcing a user-specified structure on the evolved solutions can be implemented in a number of ways. For example, one can ensure that all the initial individuals have the structure of interest and then constrain crossover and mutation so that they do not alter any of the fixed regions of a tree. An alternative approach is to evolve the various (sub)components separately. A form of constraint-directed search in GP was also proposed in Tsang and Li (2002) and Tsang and Jin (2006).

Since constraints are often driven by or expressed using a type system, a natural approach is to incorporate types and their constraints into the GP system (Montana 1995). In *strongly typed GP*, every terminal has a type, and every function has types for each of its arguments and a type for its return value. The process that generates the initial, random expressions, and all the genetic operators are implemented so as to ensure that they do not violate the type system's constraints. For example, mutation replaces subtrees with new randomly generated trees ensuring that the root of the replacement tree has the same return type as the root of the excised tree. Similarly, crossover only allows the swap of subtrees having the same return type. This basic approach to types can be extended to more complex type systems (Montana 1995; Haynes et al. 1996; Olsson 1994; Yu 2001).

Another natural way to express constraints is via *grammars*, and these have been used in GP in a variety of ways (Whigham 1996; Gruau 1996; Wong and Leung 1996; O'Neill and Ryan 2003; Hoai et al. 2003). In this sort of system, the grammar is typically used to ensure that the initial population is made up of legal programs. The grammar is also used to guide the operations of the genetic operators. Thus we need to keep track not only of the program itself, but also the syntax rules used to derive it.

What actually is evolved in a grammar-based GP system depends on the particular system. Whigham (1996), for example, evolved *derivation trees*, which effectively are a hierarchical representation of which rewrite rules must be applied, and in which order, to obtain a particular program. In this system, crossover is restricted to only swapping subtrees deriving from a common non-terminal symbol in the grammar. The actual program represented by a derivation tree can be obtained by reading out the leaves of the tree one by one from left to right.

Another approach is *grammatical evolution* (GE) which represents individuals as variable-length sequences of integers which are interpreted in the context of a user-supplied grammar (Ryan et al. 1998; O'Neill and Ryan 2003). For each rule in the grammar, the set of alternatives on the right-hand side are numbered from 0 upwards. To create a program from a GE individual one uses the values in the individual to choose which alternative to take in the production rules. If a value exceeds the number of available options it is transformed via a modulus operation.

6.5.4 Developmental GP

By using appropriate terminals, functions and/or interpreters, GP can go beyond the production of computer programs. In *cellular encoding* (Gruau and Whitley 1993; Gruau 1994a,b), programs are interpreted as sequences of instructions which modify (grow) a simple initial structure (embryo). Once the program has finished, the quality of the structure it has produced is measured and this is taken to be the fitness of the program.

Naturally, for cellular encoding to work the primitives of the language must be able to grow structures appropriate to the problem domain. Typical instructions involve the insertion and/or sizing of components, topological modifications of the structure, etc. Cellular encoding GP has successfully been used to evolve neural networks (Gruau and Whitley 1993; Gruau 1994a,b) and electronic circuits (Koza et al. 1996a; Koza et al. 1996b; Koza et al. 1999b), as well as in numerous other domains. A related approach proposed by Hoang et al. (2007) combines tree-adjoining grammars with L-systems (Lindenmayer 1968) to create a system where each stage in the developmental process is a working program that respects the grammatical constraints.

One of the advantages of indirect representations such as cellular encoding is that the standard GP operators can be used to evolve structures (such as circuits) which may have nothing in common with standard GP trees. In many of these systems, the structures being “grown” are also still meaningful (and evaluable) at each point in their development. This allows fitness evaluation. Another important advantage is that structures resulting from developmental processes often have some regularity, which other methods obtain through the use of ADFs, constraints, types, etc.

6.5.5 Probabilistic GP

Genetic programming typically uses an evolutionary algorithm as its main search engine. However, this is not the only option. This section considers work where the exploration is performed by estimation of distribution algorithms (EDAs).

EDAs (Baluja and Caruana 1995; Larrañaga and Lozano 2002) are powerful population-based searchers where the variation operations traditionally implemented via crossover and mutation in EAs are replaced by the process of random sampling from a probability distribution. The distribution is modified generation after generation, using information obtained from the fitter individuals in the population. The objective of these changes in the distribution is to increase the probability of generating individuals with high fitness.

There have been several applications of probabilistic model-based evolution in the areas of tree-based and linear GP. The first EDA-style GP system was effectively an extension of the work in Baluja and Caruana (1995) to trees called probabilistic incremental program evolution (PIPE) (Salustowicz and Schmidhuber 1997). In PIPE, the population is replaced by a hierarchy of probability tables organized into

a tree. Each table represents the probability that a particular primitive will be chosen at that specific location in a newly generated program tree. At each generation a population of programs is created based on the current tree of probability tables. Then, the fitness of the new programs is computed and the probability hierarchy is updated on the basis of these fitnesses, so as to make the generation of above-average fitness programs more likely in the next generation. More recent work includes [Yanai and Iba \(2003\)](#), [Looks \(2007\)](#), [Looks et al. \(2005\)](#) and [Poli and McPhee \(2008b\)](#).

A variety of other systems have been proposed which combine the use of grammars and probabilities ([Shan et al. 2006](#)). For example, [Ratle and Sebag \(2001\)](#) use a stochastic context-free grammar to generate program trees where the probability of applying each rewrite rule is adapted using an EDA approach. A probabilistic L-system is used by [Shan et al. \(2003\)](#) while a tree-adjunct grammar is used by [Abbass et al. \(2002\)](#) and [Shan et al. \(2002\)](#).

6.5.6 Bloat and Bloat Control

In the early 1990s, researchers began to notice that in addition to progressively increasing their mean and best fitness, GP populations also showed another phenomenon: very often the average size (number of nodes) of the programs in a population after a certain number of generations would start growing at a rapid pace. Typically the increase in program size was not accompanied by any corresponding increase in fitness. This phenomenon is known as *bloat*.

Bloat has significant practical effects: large programs are computationally expensive to evolve and later use, can be hard to interpret, and may exhibit poor generalization. Note that there are situations where one would expect to see program growth as part of the process of solving a problem. For example, GP runs typically start from populations of small random programs, and it may be necessary for the programs to grow in complexity for them to be able to comply with all the fitness cases. So, we should not equate growth with bloat and we should define bloat as *program growth without (significant) return in terms of fitness*.

Numerous empirical techniques have been proposed to control bloat ([Langdon et al. 1999](#); [Soule and Foster 1998](#)). In the rest of this section we briefly review some of the most important. In Sect. 6.7 we will review a subset of theoretical explanations for bloat. More information can be found in [Poli et al. \(2008, 2010\)](#).

Rather naturally, the first and simplest method to control code growth is the use of hard limits on the size or depth of the offspring programs generated by the genetic operators. Many implementations of this idea (e.g. [Koza 1992](#)) apply a genetic operator and then check whether the offspring is beyond the size or depth limit. If it isn't, the offspring enters the population. If, instead, the offspring exceeds the limit, one of the parents is returned. A problem with this implementation is that parent programs that are more likely to violate the size limit will tend to be copied (unaltered) more often than programs that don't. That is, the population will tend to be filled up with programs that nearly infringe the size limit, which is typically not what is desired.

However, the problem can be fixed by *not returning parents* if the offspring violates a constraint. Instead, one should either return the oversize offspring, but give it a fitness of 0 so that selection will get rid of it at the next generation, or declare the genetic operation failed, and try again.

One can also control bloat by using genetic operators which directly or indirectly have an anti-bloat effect. *Size fair crossover* and *size fair mutation* (Langdon 2000; Crawford-Marks and Spector 2002) achieve this by constraining the choices made during the execution of a genetic operation so as to actively prevent growth. In size-fair crossover, for example, the crossover point in the first parent is selected randomly, as in standard crossover. Then the size of the subtree to be excised is calculated. This is used to constrain the choice of the second crossover point so as to guarantee that the subtree chosen from the second parent will not be “unfairly” big. There are also several *mutation operators* that may help control the average tree size in the population while still introducing new genetic material (e.g. see Kinnear Jr 1993, 1994b; Angelie 1996; Langdon 1998a).

As will be clarified by the size evolution equation presented in Sect. 6.7.2, in systems with symmetric operators, bloat can only happen if there are some longer-than-average programs that are fitter than average or some shorter-than-average programs that are less fit than average, or both. So, it stands to reason that in order to control bloat one needs to somehow modulate the selection probabilities of programs based on their size.

A technique known as the Tarpeian method (Poli 2003) controls bloat by acting directly on selection probabilities. This is done by setting the fitness of randomly chosen longer-than-average programs to 0. This prevents them being parents. By changing how frequently this is done the anti-bloat intensity of Tarpeian control can be modulated. An advantage of the method is that the programs whose fitness is zeroed are never executed, thereby speeding up runs.

The well-known *parsimony pressure method* changes the selection probabilities by subtracting a value based on the size of each program from its fitness (Koza 1992; Zhang and Mühlenbein 1993, 1995; Zhang et al. 1997). Bigger programs have more subtracted and, so, have lower fitness and tend to have fewer children. That is, the new fitness function is $f(x) - c \times \ell(x)$, where $\ell(x)$ is the size of program x , $f(x)$ is its original fitness and c is a constant known as the *parsimony coefficient*. Zhang and Mühlenbein (1995) showed some benefits of adaptively adjusting the coefficient c at each generation but most implementations actually keep the parsimony coefficient constant.

Recently, a theoretically sound method for setting the parsimony coefficient in a principled manner has been proposed (Poli and McPhee 2008a). This is called the *covariant parsimony pressure method*. The method is easy to implement. It recalculates the parsimony coefficient c at each generation using $c = \text{Cov}(\ell, f)/\text{Var}(\ell)$, where $\text{Cov}(\ell, f)$ is the covariance between program size ℓ and program fitness f in the population, and $\text{Var}(\ell)$ is the variance of program sizes. Using this equation ensures that the mean program size remains at the value set by the initialization procedure. There is a variant of the method that allows the user to even decide what function the mean program size should follow over time.

Table 6.1 Eight criteria for saying that an automatically created result is human-competitive

Criterion	
A	The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention
B	The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed journal
C	The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts
D	The result is publishable in its own right as a new scientific result— independent of the fact that the result was mechanically created
E	The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions
F	The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered
G	The result solves a problem of indisputable difficulty in its field
H	The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs)

6.6 Human-Competitive Results Produced by GP

Samuel’s statement (quoted in Sect. 6.1) reflects the goal articulated by the pioneers of the 1950s in the fields of artificial intelligence and machine learning, namely to use computers to automatically produce human-like results. Indeed, getting machines to produce human-like results is *the* reason for the existence of the fields of artificial intelligence and machine learning.

To make the notion of human-competitiveness more concrete, we say that a result is “human-competitive” if it satisfies one or more of the eight criteria in Table 6.1.

As can be seen from Table 6.1, the eight criteria have the desirable attribute of being at arms-length from the fields of artificial intelligence, machine learning and GP. That is, a result cannot acquire the rating of “human competitive” merely because it is endorsed by researchers *inside* the specialized fields that are attempting to create machine intelligence. Instead, a result produced by an automated method must earn the rating of “human competitive” independent of the fact that it was generated by an automated method.

Since 2004, a competition has been held annually at ACM’s *Genetic and Evolutionary Computation Conference* (termed the Human-Competitive awards—the *Humies*). The \$10,000 prize is awarded to projects that have produced automatically-created human-competitive results according to the criteria in Table 6.1. Table 6.2 lists 71 human-competitive instances where GP produced human-competitive results between 1998 and 2009. Each entry in the table is accompanied by the criteria (from Table 6.1) that establish the basis for the claim of human-competitiveness or by the Humies competition where they won a prize or received a honorable mention.

Table 6.2: Seventy-one instances of human-competitive results produced by genetic programming

	Claimed instance	Basis for claim
1	Creation of a better-than-classical quantum algorithm for the Deutsch–Jozsa “early promise” problem (Spector et al. 1998)	B, F
2	Creation of a better-than-classical quantum algorithm for Grover’s database search problem (Spector et al. 1999b)	B, F
	Creation of a quantum algorithm for the depth-two AND/OR	
3	query problem that is better than any previously published D result (Spector et al. 1999a; Barnum et al. 2000)	D
	Creation of a quantum algorithm for the depth-one OR query	
4	problem that is better than any previously published result D (Barnum et al. 2000)	D
	Creation of a protocol for communicating information	
5	through a quantum gate that was previously thought not to D permit such communication (Spector and Bernstein 2003)	D
6	Creation of a novel variant of quantum dense coding (Spector and Bernstein 2003)	D
7	Creation of a soccer-playing program that won its first two games in the Robo Cup 1997 competition (Luke 1998)	H
	Creation of a soccer-playing program that ranked in the middle of the field of 34 human-written programs in the Robo Cup 1998 competition (Andre and Teller 1999)	H
8	Creation of four different algorithms for the transmembrane segment identification problem for proteins (Koza 1994 , Sects. 18.8 and 18.10; Koza et al. 1999b , Sects. 16.5 and 17.2)	B, E
9	Creation of a sorting network for seven items using only 16 steps (Koza et al. 1999b , Sects. 21.4.4, 23.6, and 57.8.1)	A, D
	Rediscovery of the Campbell ladder topology for lowpass	
10	and highpass filters (Koza et al. 1999b , Sect. 25.15.1; Koza et al. 2003 , Sect. 5.2)	A, F
11	Rediscovery of the Zobel “ M -derived half section” and “constant K ” filter sections (Koza et al. 1999b , Sect. 25.15.2)	A, F
12	Rediscovery of the Cauer (elliptic) topology for filters (Koza et al. 1999b , Sect. 27.3.7)	A, F
13	Automatic decomposition of the problem of synthesizing a crossover filter (Koza et al. 1999b , Sect. 32.3)	A, F
14	Rediscovery of a recognizable voltage gain stage and a Darlington emitter-follower section of an amplifier and other circuits (Koza et al. 1999b , Sect. 42.3)	A, F
15	Synthesis of 60 and 96 decibel amplifiers (Koza et al. 1999b , Sect. 45.3)	A, F

	Claimed instance	Basis for claim
17	Automatic synthesis of asymmetric bandpass filter (Koza et al. 1996b)	
18	Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions (Koza et al. 1999b, Sect. 47.5.3)	A, D, G
19	Synthesis of a real-time analog circuit for time-optimal control of a robot (Koza et al. 1999b, Sect. 48.3)	G
20	Synthesis of an electronic thermometer (Koza et al. 1999b, Sect. 49.3)	A, G
21	Synthesis of a voltage reference circuit (Koza et al. 1999b, Sect. 50.3)	A, G
22	Automatic synthesis of digital-to-analog converter (DAC) circuit (Bennett III et al. 1999)	
23	Automatic synthesis of analog-to-digital (ADC) circuit (Bennett III et al. 1999)	
24	Creation of a cellular automata rule for the majority classification problem that is better than the Gacs–Kurdyumov–Levin (GKL) rule and all other known rules written by humans (Andre et al. 1996; Koza et al. 1999b, Sect. 58.4)	D, E
25	Creation of motifs that detect the D–E–A–D box family of proteins and the manganese superoxide dismutase family (Koza et al. 1999b, Sect. 59.8)	C
26	Synthesis of topology for a PID-D2 (proportional, integrative, derivative, and second derivative) controller (Koza et al. 2003, Sect. 3.7)	A, F
27	Synthesis of an analog circuit equivalent to Philbrick circuit (Koza et al. 2003, Sect. 4.3)	A, F
28	Synthesis of NAND circuit (Koza et al. 2003, Sect. 4.4)	A, F
29	Simultaneous synthesis of topology, sizing, placement, and routing of analog electrical circuits (Koza et al. 2003, Chap. 5)	
30	Synthesis of topology for a PID (proportional, integrative, and derivative) controller (Koza et al. 2003, Sect. 9.2)	A, F
31	Rediscovery of negative feedback (Koza et al. 2003, Chap. 14)	A, E, F, G
32	Synthesis of a low-voltage balun circuit (Koza et al. 2003, Sect. 15.4.1)	A
33	Synthesis of a mixed analog-digital variable capacitor circuit (Koza et al. 2003, Sect. 15.4.2)	A
34	Synthesis of a high-current load circuit (Koza et al. 2003, Sect. 15.4.3)	A
35	Synthesis of a voltage-current conversion circuit (Koza et al. 2003, Sect. 15.4.4)	A

	Claimed instance	Basis for claim
36	Synthesis of a cubic signal generator (Koza et al. 2003 , A Sect. 15.4.5)	A
37	Synthesis of a tunable integrated active filter (Koza et al. 2003 , Sect. 15.4.6)	A
38	Creation of PID tuning rules that outperform the Ziegler–Nichols and Astrom–Hagglund tuning rules (Koza et al. 2003 , Chap. 12)	A, B, D, E, F, G
39	Creation of three non-PID controllers that outperform a PID controller that uses the Ziegler–Nichols or Astrom–Hagglund tuning rules (Koza et al. 2003 , Chap. 13)	A, B, D, E, F, G
40	An evolved antenna for deployment on NASA’s Space Technology 5 Mission (Lohn et al. 2004)	Humies 2004
41	Automatic quantum computer programming: a genetic programming Approach (Spector 2004)	Humies 2004
42	Evolving local search heuristics for SAT using genetic programming (Fukunaga 2004); automated discovery of composite SAT variable-selection heuristics (Fukunaga 2002)	Humies 2004
43	How to draw a straight line using a GP: benchmarking evolutionary design against nineteenth century kinematic synthesis (Lipson 2004)	Humies 2004
44	Organization design optimization using genetic programming (Khosraviani et al. 2004)	Humies 2004
45	Discovery of human-competitive image texture feature programs using genetic programming (Lam and Ciesielski 2004)	Humies 2004
46	Novel image filters implemented in hardware (Sekanina 2003)	Humies 2004
47	Automated re-invention of six patented optical lens systems using genetic programming: two telescope eyepieces, a telescope eyepiece system, an eyepiece for optical instruments, two wide-angle eyepieces, and a telescope eyepiece (Koza et al. 2005, 2008)	Humies 2005
48	Evolution of a human-competitive quantum fourier transform algorithm using genetic programming (Massey et al. 2005)	Humies 2005
49	Evolving assembly programs: how games help microprocessor validation (Corino et al. 2005)	Humies 2005
50	Attaining human-competitive game playing with genetic programming (Sipper 2006); GP-Gammon: using genetic programming to evolve backgammon players (Azaria and Sipper 2005b); GP-Gammon: genetically programming backgammon players (Azaria and Sipper 2005a)	Humies 2005

	Claimed instance	Basis for claim
51	GP-EndChess: using genetic programming to evolve chess endgame (Hauptman and Sipper 2005)	Humies 2005
52	GP-Robocode: using genetic programming to evolve robocode players (Shichel et al. 2005)	Humies 2005
53	Evolving dispatching rules for solving the flexible job-shop problem (Tay and Ho 2008)	Humies 2005
54	Solution of differential equations with genetic programming and the stochastic Bernstein interpolation (Howard and Kolibal 2005)	Humies 2005
55	Determining equations for vegetation-induced resistance using genetic programming (Keijzer et al. 2005)	Humies 2005
56	Sallen-Key filter (Keane et al. 2005)	
57	Using evolution to learn how to perform interest point detection (Trujillo and Olague 2006a); Synthesis of interest point detectors through genetic programming (Trujillo and Olague 2006b)	Humies 2006
58	Evolution of an efficient search algorithm for the mate-in- <i>n</i> problem in chess (Hauptman and Sipper 2007)	Humies 2007
59	Evolving local and global weighting schemes in information retrieval (Cummins and O'Riordan 2006b); An analysis of the solution space for genetically programmed term-weighting schemes in information retrieval (Cummins and O'Riordan 2006a); Term-weighting in information retrieval using genetic programming: a three-stage process (Cummins and O'Riordan 2006c)	Humies 2007
60	Real-time, non-intrusive evaluation of VoIP (Raja et al. 2007)	Humies 2007
61	Automated reverse engineering of nonlinear dynamical systems (Bongard and Lipson 2007)	Humies 2007
62	Genetic programming approach for electron–alkalimetal atom collisions (Radi and El-Bakry 2007; El-Bakry and Radi 2006); Prediction of non-linear system in optics using genetic programming (Radi 2007); Genetic programming approach for flow of steady state fluid between two eccentric spheres (El-Bakry and Radi 2007)	Humies 2007
63	Genetic programming for finite algebras (Spector et al. 2008)	Humies 2008
64	Automatic synthesis of quantum computing circuit for the two-oracle AND/OR problem (Spector and Klein 2008)	
65	Automatic synthesis of quantum computing algorithms for the parity problem a special case of the hidden subgroup problem (Stadelhofer et al. 2008)	

	Claimed instance	Basis for claim
66	Automatic synthesis of mechanical vibration absorbers (Hu et al. 2008)	
67	Automatically finding patches and automated software re-pair (Nguyen et al. 2009; Weimer et al. 2009)	Humies 2009
68	GP-Rush: using genetic programming to evolve solvers for the rush hour puzzle (Hauptman et al. 2009)	Humies 2009
69	Learning invariant region descriptor operators with genetic programming and the F-measure (Perez and Olague 2008); Evolutionary learning of local descriptor operators for object recognition (Perez and Olague 2009)	Humies 2009
70	Solution of matrix Riccati differential equation for nonlinear singular system using genetic programming (Balasubramaniam and Kumar 2009)	
71	Distilling free-form natural laws from experimental data (Schmidt and Lipson 2009a,b)	

Clearly, Table 6.2 shows GP's potential as a powerful invention machine. There are 31 instances where the human-competitive result produced by GP duplicated the functionality of a previously patented invention, infringed a previously issued patent, or created a patentable new invention. These include one instance where GP has created an entity that either infringes or duplicates the functionality of a previously patented nineteenth-century invention, 21 instances where GP has done the same with respect to previously patented twentieth-century inventions, seven instances where GP has done the same with respect to previously patented twenty-first-century inventions, and two instances where GP has created a patentable new invention. The two new inventions are general-purpose controllers that outperform controllers employing tuning rules that have been in widespread use in industry for most of the twentieth century.

6.7 Genetic Programming Theory

GP is a search technique that explores the space of computer programs. As discussed above, the search for solutions to a problem starts from a group of points (random programs) in this search space. Those points that are of above average quality are then used to generate a new generation of points through crossover, mutation, reproduction and possibly other genetic operations. This process is repeated over and over again until a termination criterion is satisfied.

If we could visualize this search, we would often find that initially the population looks a bit like a cloud of randomly scattered points, but that, generation after generation, this cloud changes shape and moves in the search space following a well-defined trajectory. Because GP is a stochastic search technique, in different runs

we would observe different trajectories. These, however, would very likely show clear regularities to our eye that could provide us with a deep understanding of how the algorithm is searching the program space for the solutions to a given problem. We could probably readily see, for example, why GP is successful in finding solutions in certain runs and with certain parameter settings, and unsuccessful in/with others.

Unfortunately, it is normally impossible to exactly visualize the program search space due to its high dimensionality and complexity, and so we cannot just use our senses to understand and predict the behavior of GP.

One approach to gain an understanding of the behavior of a GP system and predict its behavior in precise terms is to define and study mathematical models of evolutionary search. There are a number of cases where this approach has been very successful in illuminating some of the fundamental processes and biases in GP systems. In this section we will review some theoretical approaches to understanding GP. The reader is referred to [Langdon and Poli \(2002\)](#) and [Poli et al. \(2008, 2010\)](#) for more extensive reviews of GP theory.

6.7.1 Models of GP Search

Schema theories are among the oldest and the best known models of evolutionary algorithms ([Holland 1992](#); [Whitley 1994](#)). Schema theories are based on the idea of partitioning the search space into subsets, called *schemata*. They are concerned with modeling and explaining the dynamics of the distribution of the population over the schemata. Modern genetic algorithm schema theory ([Stephens and Waelbroeck 1997, 1999](#)) provides exact information about the distribution of the population at the next generation in terms of quantities measured at the current generation, without having to actually run the algorithm. Exact schema theories are also available for GP systems with a variety of genetic operators ([Poli 2000a,b, 2001a](#); [Langdon and Poli 2002](#); [Poli et al. 2004](#); [Poli and McPhee 2003a,b](#)). Markov chain theory has also started being applied to GP ([Poli et al. 2001, 2004](#); [Mitavskiy and Rowe 2006](#)), although so far this hasn't been developed as fully as the schema theory.

Exact mathematical models of GP, such as schema theories and Markov chains, are probabilistic descriptions of the operations of selection, reproduction, crossover and mutation. They explicitly represent how these operations determine which areas of the program space will be sampled by GP, and with what probability. These models treat the fitness function as a black box, however. That is, there is no representation of the fact that in GP, unlike in other evolutionary techniques, the fitness function involves the execution of computer programs on a variety of inputs. In other words, schema theories and Markov chains do not tell us how fitness is distributed in the search space. Yet, without this information, we have no way of closing the loop and fully characterizing the behavior of a GP systems which is always the result of the interaction between the fitness function and the search biases of the representation and genetic operations used in the system.

Fortunately, the characterization of the space of computer programs explored by GP has been another main topic of theoretical research (Langdon and Poli 2002). In this category are theoretical results showing that the distribution of functionality of non-Turing-complete programs approaches a limit as program length increases. That is, although the number of programs of a particular length grows exponentially with length, beyond a certain threshold the fraction of programs implementing any particular functionality is effectively constant. There is a substantial body of empirical evidence indicating that this happens in a variety of systems. In fact, there are also mathematical proofs of these convergence results for two important forms of programs: Lisp (tree-like) S-expressions (without side effects) and machine code programs without loops (Langdon and Poli 2002; Langdon 2002, 2003a,b,c, 2005). Also, recently, Langdon and Poli (2006) and Poli and Langdon (2006) started extending these results to Turing-complete machine code programs.

6.7.2 Bloat

In Sect. 6.5.6 we introduced the notion of bloat and described some effective mechanisms for controlling it. Below we review a subset of theoretical models and explanations for bloat. More information can be found in Poli et al. (2008, 2010) and Langdon and Poli (2002).

There have been some efforts to approximately mathematically model bloat. For example, Banzhaf and Langdon (2002) defined an *executable model of bloat* where only the fitness, the size of active code and the size of inactive code were represented (i.e. there was no representation of program structures). Fitnesses of individuals were drawn from a bell-shaped distribution, while active and inactive code lengths were modified by a size-unbiased mutation operator. Various interesting effects were reported which are very similar to corresponding effects found in GP runs. Rosca (2003) proposed a similar, but slightly more sophisticated model which also included an analogue of crossover.

A strength of these executable models is their simplicity. A weakness is that they suppress or remove many details of the representation and operators typically used in GP. This makes it difficult to verify whether all the phenomena observed in the model have analogues in GP runs, and whether all important behaviors of GP in relation to bloat are captured by the model.

In Poli (2001b) and Poli and McPhee (2003b), a *size evolution equation* for GP was developed, which provided an exact formalization of the dynamics of average program size. The equation has recently been simplified (Poli and McPhee 2008a) giving

$$E[\mu(t+1) - \mu(t)] = \sum_{\ell} \ell \times (p(\ell, t) - \Phi(\ell, t)), \quad (6.1)$$

where $\mu(t+1)$ is the mean size of the programs in the population at generation $t+1$, E is the expectation operator, ℓ is a program size, $p(\ell, t)$ is the probability of selecting programs of size ℓ from the population in generation t , and $\Phi(\ell, t)$ is

the proportion of programs of size ℓ in the current generation. The equation applies to a GP system with selection and any form of symmetric subtree crossover. (In a symmetric operator the probability of selecting particular crossover points in the parents does not depend on the order in which the parents are drawn from the population.) Note that the equation constrains what can and cannot happen size-wise in GP populations. Any explanation for bloat has to agree with it.

In particular, Eq. (6.1) shows that there can be bloat only if the selection probability $p(\ell, t)$ is different from the proportion $\Phi(\ell, t)$ for at least some ℓ . So, for bloat to happen there will have to be some small ℓ for which $p(\ell, t) < \Phi(\ell, t)$ and also some bigger ℓ for which $p(\ell, t) > \Phi(\ell, t)$ (at least on average).

We conclude this section with a recent promising explanation for bloat called the *crossover bias theory* (Poli et al. 2007; Dignum and Poli 2007), which is based on and is consistent with Eq. (6.1). The theory goes as follows. On average, each application of subtree crossover removes as much genetic material as it inserts; consequently crossover on its own does not produce growth or shrinkage. While the *mean* program size is unaffected, however, *higher moments* of the distribution are. In particular, crossover pushes the population towards a particular distribution of program sizes, known as a Lagrange distribution of the second kind, where small programs have a much higher frequency than longer ones. For example, crossover generates a very high proportion of single-node individuals. In virtually all problems of practical interest, however, very small programs have no chance of solving the problem. As a result, programs of above average size have a selective advantage over programs of below average size, and the mean program size increases. Because crossover will continue to create small programs, which will then be ignored by selection (in favor of the larger programs), the increase in average size will continue generation by generation.

6.8 Conclusions

In his seminal 1948 paper entitled *Intelligent Machinery*, Turing identified three ways by which human-competitive machine intelligence might be achieved. In connection with one of those ways, [Turing \(1948\)](#) said:

There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value.

Turing did not specify how to conduct the “genetical or evolutionary search” for machine intelligence. In particular, he did not mention the idea of a population-based in conjunction with sexual recombination (crossover) as described in John Holland’s 1975 book *Adaptation in Natural and Artificial Systems*. However, in his 1950 paper *Computing Machinery and Intelligence*, [Turing \(1950\)](#) did point out

We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications

Structure of the child machine = Hereditary material
Changes of the child machine = Mutations
Natural selection = Judgment of the experimenter.

That is, Turing perceived in 1948 and 1950 that one possibly productive approach to machine intelligence would involve an evolutionary process in which a description of a computer program (the hereditary material) undergoes progressive modification (mutation) under the guidance of natural selection (i.e. selective pressure in the form of what we now call fitness).

Today, many decades later, we can see that indeed Turing was right. GP has started fulfilling Turing's dream by providing us with a systematic method, based on Darwinian evolution, for getting computers to automatically solve hard real-life problems. To do so, it simply requires a high-level statement of what needs to be done (and enough computing power).

Turing also understood the need to evaluate objectively the behavior exhibited by machines, to avoid human biases when assessing their intelligence. This led him to propose an imitation game, now known as the Turing test for machine intelligence, whose goals are wonderfully summarized by Arthur Samuel's position statement quoted in the introduction of this chapter.

At present GP is certainly not in a position to produce computer programs that would pass the full Turing test for machine intelligence, and it might not be ready for this immense task for centuries. Nonetheless, thanks to the constant technological improvements in GP technology, in its theoretical foundations and in computing power, GP has been able to solve tens of difficult problems with human-competitive results (see Table 6.2) in the recent past. These are a small step towards fulfilling Turing's and Samuel's dreams, but they are also early signs of things to come. It is, indeed, arguable that in a few years' time GP will be able to routinely and competently solve important problems for us in a variety of specific domains of application, even when running on a personal computer, thereby becoming an essential collaborator for many of human activities. This, we believe, will be a remarkable step forward towards achieving true, human-competitive machine intelligence.

Tricks of the Trade

Newcomers to the field of GP often ask themselves (and/or other more experienced genetic programmers) questions such as the following:

1. Will GP be able to solve my problem?
2. What is the best way to get started with GP? Which books or papers should I read?
3. Should I implement my own GP system or should I use an existing package? If so, what package should I use?

In the rest of this section we will try to answer the first of these questions by considering the ingredients of successful GP applications, while in the next section

we will review some of the wide variety of available sources on GP which should assist readers who wish to explore further.

Based on the experience of numerous researchers over many years, it appears that GP and other evolutionary computation methods have been especially productive in areas having some or all of the following properties:

- The interrelationships among the relevant variables is unknown or poorly understood (or where it is suspected that the current understanding may possibly be wrong).
- Finding the size and shape of the ultimate solution is a major part of the problem.
- Significant amounts of test data are available in computer-readable form.
- There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions.
- Conventional mathematical analysis does not, or cannot, provide analytic solutions.
- An approximate solution is acceptable (or is the only result that is ever likely to be obtained).
- Small improvements in performance are routinely measured (or easily measurable) and highly prized.

Sources of Additional Information

Key Books and Journals

There are more than 30 books written in English principally on GP or its applications with more being written. These start with Koza's 1992 book *Genetic Programming* (often referred to as Jaws). Koza has subsequently published three additional books on GP: *Genetic Programming II: Automatic Discovery of Reusable Programs* (1994) deals with ADFs; *Genetic Programming 3* (1999) covers, in particular, the evolution of analog circuits; *Genetic Programming 4* (2003) uses GP for automatic invention. MIT Press published three volumes in the series *Advances in Genetic Programming* (Kinnear Jr 1994a; Angeline and Kinnear Jr 1996; Spector et al. 1999c). The joint GP/genetic algorithms Kluwer book series now contains over 10 books starting with *Genetic Programming and Data Structures* (Langdon 1998b). Apart from Jaws, these tend to be for the GP specialist. The late 1990s saw the introduction of the first textbook dedicated to GP (Banzhaf et al. 1998).

The 2008 book *A Field Guide to Genetic Programming* (Poli et al. 2008) provides a gentle introduction to GP as well as a review of its different flavors and application domains. The book is freely available on the Internet in PDF and HTML formats.

Other titles include Iba (1996), Jacob (1997, 2001), Wong and Leung (2000), Nordin and Johanna (2003), Ryan (1999), Nordin (1997), Bickle (1996), Babovic (1996), Balic (1999), Bhanu et al. (2005), Brabazon and O'Neill (2006), Brezocnik (2000), Chen (2002), Dracopoulos (1997), Eiben and Smith (2003), Krawiec

(2004), Nikolaev and Iba (2006), Riolo and Worzel (2003), Rothlauf (2006), Sekanina (2003) and Spector (2004).

Readers interested in mathematical and empirical analyses of GP behavior may find *Foundations of Genetic Programming* (Langdon and Poli 2002) useful.

Each of Koza's four books has an accompanying video. These videos are now available in DVD format. Also, a small set of videos on specific GP techniques and applications is available via online resources such as Google Video and YouTube.

In addition to GP's own *Genetic Programming and Evolvable Machines* journal, *Evolutionary Computation*, the *IEEE transaction on Evolutionary Computation*, *Complex Systems* (Complex Systems Publication, Inc.), and many others publish GP articles. The GP bibliography (Langdon et al. 1995–2012) lists several hundred different journals worldwide that have published articles related to GP.

GP Implementations

One of the reasons behind the success of GP is that it is easy to implement own versions, and implementing a simple GP system from scratch remains an excellent way to make sure one really understands the mechanics of GP. In addition to being an exceptionally useful exercise, it is often easier to customize (e.g. adding new, application specific genetic operators or implementing unusual, knowledge-based initialization strategies) a system one has built for new purposes than a large GP distribution. All of this, however, requires reasonable programming skills and the will to thoroughly test the resulting system until it behaves as expected.

This is actually an extremely tricky issue in highly stochastic systems such as GP. The problem is that almost any system will produce “interesting” behavior, but it is typically very hard to test whether it is exhibiting the *correct* interesting behavior. It is remarkably easy for small mistakes to go unnoticed for extended periods of time (even years). It is also easy to incorrectly assume that “minor” implementation decisions will not significantly affect the behavior of the system.

An alternative is to use one of the many public domain GP implementations and adapt this for one's purposes. This process is faster, and good implementations are often robust, efficient, well documented and comprehensive. The small price to pay is the need to study the available documentation and examples. These often explain how to modify the GP system to some extent. However, deeper modifications (such as the introduction of new or unusual operators) will often require studying the actual source code and a substantial amount of trial and error. Good publicly available GP implementations include Lil-GP (Punch and Zongker 1998), ECJ (Luke et al. 2000–2013), Open Beagle (Gagné and Parizeau 2002) and GPC++ (Fraser and Weinbrenner 1993–1997). The most prominent commercial implementation remains Discipulus (RML Technologies 1998–2011); see Foster (2001) for a review.

While the earliest GP systems were implemented in Lisp, people have since coded GP in a huge range of different languages, including C/C++, Java, Python,

JavaScript, Perl, Prolog, Mathematica, Pop-11, MATLAB, Fortran, Occam and Haskell. Typically, these evolve expressions and programs which look like simplified Lisp. More complex target languages can be supported, however, especially with the use of more advanced tools such as grammars and type systems. Conversely, many successful programs in machine code or low-level languages have also climbed from the primordial ooze of initial randomness.

Online Resources

Online resources appear, disappear, and move with great speed, so the addresses here, which were correct at the time of writing, are obviously subject to change without notice after publication. Hopefully, the most valuable resources should be readily findable using standard search tools.

A key online resource is the GP bibliography ([Langdon et al. 1995–2012](#)) available from <http://www.cs.bham.ac.uk/~wbl/biblio/>. At the time of writing, this bibliography contains over 8,000 GP entries, roughly half of which can be downloaded immediately.

The GP bibliography has a variety of interfaces. It allows for quick jumps between papers linked by authors and allows one to sort the author list by the number of GP publications. Full references are provided in both `BIBTEX` and `Refer` formats for direct inclusion in papers written in `LATEX` and Microsoft Word, respectively. The GP bibliography is also part of the Collection of Computer Sciences Bibliographies ([Achilles and Ortyl 1995–2012](#)), which provides a comprehensive Lucerne syntax search engine.

From early on there has been an active, open email discussion list: the GP-list ([Genetic Programming Mailing List 2001–2013](#)). The `EC-Digest` ([1985–2013](#)) is a moderated list covering evolutionary computation more broadly, and often contains GP related announcements.

Koza's <http://www.genetic-programming.org/> contains a ton of useful information for the novice, including a short tutorial on “What is Genetic Programming” and the Lisp implementation of GP from *Genetic Programming* ([Koza 1992](#)).

References

- Abbass H, Hoai N, McKay R (2002) AntTAG: a new method to compose computer programs using colonies of ants. In: Proceedings of the CEC 2002, Honolulu, pp 1654–1659
- Achilles A-C, Ortyl P (1995–2013) The collection of computer science bibliographies. Available from <http://liinwww.ira.uka.de/bibliography/>

- Andre D, Teller A (1999) Evolving team Darwin united. In: Asada M, Kitano H (eds) RoboCup-98: robot soccer world cup II. LNCS 1604. Springer, Berlin, pp 346–351
- Andre D, Bennett FH III, Koza JR (1996) Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming, Stanford. MIT, Cambridge, pp 3–11
- Angeline PJ (1996) An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming, Stanford. MIT, Cambridge, pp 21–29
- Angeline PJ, Kinnear KE Jr (eds) (1996) Advances in genetic programming 2. MIT, Cambridge
- Angeline PJ, Pollack JB (1992) The evolutionary induction of subroutines. In: Proceedings of the 14th annual conference of the cognitive science society. Lawrence Erlbaum, Abingdon, Indiana University, Bloomington, pp 236–241
- Azaria Y, Sipper M (2005a) GP-gammon: genetically programming backgammon players. *Genet Program Evol Mach* 6:283–300. Published online: 12 Aug 2005
- Azaria Y, Sipper M (2005b) GP-gammon: using genetic programming to evolve backgammon players. In: Keijzer M et al (eds) Proceedings of the 8th European conference on genetic programming, Lausanne. LNCS 3447. Springer, Berlin, pp 132–142
- Babovic V (1996) Emergence, evolution, intelligence; hydroinformatics—a study of distributed and decentralised computing using intelligent agents. AA Balkema, Rotterdam
- Balasubramaniam P, Kumar AVA (2009) Solution of matrix Riccati differential equation for nonlinear singular system using genetic programming. *Genet Program Evol Mach* 10:71–89
- Balic J (1999) Flexible manufacturing systems; development—structure—operation—handling—tooling. Manufacturing technology. DAAAM International, Vienna
- Baluja S, Caruana R (1995) Removing the genetics from the standard genetic algorithm. In: Prieditis A, Russell S (eds) Proceedings of the 12th international conference on machine learning, Tahoe City. Morgan Kaufmann, San Francisco, pp 38–46
- Banzhaf W, Langdon WB (2002) Some considerations on the reason for bloat. *Genet Program Evol Mach* 3:81–91
- Banzhaf W, Nordin P, Keller RE, Francone FD (1998) Genetic programming—an introduction; on the automatic evolution of computer programs and its applications. Morgan Kaufmann, San Francisco
- Barnum H, Bernstein HJ, Spector L (2000) Quantum circuits for OR and AND of ORs. *J Phys A* 33:8047–8057
- Bennett FH III, Koza JR, Keane MA, Yu J, Mydlowec W, Stiffelman O (1999) Evolution by means of genetic programming of analog circuits that perform digital functions. In: Banzhaf W et al (eds) GECCO 1999, Orlando, vol 2. Morgan Kaufmann, San Mateo, pp 1477–1483

- Bhanu B, Lin Y, Krawiec K (2005) Evolutionary synthesis of pattern recognition systems. Monographs in computer science. Springer, New York
- Blickle T (1996) Theory of evolutionary algorithms and application to system synthesis. PhD thesis, Swiss Federal Institute of Technology, Zurich
- Bongard J, Lipson H (2007) Automated reverse engineering of nonlinear dynamical systems. *Proc Natl Acad Sci* 104:9943–9948
- Brabazon A, O'Neill M (2006) Biologically inspired algorithms for financial modelling. Natural computing series. Springer, Berlin
- Brameier M, Banzhaf W (2007) Linear genetic programming. Genetic and evolutionary computation series, vol XVI. Springer, Berlin
- Brezocnik M (2000) Uporaba genetskega programiranja v inteligentnih proizvodnih sistemih. University of Maribor, Slovenia
- Chen S-H (ed) (2002) Genetic algorithms and genetic programming in computational finance. Kluwer, Dordrecht
- Corno F, Sanchez E, Squillero G (2005) Evolving assembly programs: how games help microprocessor validation. *IEEE Trans Evol Comput* 9:695–706
- Crawford-Marks R, Spector L (2002) Size control via size fair genetic operators in the PushGP genetic programming system. In: Langdon WB et al (eds) GECCO 2002, New York. Morgan Kaufmann, San Mateo, pp 733–739
- Cummins R, O'Riordan C (2006a) An analysis of the solution space for genetically programmed term-weighting schemes in information retrieval. In: Bell DA (ed) AICS 2006, Belfast
- Cummins R, O'Riordan C (2006b) Evolving local and global weighting schemes in information retrieval. *Inf Retr* 9:311–330
- Cummins R, O'Riordan C (2006c) Term-weighting in information retrieval using genetic programming: a three stage process. In: Brewka G et al (eds) The 17th European conference on artificial intelligence, Riva del Garda. IOS, Amsterdam, pp 793–794
- Dignum S, Poli R (2007) Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In: Thierens D et al (eds) GECCO 2007, London, vol 2. ACM, New York, pp 1588–1595
- Dracopoulos DC (1997) Evolutionary learning algorithms for neural adaptive control. Perspectives in neural computing. Springer, Berlin
- EC-Digest (1985–2013). Available from <http://ec-digest.research.ucf.edu/>
- Eiben AE, Smith JE (2003) Introduction to evolutionary computing. Springer, Berlin
- El-Bakry SY, Radi A (2006) Genetic programming approach for electron–alkali–metal atom collisions. *Int J Mod Phys B* 20:5463–5471
- El-Bakry MY, Radi A (2007) Genetic programming approach for flow of steady state fluid between two eccentric spheres. *Appl Rheol* 17:68210
- Foster JA (2001) Review: discipulus: a commercial genetic programming system. *Genet Program Evol Mach* 2:201–203
- Fraser A, Weinbrenner T (1993–1997) GPC++ genetic programming C++ class library. Available from <http://www0.cs.ucl.ac.uk/staff/ucacbb/ftp/weinbenner/gp.html>

- Fukunaga A (2002) Automated discovery of composite SAT variable selection heuristics. In: Proceedings of the national conference on artificial intelligence, Edmonton, pp 641–648
- Fukunaga AS (2004) Evolving local search heuristics for SAT using genetic programming. In: Deb K et al (eds) GECCO 2004, Seattle. LNCS 3103. Springer, Berlin, pp 483–494
- Gagné C, Parizeau M (2002) BEAGLE: a new C++ evolutionary computation framework. In: Langdon WB et al (eds) Proceedings of the GECCO. Morgan Kaufmann, San Mateo, New York, p 888
- Genetic Programming Mailing List (2001–2013). Available at http://tech.groups.yahoo.com/group/genetic_programming/
- Gruau F (1994a) Neural network synthesis using cellular encoding and the genetic algorithm. PhD thesis, Laboratoire de l’Informatique du Parallelisme, Ecole Normale Supérieure de Lyon
- Gruau F (1994b) Genetic micro programming of neural networks. In: Kinnear KE Jr (ed) Advances in genetic programming, ch 24. MIT, Cambridge, pp 495–518
- Gruau F (1996) On using syntactic constraints with genetic programming. In: Angeline PJ, Kinnear KE Jr (eds) Advances in genetic programming 2, ch 19. MIT, Cambridge, pp 377–394
- Gruau F, Whitley D (1993) Adding learning to the cellular development process: a comparative study. *Evol Comput* 1:213–233
- Hauptman A, Sipper M (2005) GP-endchess: using genetic programming to evolve chess endgame players. In: Keijzer M et al (eds) Proceedings of the 8th European conference on genetic programming, Lausanne. LNCS 3447. Springer, Berlin, pp 120–131
- Hauptman A, Sipper M (2007) Evolution of an efficient search algorithm for the mate-in-N problem in chess. In: Ebner M et al (eds) Proceedings of the 10th European conference on genetic programming, Valencia. LNCS 4445. Springer, Berlin, pp 78–89
- Hauptman A, Elyasaf A, Sipper M, Karmon A (2009) GP-rush: using genetic programming to evolve solvers for the rush hour puzzle. In: Raidl G et al (eds) GECCO 2009, Montreal. ACM, New York, pp 955–962
- Haynes TD, Schoenfeld DA, Wainwright RL (1996) Type inheritance in strongly typed genetic programming. In: Angeline PJ, Kinnear KE Jr (eds) Advances in genetic programming 2, ch 18. MIT, Cambridge, pp 359–376
- Hoai NX, McKay RI, Abbass HA (2003) Tree adjoining grammars, language bias, and genetic programming. In: Ryan C et al (eds) Proceedings of the EuroGP 2003, Essex. LNCS 2610. Springer, Berlin, pp 335–344
- Hoang T-H, Essam D, McKay RI, Nguyen XH (2007) Building on success in genetic programming: adaptive variation and developmental evaluation. In: Proceedings of the 2007 international symposium on intelligent computation and applications, Wuhan. China University of Geosciences Press
- Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor

- Holland JH (1992) *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence*. MIT, Cambridge. First published by University of Michigan Press 1975
- Howard D, Kolibal K (2005) Solution of differential equations with genetic programming and the stochastic Bernstein interpolation. Technical report BDS-TR-2005-001, University of Limerick
- Hu J, Goodman ED, Li S, Rosenberg R (2008) Automated synthesis of mechanical vibration absorbers using genetic programming. *Artif Intell Eng Des Anal Manuf* 22:207–217
- Iba H (1996) *Genetic programming*. Tokyo Denki University Press, Tokyo
- Jacob C (1997) *Principia Evolvica—Simulierte Evolution mit Mathematica*. dpunkt.verlag, Heidelberg
- Jacob C (2001) *Illustrating evolutionary computation with mathematica*. Morgan Kaufmann, San Mateo
- Keane MA, Koza JR, Streeter MJ (2005) Human-competitive automated engineering design and optimization by means of genetic programming. In: Periaux J et al (eds) *Evolutionary algorithms and intelligent tools in engineering optimization*. WIT, Southampton
- Keijzer M, Baptist M, Babovic V, Uthurburu JR (2005) Determining equations for vegetation induced resistance using genetic programming. In: Beyer H-G et al (eds) *GECCO 2005*, Washington, DC, vol 2. ACM, New York, pp 1999–2006
- Khosroviani B, Levitt RE, Koza JR (2004) Organization design optimization using genetic programming. In: Keijzer M (ed) *Late breaking papers at GECCO 2004*, Seattle
- Kinnear KE Jr (1993) Evolving a sort: lessons in genetic programming. In: *Proceedings of the 1993 international conference on neural networks*, vol 2. IEEE, Piscataway, San Francisco, CA, pp 881–888
- Kinnear KE Jr (ed) (1994a) *Advances in genetic programming*. MIT, Cambridge
- Kinnear KE Jr (1994b) Fitness landscapes and difficulty in genetic programming. In: *Proceedings of the 1994 IEEE world conference on computational intelligence*, Orlando, vol 1. IEEE, Piscataway, pp 142–147
- Koza JR (1992) *Genetic programming: on the programming of computers by means of natural selection*. MIT, Cambridge
- Koza JR (1994) *Genetic programming II: automatic discovery of reusable programs*. MIT, Cambridge
- Koza JR (1995) Two ways of discovering the size and shape of a computer program to solve a problem. In: Eshelman L (ed) *Proceedings of the 6th international conference on genetic algorithms*, Pittsburgh. Morgan Kaufmann, San Mateo, pp 287–294
- Koza JR, Andre D, Bennett FH III, Keane MA (1996a) Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In: Koza JR et al (eds) *Proceedings of the 1st annual conference on genetic programming 1996*, Stanford. MIT, Cambridge, pp 132–149

- Koza JR, Bennett FH III, Andre D, Keane MA (1996b) Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming 1996, Stanford. MIT, Cambridge, pp 123–131
- Koza JR, Bennett FH III, Andre D, Keane MA (1999a) The design of analog circuits by means of genetic programming. In: Bentley P (ed) Evolutionary design by computers, ch 16. Morgan Kaufmann, San Francisco, pp 365–385
- Koza JR, Andre D, Bennett FH III, Keane MA (1999b) Genetic programming 3: Darwinian invention and problem solving. Morgan Kaufman, San Mateo
- Koza JR, Keane MA, Streeter MJ, Mydlowec W, Yu J, Lanza G (2003) Genetic programming IV: routine human-competitive machine intelligence. Kluwer, Dordrecht
- Koza JR, Al-Sakran SH, Jones LW (2005) Automated re-invention of six patented optical lens systems using genetic programming. In: Beyer H-G et al (eds) GECCO 2005, Washington, DC, vol 2. ACM, New York, pp 1953–1960
- Koza JR, Al-Sakran SH, Jones LW (2008) Automated ab initio synthesis of complete designs of four patented optical lens systems by means of genetic programming. *Artif Intell Eng Des Anal Manuf* 22:249–273
- Krawiec K (2004) Evolutionary feature programming: cooperative learning for knowledge discovery and computer vision, vol 385. Wydawnictwo Politechniki Poznanskiej, Poznań
- Lam B, Ciesielski V (2004) Discovery of human-competitive image texture feature extraction programs using genetic programming. In: Deb K et al (eds) GECCO 2004, Seattle. LNCS 3103. Springer, Berlin, pp 1114–1125
- Langdon WB (1998a) The evolution of size in variable length representations. In: IEEE international conference on evolutionary computation, Anchorage. IEEE, Piscataway, pp 633–638
- Langdon WB (1998b) Genetic programming and data structures: genetic programming + data structures = automatic programming! Genetic programming, vol 1. Kluwer, Boston
- Langdon WB (2000) Size fair and homologous tree genetic programming crossovers. *Genet Program Evol Mach* 1:95–119
- Langdon WB (2002) Convergence rates for the distribution of program outputs. In: Langdon WB et al (eds) GECCO 2002, New York. Morgan Kaufmann, San Mateo, pp 812–819
- Langdon WB (2003a) How many good programs are there? How long are they? In: De Jong KA et al (eds) Foundations of genetic algorithms VII. Morgan Kaufmann, San Mateo, pp 183–202
- Langdon WB (2003b) Convergence of program fitness landscapes. In: Cantú-Paz E et al (eds) GECCO 2003, Chicago. LNCS 2724. Springer, Berlin, pp 1702–1714
- Langdon WB (2003c) The distribution of reversible functions is normal. In: Riolo RL, Worzel B (eds) Genetic programming theory and practise, ch 11. Kluwer, Dordrecht, pp 173–188

- Langdon WB (2005) The distribution of amorphous computer outputs. In: Stepney S, Emmott S (eds) *The grand challenge in non-classical computation: international workshop*, York
- Langdon WB, Poli R (2002) Foundations of genetic programming. Springer, Berlin
- Langdon WB, Poli R (2006) The halting probability in von Neumann architectures. In: Collet P, Tomassini M, Ebner M et al (eds) *Proceedings of the 9th European conference on genetic programming*, Budapest. LNCS 3905. Springer, Berlin, pp 225–237
- Langdon WB, Gustafson SM, Koza J (1995–2012) The genetic programming bibliography. Available at <http://www.cs.bham.ac.uk/~wbl/biblio/>
- Langdon WB, Soule T, Poli R, Foster JA (1999) The evolution of size and shape. In: Spector L et al (eds) *Advances in genetic programming 3*, ch 8. MIT, Cambridge, pp 163–190
- Larrañaga P, Lozano JA (2002) Estimation of distribution algorithms, a new tool for evolutionary computation. Kluwer, Dordrecht
- Lindenmayer A (1968) Mathematic models for cellular interaction in development I and II. *J Theor Biol* 18:280–299, 300–315
- Lipson H (2004) How to draw a straight line using a GP: benchmarking evolutionary design against 19th century kinematic synthesis. In: Keijzer M (ed) *Late breaking papers at GECCO 2004*, Seattle
- Lohn J, Hornby G, Linden D (2004) Evolutionary antenna design for a NASA space-craft. In: O'Reilly U-M et al (eds) *Genetic programming theory and practice II*, ch 18. Springer, Berlin, pp 301–315
- Looks M (2007) Scalable estimation-of-distribution program evolution. In: Lipson H (ed) *GECCO 2007*, London. ACM, New York, pp 539–546
- Looks M, Goertzel B, Pennachin C (2005) Learning computer programs with the Bayesian optimization algorithm. In: Beyer H-G et al (eds) *GECCO 2005*, Washington, DC, vol 1. ACM, New York, pp 747–748
- Luke S (1998) Genetic programming produced competitive soccer softbot teams for robocup97. In: Koza JR, Banzhaf W, Chellapilla K et al (eds) *Proceedings of the 3rd annual conference on genetic programming 1998*, Madison. Morgan Kaufmann, San Mateo, pp 214–222
- Luke S, Panait L, Balan G et al (2000–2013) ECJ: a java-based evolutionary computation research system. Available at <http://cs.gmu.edu/~eclab/projects/ecj/>
- Massey P, Clark JA, Stepney S (2005) Evolution of a human-competitive quantum Fourier transform algorithm using genetic programming. In: Beyer H-G et al (eds) *GECCO 2005*, Washington, DC, vol 2. ACM, New York, pp 1657–1663
- Mitavskiy B, Rowe J (2006) Some results about the Markov chains associated to GPs and to general EAs. *Theor Comput Sci* 361:72–110
- Montana DJ (1995) Strongly typed genetic programming. *Evol Comput* 3:199–230
- Nguyen TV, Weimer W, Le Goues C, Forrest S (2009) Using execution paths to evolve software patches. In: McMinn P, Feldt R (eds) *International conference on software testing, verification and validation workshops*, Denver, pp 152–153

- Nikolaev N, Iba H (2006) Adaptive learning of polynomial networks genetic programming, backpropagation and Bayesian methods. *Genetic and evolutionary computation*, vol 4. Springer, Berlin
- Nordin P (1997) Evolutionary program induction of binary machine code and its applications. PhD thesis, der Universitat Dortmund am Fachereich Informatik
- Nordin P, Johanna W (2003) Humanoider: Sjävlaraende robotar och artificiell intelligens. Liber, Stockholm
- Olsson JR (1994) Inductive functional programming using incremental program transformation and execution of logic programs by iterative-deepening A* SLD-tree search. PhD thesis, University of Oslo
- O'Neill M, Ryan C (2003) Grammatical evolution: evolutionary automatic programming in a arbitrary language. *Genetic programming*, vol 4. Kluwer, Dordrecht
- Perez CB, Olague G (2008) Learning invariant region descriptor operators with genetic programming and the F-measure. In: 19th international conference on pattern recognition, Tampa, pp 1–4
- Perez CB, Olague G (2009) Evolutionary learning of local descriptor operators for object recognition. In: Raidl G et al (eds) GECCO 2009, Montreal. ACM, New York, pp 1051–1058
- Poli R (2000a) Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory. In: Poli R et al (eds) Proceecings of the EuroGP 2000 on genetic programming, Tübingen. LNCS 1802. Springer, Berlin, pp 163–180
- Poli R (2000b) Exact schema theorem and effective fitness for GP with one-point crossover. In: Whitley D et al (eds) GECCO 2000, Las Vegas. Morgan Kaufmann, San Mateo, pp 469–476
- Poli R (2001a) Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genet Program Evol Mach* 2:123–163
- Poli R (2001b) General schema theory for genetic programming with subtree-swapping crossover. In: Proceedings of the EuroGP 2001 on genetic programming, Como. LNCS 2038. Springer, Berlin
- Poli R (2003) A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan C et al (eds) Proceedings of the EuroGP 2003 on genetic programming, Essex. LNCS 3003. Springer, Berlin, pp 211–223
- Poli R, Langdon WB (2006) Efficient Markov chain model of machine code program execution and halting. In: Riolo RL et al (eds) *Genetic programming theory and practice IV*. *Genetic and evolutionary computation*, vol 5, ch 13. Springer, Berlin
- Poli R, McPhee NF (2003a) General schema theory for genetic programming with subtree-swapping crossover: I. *Evol Comput* 11:53–66
- Poli R, McPhee NF (2003b) General schema theory for genetic programming with subtree-swapping crossover: II. *Evol Comput* 11:169–206
- Poli R, McPhee NF (2008a) Covariant parsimony pressure in genetic programming. Technical report CES-480, University of Essex

- Poli R, McPhee NF (2008b) A linear estimation-of-distribution GP system. In: O'Neill M et al (eds) Proceedings of the EuroGP 2008, Naples. LNCS 4971. Springer, Berlin, pp 206–217
- Poli R, Rowe JE, McPhee NF (2001) Markov chain models for GP and variable-length GAs with homologous crossover. In: Spector L et al (eds) GECCO 2001, San Francisco. Morgan Kaufmann, San Mateo, pp 112–119
- Poli R, McPhee NF, Rowe JE (2004) Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genet Program Evol Mach* 5:31–70
- Poli R, Langdon WB, Dignum S (2007) On the limiting distribution of program sizes in tree-based genetic programming. In: Ebner M et al (eds) Proceedings of the 10th European conference on genetic programming, Valencia. LNCS 4445. Springer, Berlin, pp 193–204
- Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Published via <http://lulu.com> and <http://www.gp-field-guide.org.uk> (with contributions by J. R. Koza)
- Poli R, Vanneschi L, Langdon WB, McPhee NF (2010) Theoretical results in genetic programming: the next ten years? *Genet Program Evol Mach* 11:285–320. 10th anniversary issue: progress in genetic programming and evolvable machines
- Punch B, Zongker D (1998) lil-gp genetic programming system. Available at <http://garage.cse.msu.edu/software/lil-gp/>
- Radi A (2007) Prediction of non-linear system in optics using genetic programming. *Int J Mod Phys C* 18:369–374
- Radi AM, El-Bakry SY (2007) Genetic programming approach for positron collisions with alkali-metal atom. In: Thierens D et al (eds) GECCO 2007, London, vol 2. ACM, New York, pp 1756–1756
- Raja A, Atif Azad RM, Flanagan C, Ryan C (2007) Real-time, non-intrusive evaluation of voIP. In: Ebner M et al (eds) Proceedings of the 10th European conference on genetic programming, Valencia. LNCS 4445. Springer, Berlin, pp 217–228
- Ratle A, Sebag M (2001) Avoiding the bloat with probabilistic grammar-guided genetic programming. In: Collet P et al (eds) Artificial evolution 5th international conference on evolution artificielle, EA, Le Creusot. LNCS 2310. Springer, Berlin, pp 255–266
- Riolo RL, Worzel B (eds) (2003) Genetic programming theory and practice. Genetic programming, vol 6. Kluwer, Boston
- RML Technologies (1998–2011) Discipulus genetic programming software. Available from <http://www.rmltech.com/>
- Rosca J (2003) A probabilistic model of size drift. In: Riolo RL, Worzel B (eds) Genetic programming theory and practice, ch 8. Kluwer, Dordrecht, pp 119–136
- Rosca JP, Ballard DH (1996) Discovery of subroutines in genetic programming. In: Angelie PJ, Kinnear KE Jr (eds) Advances in genetic programming 2, ch 9. MIT, Cambridge, pp 177–202
- Rothlauf F (2006) Representations for genetic and evolutionary algorithms, 2nd edn. Springer, Berlin. First published 2002, 2nd edn available electronically

- Ryan C (1999) Automatic re-engineering of software using genetic programming. *Genetic programming*, vol 2. Kluwer, Dordrecht
- Ryan C, Collins JJ, O'Neill M (1998) Grammatical evolution: evolving programs for an arbitrary language. In: Banzhaf W et al (eds) *Proceedings of the 1st European workshop on genetic programming*, Paris. LNCS 1391. Springer, Berlin, pp 83–95
- Salustowicz RP, Schmidhuber J (1997) Probabilistic incremental program evolution. *Evol Comput* 5:123–141
- Schmidt M, Lipson H (2009a) Distilling free-form natural laws from experimental data. *Science* 324:81–85
- Schmidt MD, Lipson H (2009b) Solving iterated functions using genetic programming. In: Esparcia AI et al (eds) *GECCO 2009 late-breaking papers*, Montreal. ACM, New York, pp 2149–2154
- Sekanina L (2003) Evolvable components: from theory to hardware implementations. *Natural computing*. Springer, Berlin
- Shan Y, Abbass H, McKay RI, Essam D (2002) AntTAG: a further study. In: Sarker R, McKay B (eds) *Proceedings of the 6th Australia–Japan joint workshop on intelligent and evolutionary systems*, Canberra
- Shan Y, McKay RI, Abbass HA, Essam D (2003) Program evolution with explicit learning: a new framework for program automatic synthesis. In: Sarker R et al (eds) *Proceedings of the CEC 2003*, Canberra. IEEE, Piscataway, pp 1639–1646
- Shan Y, McKay RI, Essam D, Abbass HA (2006) A survey of probabilistic model building genetic programming. In: Pelikan M et al (eds) *Scalable optimization via probabilistic modeling: from algorithms to applications. Studies in computational intelligence*, vol 33, ch 6. Springer, Berlin, pp 121–160
- Shichel Y, Ziserman E, Sipper M (2005) GP-roboCode: using genetic programming to evolve roboCode players. In: Keijzer M et al (eds) *Proceedings of the 8th European conference on genetic programming*, Lausanne. LNCS 3447. Springer, Berlin, pp 143–154
- Sipper M (2006) Attaining human-competitive game playing with genetic programming. In: El Yacoubi S et al (eds) *Proceedings of the 7th international conference on cellular automata, for research and industry*, Perpignan. LNCS 4173. Springer, Berlin, p 13. (invited lectures)
- Soule T, Foster JA (1998) Effects of code growth and parsimony pressure on populations in genetic programming. *Evol Comput* 6:293–309
- Spector L (2004) Automatic quantum computer programming: a genetic programming approach. *Genetic programming*, vol 7. Kluwer, Boston
- Spector L, Bernstein HJ (2003) Communication capacities of some quantum gates, discovered in part through genetic programming. In: Shapiro JH, Hirota O (eds) *Proceedings of the 6th international conference on quantum communication, measurement, and computing*, Cambridge. Rinton, Princeton, pp 500–503
- Spector L, Klein J (2008) Machine invention of quantum computing circuits by means of genetic programming. *Artif Intell Eng Des Anal Manuf* 22:275–283

- Spector L, Barnum H, Bernstein HJ (1998) Genetic programming for quantum computers. In: Koza JR et al (eds) Proceedings of the 3rd annual conference on genetic programming 1998, Madison. Morgan Kaufmann, San Mateo, pp 365–373
- Spector L, Barnum H, Bernstein HJ, Swamy N (1999a) Finding a better-than-classical quantum AND/OR algorithm using genetic programming. In: Angeline PJ et al (eds) Proceedings of the CEC 1999, Washington, DC, vol 3. IEEE, Piscataway, pp 2239–2246
- Spector L, Barnum H, Bernstein HJ, Swamy N (1999b) Quantum computing applications of genetic programming. In: Spector L et al (eds) Advances in genetic programming 3, ch 7. MIT, Cambridge, pp 135–160
- Spector L, Langdon WB, O'Reilly UM, Angeline PJ (eds) (1999c) Advances in genetic programming 3. MIT, Cambridge
- Spector L, Clark DM, Lindsay I, Barr B, Klein J (2008) Genetic programming for finite algebras. In: Keijzer M et al (eds) GECCO 2008, Atlanta. ACM, New York, pp 1291–1298
- Stadelhofer R, Banzhaf W, Suter D (2008) Evolving blackbox quantum algorithms using genetic programming. *Artif Intell Eng Des Anal Manuf* 22:285–297
- Stephens CR, Waelbroeck H (1997) Effective degrees of freedom in genetic algorithms and the block hypothesis. In: Bäck T (ed) Proceedings of the 7th international conference on genetic algorithms, East Lansing. Morgan Kaufmann, San Mateo, pp 34–40
- Stephens CR, Waelbroeck H (1999) Schemata evolution and building blocks. *Evol Comput* 7:109–124
- Tay JC, Ho NB (2008) Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Comput Ind Eng* 54:453–473
- Trujillo L, Olague G (2006a) Using evolution to learn how to perform interest point detection. In: Tang XY et al (ed) ICPR 2006, Hong Kong, vol 1, pp 211–214
- Trujillo L, Olague G (2006b) Synthesis of interest point detectors through genetic programming. In: Keijzer M et al (eds) GECCO 2006, Seattle, vol 1. ACM, New York, pp 887–894
- Tsang E, Jin N (2006) Incentive method to handle constraints in evolutionary. In: Collet P et al (eds) Proceedings of the 9th European conference on genetic programming, Budapest. LNCS 3905. Springer, Berlin, pp 133–144
- Tsang EPK, Li J (2002) EDDIE for financial forecasting. In: Chen S-H (ed) Genetic algorithms and genetic programming in computational finance, ch 7. Kluwer, Dordrecht, pp 161–174
- Turing AM (1948) Intelligent machinery. National Physical Laboratory Report. Reprinted in Ince DC (ed) (1992) Mechanical intelligence: collected works of A. M. Turing, pp 107–127. North-Holland, Amsterdam. Also reprinted in Meltzer B, Michie D (eds) (1969) Machine intelligence 5. Edinburgh University Press
- Turing AM (1950) Computing machinery and intelligence. *Mind* 49:433–460
- Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: Fickas S (ed) International conference on software engineering, Vancouver, pp 364–374

- Whigham PA (1996) Search bias, language bias, and genetic programming. In: Koza JR et al (eds) *Proceedings of the 1st annual conference on genetic programming 1996*, Stanford. MIT, Cambridge, pp 230–237
- Whitley LD (1994) A genetic algorithm tutorial. *Stat Comput* 4:65–85
- Wong ML, Leung KS (1996) Evolving recursive functions for the even-parity problem using genetic programming. In: Angeline PJ, Kinnear KE Jr (eds) *Advances in genetic programming 2*, ch 11. MIT, Cambridge, pp 221–240
- Wong ML, Leung KS (2000) Data mining using grammar based genetic programming and applications. *Genetic programming*, vol 3. Kluwer, Dordrecht
- Yanai K, Iba H (2003) Estimation of distribution programming based on bayesian network. In: Sarker R et al (eds) *Proceedings of the CEC 2003*, Canberra. IEEE, Piscataway, pp 1618–1625
- Yu T (2001) Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. *Genet Program Evol Mach* 2:345–380
- Zhang B-T, Mühlenbein H (1993) Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Syst* 7:199–220
- Zhang B-T, Mühlenbein H (1995) Balancing accuracy and parsimony in genetic programming. *Evol Comput* 3:17–38
- Zhang B-T, Ohm P, Mühlenbein H (1997) Evolutionary induction of sparse neural trees. *Evol Comput* 5:213–236

Chapter 7

Artificial Immune Systems

Uwe Aickelin, Dipankar Dasgupta, and Feng Gu

7.1 Introduction

The biological immune system is a robust, complex, adaptive system that defends the body from foreign pathogens. It is able to categorize all cells (or molecules) within the body as self or nonself substances. It does this with the help of a distributed task force that has the intelligence to take action from a local and also a global perspective using its network of chemical messengers for communication. There are two major branches of the immune system. The innate immune system is an unchanging mechanism that detects and destroys certain invading organisms, whilst the adaptive immune system responds to previously unknown foreign cells and builds a response to them that can remain in the body over a long period of time. This remarkable information processing biological system has caught the attention of computer science in recent years.

A novel computational intelligence technique inspired by immunology has emerged, called Artificial Immune Systems. Several concepts from the immune system have been extracted and applied for solution to real world science and engineering problems. In this chapter we briefly describe the immune system metaphors that are relevant to existing Artificial Immune Systems methods. We will then show illustrative real-world problems suitable for Artificial Immune Systems and give a step-by-step algorithm walkthrough for one such problem. A comparison of the Artificial Immune Systems to other well-known algorithms, areas for future work, tips and tricks and a list of resources will round this chapter off. It should be noted

U. Aickelin (✉) • F. Gu
University of Nottingham, Nottingham, UK
e-mail: uwe.aickelin@nottingham.ac.uk

D. Dasgupta
University of Memphis, Memphis, TN, USA

that as Artificial Immune Systems is still a young and evolving field, there is not yet a fixed algorithm template and hence actual implementations might differ somewhat from time to time and from those examples given here.

7.2 Overview of the Biological Immune System

The biological immune system is an elaborate defense system which has evolved over millions of years. While many details of the immune mechanisms (innate and adaptive) and processes (humeral and cellular) are yet unknown (even to immunologists), it is, however, well known that the immune system uses multilevel (and overlapping) defense both in parallel and sequential fashion. Depending on the type of the pathogen, and the way it gets into the body, the immune system uses different response mechanisms (differential pathways) either to neutralize the pathogenic effect or to destroy the infected cells. A detailed overview of the immune system can be found in many textbooks, for instance [Goldsby et al. \(2006\)](#). The immune features that are particularly relevant to our tutorial are matching, diversity and distributed control. Matching refers to the binding between antibodies and antigens. Diversity refers to the fact that, in order to achieve optimal antigen space coverage, antibody diversity must be encouraged ([Hightower et al. 1995](#)). Distributed control means that there is no central controller; rather, the immune system is governed by local interactions among immune cells and antigens.

Two important groups of cells in this process are dendritic cells and white blood cells, e.g. T-cells and B-cells. All of these originate in the bone marrow, but T-cells pass on to the thymus to mature, before they circulate the body in the blood and lymphatic vessels.

Dendritic cells (DCs) act as messengers between innate immune system and adaptive immune system, as well as mediators of various immune responses. They exist in one of three states, namely immature, semi-mature and mature. Initially immature DCs keep collecting antigens and molecular information in tissue until certain conditions are triggered. They then differentiate into either semi-mature or fully mature DCs and migrate to lymph nodes where they interact with T-cells.

T-cells are of three types; helper T-cells which are essential to the activation of B-cells, killer T-cells which bind to foreign invaders and inject poisonous chemicals into them causing their destruction, and suppressor T-cells which inhibit the action of other immune cells thus preventing allergic reactions and autoimmune diseases.

B-cells are responsible for the production and secretion of antibodies, which are specific proteins that bind to the antigen. Each B-cell can only produce one particular antibody. The antigen is found on the surface of the invading organism and the binding of an antibody to the antigen is a signal to destroy the invading cell as shown in Fig. 7.1.

As mentioned above, the human body is protected against foreign invaders by a multi-layered system. The immune system is composed of physical barriers such as the skin and respiratory system; physiological barriers such as destructive enzymes

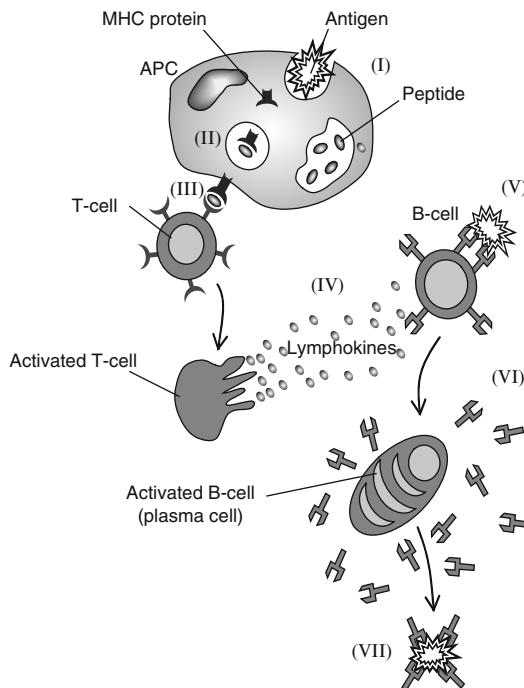


Fig. 7.1 Pictorial representation of the essence of the acquired immune system mechanism (Taken from De Castro and Van Zuben 1999): I-II show the invade entering the body and activating T-cells, which then in IV activate the B-cells, V is the antigen matching, VI the antibody production and VII the antigen's destruction

and stomach acids; and the immune system, which has can be broadly divided under two heads—innate (non-specific) immunity and adaptive (specific) immunity, which are inter-linked and influence each other. Adaptive immunity is further subdivided under two heads—humoral immunity and cell-mediated immunity.

Innate immunity. Innate immunity is present at birth. Physiological conditions such as pH, temperature and chemical mediators provide inappropriate living conditions for foreign organisms. Also micro-organisms are coated with antibodies and/or complement products (opsonization) so that they are easily recognized. Extracellular material is then ingested by macrophages by a process called phagocytosis. Also T_{DH} cells influence the phagocytosis of macrophages by secreting certain chemical messengers called lymphokines. The low levels of sialic acid on foreign antigenic surfaces make C_{3b} bind to these surfaces for a long time and thus activate alternative pathways. Thus MAC is formed, which puncture the cell surfaces and kill the foreign invader.

Adaptive immunity. Adaptive immunity is the main focus of interest here as learning, adaptability, and memory are important characteristics of adaptive immunity. It is subdivided under two heads—humoral immunity and cell-mediated immunity.

Humoral immunity. Humoral immunity is mediated by antibodies contained in body fluids (known as humors). The humoral branch of the immune system involves interaction of B-cells with antigen and their subsequent proliferation and differentiation into antibody-secreting plasma cells. Antibody functions as the effectors of the humoral response by binding to antigen and facilitating its elimination. When an antigen is coated with antibody, it can be eliminated in several ways. For example, antibody can cross-link the antigen, forming clusters that are more readily ingested by phagocytic cells. Binding of antibody to antigen on a micro-organism also can activate the complement system, resulting in lysis of the foreign organism.

Cellular immunity. Cellular immunity is cell-mediated; effector T-cells generated in response to antigen are responsible for cell-mediated immunity. Cytotoxic T-lymphocytes (CTLs) participate in cell-mediated immune reactions by killing altered self-cells; they play an important role in the killing of virus-infected cells and tumor cells. Cytokines secreted by T_{DH} can mediate the cellular immunity, and activate various phagocytic cells, enabling them to phagocytose and kill micro-organisms more effectively. This type of cell-mediated immune response is especially important in host defense against intracellular bacteria and protozoa.

Whilst there is more than one mechanism at work (for more details see Farmer et al. 1986; Kindt et al. Jerne 1973), the essential process is the matching of antigen and antibody, which leads to increased concentrations (proliferation) of more closely matched antibodies. In particular, idiotypic network theory, negative selection mechanism, and the *clonal selection* and *somatic hypermutation* theories are primarily used in Artificial Immune System models.

7.2.1 Dendritic Cells

Dendritic cells (DCs) are exposed to various molecular information or signals and antigens at their immature state in tissue. There are three main types of signals involved, including pathogen-associated molecular patterns (PAMPs), danger signals derived from uncontrolled cell death (necrosis), and safe signals resulting from programmed cell death (apoptosis).

If more PAMPs and danger signals are presented, DCs tend to differentiate into fully mature state and report an anomalous status in tissue. Conversely, if more safe signals are presented, DCs tend to differentiate into semi-mature state and report a normal status in tissue. After entering the matured states, DCs migrates from tissue to lymph nodes through blood vessels, to interact with T-cells.

Semi-mature DCs have suppressive effect on T-cells, this makes T-cells inhibited and is vital for the tolerance property of the immune system. Mature DCs have active effect on T-cells, they activate and bind with T-cells, so that they can circulate back to tissue and initialize immune responses against potential threats caused by certain antigens.

7.2.2 Immune Network Theory

The immune network theory was proposed in the mid-1970s ([Jerne 1973](#)). The hypothesis was that the immune system maintains an idiotypic network of interconnected B-cells for antigen recognition. These cells both stimulate and suppress each other in certain ways that lead to the stabilization of the network. Two B-cells are connected if the affinities they share exceed a certain threshold, and the strength of the connection is directly proportional to the affinity they share.

7.2.3 Negative Selection Mechanism

The purpose of negative selection is to provide tolerance for self cells. It deals with the immune system's ability to detect unknown antigens while not reacting to the self cells. During the generation of T-cells, receptors are made through a pseudo-random genetic rearrangement process. Then, they undergo a censoring process in the thymus, called the negative selection. There, T-cells that react against self-proteins are destroyed; thus, only those that do not bind to self-proteins are allowed to leave the thymus. These matured T-cells then circulate throughout the body to perform immunological functions and protect the body against foreign antigens.

7.2.4 Clonal Selection Principle

The clonal selection principle describes the basic features of an immune response to an antigenic stimulus. It establishes the idea that only those cells that recognize the antigen proliferate, thus being selected against those that do not. The main features of the clonal selection theory are that

- The new cells are copies of their parents (clone) subjected to a mutation mechanism with high rates (somatic hypermutation);
- Elimination of newly differentiated lymphocytes carrying self-reactive receptors;
- Proliferation and differentiation on contact of mature cells with antigens.

When an antibody strongly matches an antigen the corresponding B-cell is stimulated to produce clones of itself that then produce more antibodies. This (hyper) mutation, is quite rapid, often as much as “one mutation per cell division” ([de Castro and Von Zuben 1999](#)). This allows a very quick response to the antigens. It should be noted here that in the Artificial Immune Systems literature, often no distinction is made between B-cells and the antibodies they produce. Both are subsumed under the word *antibody* and statements such as mutation of antibodies (rather than mutation of B-cells) are common.

There are many more features of the immune system, including adaptation, immunological memory and protection against auto-immune attacks, not discussed

here. In the following sections, we will revisit some important aspects of these concepts and show how they can be modeled in *artificial* immune systems and then used to solve real-world problems. First, let us give an overview of typical problems that we believe are amenable to being solved by Artificial Immune Systems.

7.3 Illustrative Problems

7.3.1 *Intrusion Detection Systems*

Anyone keeping up to date with current affairs in computing can confirm numerous cases of attacks made on computer servers of well-known companies. These attacks range from denial-of-service attacks to extracting credit-card details and sometimes we find ourselves thinking, “haven’t they installed a firewall?” The fact is they often have a firewall. A firewall is useful, but current firewall technology is insufficient to detect and block all kinds of attacks.

On ports that need to be open to the internet, a firewall can do little to prevent attacks. Moreover, even if a port is blocked from internet access, this does not stop an attack from inside the organization. This is where Intrusion Detection Systems come in. As the name suggests, Intrusion Detection Systems are installed to identify (potential) attacks and to react by usually generating an alert or blocking the unscrupulous data.

The main goal of Intrusion Detection Systems is to detect unauthorized use, misuse and abuse of computer systems by both system insiders and external intruders. Most current Intrusion Detection Systems define suspicious signatures based on known intrusions and probes. The obvious limit of this type of Intrusion Detection Systems is its failure of detecting previously unknown intrusions. In contrast, the human immune system adaptively generates new immune cells so that it is able to detect previously unknown and rapidly evolving harmful antigens ([Forrest et al. 1994](#)). This type of detection mechanism is known as anomaly detection where the profile of normality is generated through training, and any new incoming data that deviates from the normal profile up to certain threshold is classified as anomalous. Thus the challenge is to emulate the success of the natural systems that utilize anomaly detection mechanisms.

Solutions in Artificial Immune Systems related to negative selection and clonal selection are demonstrated by [Kim et al. \(2007\)](#). Approaches derived from dendritic cells are presented by [Greensmith \(2007\)](#), recent development and applications can be found in [Al-Hammadi et al. \(2008\)](#) for Bot detection, [Gu et al. \(2009\)](#) for real-time analysis of intrusion detection, and [Oates et al. \(2007\)](#) for robotic security.

7.3.2 *Data Mining: Collaborative Filtering and Clustering*

Collaborative filtering is the term for a broad range of algorithms that use similarity measures to obtain recommendations. The best-known example is probably

the “people who bought this also bought” feature of the internet company ([Amazon 2003](#)). However, any problem domain where users are required to rate items is amenable to collaborative filtering techniques. Commercial applications are usually called recommender systems ([Resnick and Varian 1997](#)). A canonical example is movie recommendation.

In traditional collaborative filtering, the items to be recommended are treated as *black boxes*. That is, your recommendations are based purely on the votes of other users, and not on the content of the item. The preferences of a user, usually a set of votes on an item, comprise a user profile, and these profiles are compared in order to build a neighborhood. The key decision is what similarity measure is used. The most common method to compare two users is a correlation-based measure like Pearson or Spearman, which gives two neighbors a matching score between -1 and 1 . The canonical example is the k -nearest-neighbor algorithm, which uses a matching method to select k reviewers with high similarity measures. The votes from these reviewers, suitably weighted, are used to make predictions and recommendations.

The evaluation of a collaborative filtering algorithm usually centers on its accuracy. There is a difference between prediction (given a movie, predict a given user’s rating of that movie) and recommendation (given a user, suggest movies that are likely to attract a high rating). Prediction is easier to assess quantitatively but recommendation is a more natural fit to the movie domain. A related problem to collaborative filtering is that of clustering data or users in a database. This is particularly useful in very large databases, which have become too large to handle. Clustering works by dividing the entries of the database into groups, which contain people with similar preferences or in general data of similar type.

7.4 Artificial Immune System Basic Concepts

7.4.1 Initialization/Encoding

To implement a basic Artificial Immune System, four decisions have to be made: encoding, similarity measure, selection and mutation. Once an encoding has been fixed and a suitable similarity measure has been chosen, the algorithm will then perform selection and mutation, both based on the similarity measure, until stopping criteria are met. In this section, we describe each of these components in turn.

Along with other heuristics, choosing a suitable encoding is very important for the algorithm’s success. Similar to genetic algorithms, there is close inter-play between the encoding and the fitness function (the latter is in Artificial Immune Systems referred to as the *matching* or *affinity* function). Hence both ought to be thought about at the same time. For the current discussion, let us start with the encoding.

First, let us define what we mean by *antigen* and *antibody* in the context of an application domain. Typically, an antigen is the target or solution, e.g. the data item we need to check to see if it is an intrusion, or the user that we need to cluster or

make a recommendation for. The antibodies are the remainder of the data, e.g. other users in the data base, a set of network traffic that has already been identified etc. Sometimes, there can be more than one antigen at a time and there are usually a large number of antibodies present simultaneously.

Antigens and antibodies are represented or encoded in the same way. For most problems the most obvious representation is a string of numbers or features, where the length is the number of variables, the position is the variable identifier and the value (could be binary or real) of the variable. For instance, in a five-variable binary problem, an encoding could be (10010).

As mentioned previously, for data mining and intrusion detection applications. What would an encoding look like in these cases? For data mining, let us consider the problem of recommending movies. Here the encoding has to represent a user's profile with regards to the movies he has seen and how much he has (dis)liked them. A possible encoding for this could be a list of numbers, where each number represents the "vote" for an item. Votes could be binary (e.g. Did you visit this web page?), but can also be integers in a range (say [0, 5], i.e. 0—did not like the movie at all, 5—liked the movie very much).

Hence for the movie recommendation, a possible encoding is

$$User = \{\{id_1, score_1\}, \{id_2, score_2\} \dots \{id_n, score_n\}\}$$

where id corresponds to the unique identifier of the movie being rated and score to this user's score for that movie. This captures the essential features of the data available (Cayzer and Aickelin 2002).

For intrusion detection, the encoding may be to encapsulate the essence of each data packet transferred, for example

$[\langle \text{protocol} \rangle \langle \text{source ip} \rangle \langle \text{source port} \rangle \langle \text{destination ip} \rangle \langle \text{destination port} \rangle],$

example: $[\langle \text{tcp} \rangle \langle 113.112.255.254 \rangle \langle 108.200.111.12 \rangle \langle 25 \rangle],$

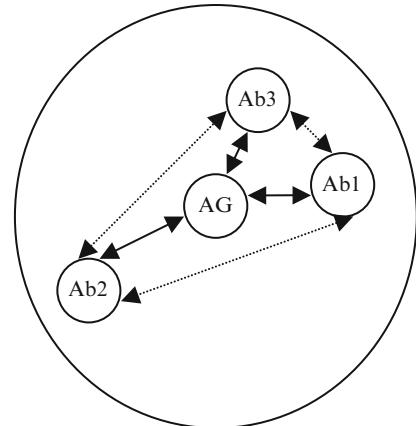
which represents an incoming data packet send to port 25. In these scenarios, wild-cards like "any port" are also often used.

7.4.2 Similarity or Affinity Measure

As mentioned in the previous section, similarity measure or matching rule is one of the most important design choices in developing an Artificial Immune Systems algorithm, and is closely coupled to the encoding scheme.

Two of the simplest matching algorithms are best explained using binary encoding: consider the strings (00000) and (00011). If one does a bit-by-bit comparison, the first 3 bits are identical and hence we could give this pair a matching score of 3. In other words, we compute the opposite of the Hamming distance (which is defined as the number of bits that have to be changed in order to make the two strings identical).

Fig. 7.2 Illustration of the idiotypic effect



Now consider this pair: (00000) and (01010). Again, simple bit matching gives us a similarity score of 3. However, the matching is quite different as the three matching bits are not connected. Depending on the problem and encoding, this might be better or worse. Thus, another simple matching algorithm is to count the number of continuous bits that match and return the length of the longest matching as the similarity measure. For the first example above this would still be 3, for the second example this would be 1.

If the encoding is non-binary, e.g. real variables, there are even more possibilities to compute the *distance* between the two strings, for instance we could compute the geometrical (Euclidian) distance, etc.

For data mining problems, like the movie recommendation system, similarity often means *correlation*. Take the movie recommendation problem as an example and assume that we are trying to find users in a database that are similar to the key user who's profile we are trying to match in order to make recommendations. In this case, what we are trying to measure is how similar are the two users' tastes. One of the easiest ways of doing this is to compute the Pearson correlation coefficient between the two users.

That is, if the Pearson measure is used to compare two user's u and v , then

$$r = \frac{\sum_{i=1}^n (u_i - \bar{u})(v_i - \bar{v})}{\sqrt{\sum_{i=1}^n (u_i - \bar{u})^2 \sum_{i=1}^n (v_i - \bar{v})^2}},$$

where u and v are users, n is the number of overlapping votes (i.e. movies for which both u and v have voted), u_i is the vote of user u for movie i and \bar{u} is the average vote of user u over all films (not just the overlapping votes). The measure is amended so default to a value of 0 if the two users have no films in common. During our research reported in [Cayzer and Aickelin \(2002a,b\)](#) we also found it useful to introduce a penalty parameter (analogous to penalties in genetic algorithms) for users who only have very few films in common, which in essence reduces their correlation.

The outcome of this measure is a value between -1 and 1 , where values close to 1 mean strong agreement, values near to -1 mean strong disagreement and values around 0 mean no correlation. From a data mining point of view, those users who score either 1 or -1 are the most useful and hence will be selected for further treatment by the algorithm.

For other applications, *matching* might not actually be beneficial and hence those items that match might be eliminated. This approach is known as *negative selection* and mirrors what is believed to happen during the maturation of B-cells who have to learn not to *match* our own tissues as otherwise we would be subject to auto-immune diseases.

Under what circumstance would a negative selection algorithm be suitable for an Artificial Immune Systems implementation? Consider the case of intrusion detection as solved by [Hofmeyr and Forrest \(2000\)](#). One way of solving this problem is by defining a set of *self*, i.e. a trusted network, our company's computers, known partners, etc. During the initialization of the algorithm, we would then randomly create a large number of so-called *detectors*, i.e. strings that looks similar to the sample intrusion detection systems encoding given above. We would then subject these detectors to a matching algorithm that compares them to our *self*. Any matching detector would be eliminated and hence we select those that do no match (negative selection). All non-matching detectors will then form our final detector set. This detector set is then used in the second phase of the algorithm to continuously monitor all network traffic. Should a match be found now the algorithm would report this as a possible alert or *nonself*. There are a number of problems with this approach, which we shall discuss further in the section *Promising Areas for Future Applications*.

7.4.3 Negative, Clonal or Neighborhood Selection

The meaning of this step differs somewhat depending on the exact problem the Artificial Immune Systems is applied to. We have already described the concept of negative selection. For the film recommender, choosing a suitable neighborhood means choosing good correlation scores and hence we perform *positive* selection. How would the algorithm use this?

Consider the Artificial Immune Systems to be empty at the beginning. The target user is encoded as the antigen, and all other users in the database are possible antibodies. We add the antigen to the Artificial Immune Systems and then we add one candidate antibody at a time. Antibodies will start with a certain concentration value. This value is decreasing over time (death rate), similar to the evaporation in Ant Systems. Antibodies with a sufficiently low concentration are removed from the system, whereas antibodies with a high concentration may saturate. However, an antibody can increase its concentration by matching the antigen—the better the match the higher the increase (a process called *stimulation*). The process of stimulation or increasing concentration can also be regarded as “cloning” if one thinks in a discrete setting. Once enough antibodies have been added to the system, it starts to

iterate a loop of reducing concentration and stimulation until at least one antibody drops out. A new antibody is added and the process repeated until the Artificial Immune Systems is stabilized, i.e. there are no more drop-outs for a certain period of time.

Mathematically, at each step (iteration) an antibody's concentration is increased by an amount dependent on its matching to each antigen. In the absence of matching, an antibody's concentration will slowly decrease over time. Hence an Artificial Immune Systems iteration is governed by the following equation, based on [Farmer et al. \(1986\)](#):

$$\frac{dx_i}{dt} = [\text{antigens recognized} - \text{death rate}] = \left[k_2 \left(\sum_{j=1}^N m_{ji} x_i y_j \right) - k_3 x_i \right],$$

where

- N is the number of antigens
- x_i is the concentration of antibody i
- y_j is the concentration of antigen j
- k_2 is the stimulation effect and k_3 is the death rate
- m_{ji} is the matching function between antibody i and antibody (or antigen) j .

The following pseudo-code summarizes the Artificial Immune Systems of the movie recommender:

```

Initialize Artificial Immune Systems
Encode user for whom to make predictions as antigen Ag
WHILE (Artificial Immune Systems not Full) & (More Antibodies) DO
    Add next user as an antibody Ab
    Calculate matching scores between Ab and Ag
    WHILE (Artificial Immune Systems at full size) & (Artificial Immune
        Systems not Stabilized) DO
        Reduce Concentration of all Abs by a fixed amount
        Match each Ab against Ag and stimulate as necessary
    OD
OD
Use final set of Antibodies to produce recommendation.

```

In this example, the Artificial Immune Systems is considered stable after iterating for ten iterations without changing in size. Stabilization thus means that a sufficient number of *good* neighbors have been identified and therefore a prediction can be made. *Poor* neighbors would be expected to drop out of the Artificial Immune Systems after a few iterations. Once the Artificial Immune Systems has stabilized using the above algorithm, we use the antibody concentration to weigh the neighbors and then perform a weighted average type recommendation.

7.4.4 Somatic Hypermutation

The mutation most commonly used in Artificial Immune Systems is very similar to that found in genetic algorithms—for example, for binary strings bits are flipped, for real value strings one value is changed at random, or for others the order of elements is swapped. In addition, the mechanism is often enhanced by the *somatic* idea—i.e. the closer the match (or the less close the match, depending on what we are trying to achieve), the more (or less) disruptive the mutation.

However, mutating the data might not make sense for all problems. For instance, it would not be suitable for the movie recommender. Certainly, mutation could be used to make users more similar to the target; however, the validity of recommendations based on these artificial users is questionable and, if overdone, we would end up with the target user itself. Hence for some problems, somatic hypermutation is not used, since it is not immediately obvious how to mutate the data sensibly such that these artificial entities still represent plausible data.

Nevertheless, for other problem domains, mutation might be very useful. For instance, taking the negative-selection approach to intrusion detection, rather than throwing away matching detectors in the first phase of the algorithm, these could be mutated to save time and effort. Also, depending on the degree of matching the mutation could be more or less strong. This was in fact one extension implemented by [Hofmeyr and Forrest \(2000\)](#).

For data mining problems, mutation might also be useful, if for instance the aim is to cluster users. Then the center of each cluster (the antibodies) could be an artificial pseudo-user that can be mutated at will until the desired degree of matching between the center and antigens in its cluster is reached. This is an approach implemented by Castro and von Zuben (2001).

7.4.5 Dendritic-Cell-Based Approaches

One of the approaches based on the behavior of dendritic cells is known as the dendritic cell algorithm (DCA) ([Greensmith 2007](#)). It is a population-based algorithm that incorporates a novel theory in immunology—danger theory, details of which are discussed in Sect. 7.6. Here we focus on describing the algorithmic properties of the DCA.

In the algorithm, there are two data streams, namely signals and antigens. Signals are represented as real-valued numbers and antigens are categorical values of the objects to be classified. The algorithm is based on a multi-agent framework, where each cell processes its own environmental signals and collects antigens. Diversity is generated within the cell population through the application of a *migration threshold*—this value limits the number of signal instances an individual cell can process during its lifespan. This creates a variable time window effect, with different cells processing the signal and antigen input streams over a range of time periods. The combination of signal/antigen correlation and the dynamics of a cell population are responsible for the detection capabilities of the DCA.

7.5 Comparison of Artificial Immune Systems to Genetic Algorithms and Neural Networks

Genetic algorithms and neural networks have already been mentioned a number of times. In fact, they both have a number of ideas in common with Artificial Immune Systems and Table 7.1 highlights their similarities and differences (see [Dasgupta 1999](#)). Evolutionary computation shares many elements; concepts like population, genotype phenotype mapping, and proliferation of the fittest are present in different Artificial Immune System methods.

Artificial Immune System models based on immune networks resemble the structures and interactions of connectionist models. Some works have pointed out the similarities and differences between Artificial Immune Systems and artificial neural networks ([Dasgupta 1999](#); [De Castro and Von Zuben 2002](#)). De Castro has also used Artificial Immune Systems to initialize the centers of radial basis function neural networks and to produce a good initial set of weights for feed-forward neural networks.

It should be noted that some of the items in Table 7.1 are gross simplifications, both to benefit the design of the table and so as not to overwhelm the reader. Some of the points are debatable; nevertheless, we believe that this comparison is valuable to show exactly how Artificial Immune Systems fit with genetic algorithms and neural networks. The comparisons are based on a genetic algorithm used for optimization and a neural network used for classification.

Table 7.1 Comparison of artificial immune systems to genetic algorithms (GAs) and neural networks (NNs)

	GA (optimization)	NN (classification)	Artificial immune systems
Components	Chromosome strings	Artificial neurons	Attribute strings
Location of components	Dynamic	Pre-defined	Dynamic
Structure	Discrete components	Networked components	Discrete/networked components
Knowledge storage	Chromosome strings	Connection strengths	Component concentration/network connections
Dynamics	Evolution	Learning	Evolution/learning
Meta-dynamics	Recruitment/elimination of components	Construction/pruning of connections	Recruitment/elimination of components
Interaction between components	Crossover	Network connections	Recognition/network connections
Interaction with environment	Fitness function	External stimuli	Recognition/objective function
Threshold activity	Crowding/sharing	Neuron activation	Component affinity

7.6 Extensions of Artificial Immune Systems

7.6.1 Idiotypic Networks: Network Interactions (Suppression)

The idiotypic effect builds on the premise that antibodies can match other antibodies as well as antigens. It was first proposed by [Jerne \(1973\)](#) and formalized into a model by [Farmer et al. \(1986\)](#). The theory is currently debated by immunologists, with no clear consensus yet on its effects in the humoral immune system ([Kindt et al.](#)). The idiotypic network hypothesis builds on the recognition that antibodies can match other antibodies as well as antigens. Hence, an antibody may be matched by other antibodies, which in turn may be matched by yet other antibodies. This activation can continue to spread through the population and potentially has much explanatory power. It could, for example, help explain how the memory of past infections is maintained. Furthermore, it could result in the suppression of similar antibodies thus encouraging diversity in the antibody pool. The idiotypic network has been formalized by a number of theoretical immunologists ([Perelson and Weisbuch 1997](#)):

$$\frac{dx_i}{dt} = c[(\text{antibodies recognized}) - (\text{I am recognized}) + (\text{antigens recognized})] \\ - (\text{death rate}) = c \left[\sum_{j=1}^N m_{ji}x_i x_j - k_1 \sum_{j=1}^N m_{ij}x_i x_j + \sum_{j=1}^n m_{ji}x_i y_j \right] - k_2 x_i, \quad (7.1)$$

where

- N is the number of antibodies and n is the number of antigens
- x_i (or x_j) is the concentration of antibody i (or j)
- y_i is the concentration of antigen j
- c is a rate constant
- k_1 is a suppressive effect and k_2 is the death rate
- m_{ji} is the matching function between antibody i and antibody (or antigen) j .

As can be seen from Eq. (7.1), the nature of an idiotypic interaction can be either positive or negative. Moreover, if the matching function is symmetric, then the balance between “I am recognized” and “Antibodies recognized” (parameters c and k_1 in the equation) wholly determines whether the idiotypic effect is positive or negative, and we can simplify the equation. We can also simplify Eq. (7.1) if we only allow one antigen in the Artificial Immune Systems. In the new Eq. (7.2), the first term is simplified as we only have one antigen, and the suppression term is normalized to allow a *like for like* comparison between the different rate constants:

$$\frac{dx_i}{dt} = k_1 m_i x_i y - \frac{k_2}{n} \sum_{j=1}^n m_{ij} x_i x_j - k_3 x_i, \quad (7.2)$$

where

- k_1 is stimulation, k_2 suppression and k_3 death rate
- m_i is the correlation between antibody i and the (sole) antigen
- x_i (or x_i) is the concentration of antibody i (or j)
- y is the concentration of the (sole) antigen
- m_{ij} is the correlation between antibodies i and j
- n is the number of antibodies.

Why would we want to use the idiotypic effect? Because it might provide us with a way of achieving *diversity*, similar to *crowding* or *fitness sharing* in a genetic algorithm. For instance, in the movie recommender, we want to ensure that the final neighborhood population is diverse, so that we get more interesting recommendations. Hence, to use the idiotypic effect in the movie recommender system mentioned previously, the pseudo-code would be amended by adding the following lines in italic:

```

Initialize Artificial Immune Systems
Encode user for whom to make predictions as antigen Ag
WHILE (Artificial Immune Systems not Full) & (More Antibodies) DO
    Add next user as an antibody Ab
    Calculate matching scores between Ab and Ag and Ab and other Abs
    WHILE (Artificial Immune Systems at full size) & (Artificial Immune
        Systems not Stabilized) DO
        Reduce Concentration of all Abs by a fixed amount
        Match each Ab against Ag and stimulate as necessary
        Match each Ab against each other Ab and execute idiotypic effect
    OD
OD
Use final set of Antibodies to produce recommendation.

```

Figure 7.2 shows the idiotypic effect using dotted arrows whereas standard stimulation is shown using black arrows. In the left diagram antibodies Ab1 and Ab3 are very similar and they would have their concentrations reduced in the “Iterate Artificial Immune Systems” stage of the algorithm above. However, in the right diagram, the four antibodies are well separated from each other as well as being close to the antigen and so would have their concentrations increased.

At each iteration of the film recommendation Artificial Immune Systems the concentration of the antibodies is changed according to Eq. (7.2). This will increase the concentration of antibodies that are similar to the antigen and can allow either the stimulation, suppression, or both, of antibody–antibody interactions to have an effect on the antibody concentration. More detailed discussion of these effects on recommendation problems is given by Cayzer and Aickelin (2002a,b).

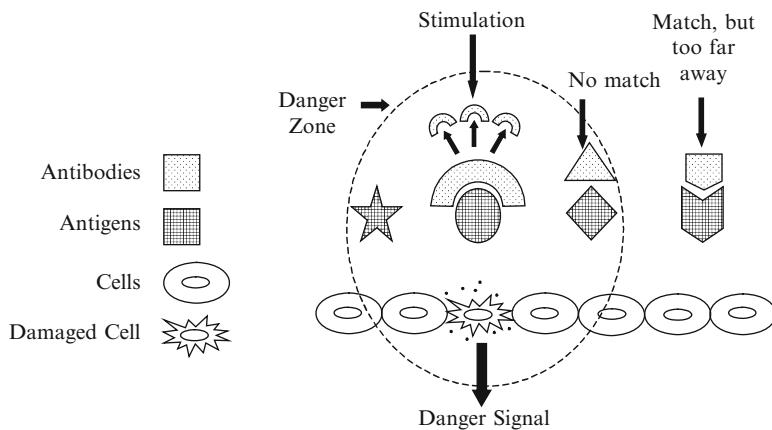


Fig. 7.3 Danger theory illustration

7.6.2 Danger Theory

Over the last decade, a new theory has become popular amongst immunologists. Called the Danger Theory, its chief advocate is [Matzinger \(1994, 2001, 2002\)](#). A number of advantages are claimed for this theory; not least that it provides a method of *grounding* the immune response. The theory is not complete, and there are some doubts about how much it actually changes behavior and/or structure. Nevertheless, the theory contains enough potentially interesting ideas to make it worth assessing its relevance to Artificial Immune Systems.

However, it is not simply a question of matching in the humoral immune system. It is fundamental that only the “correct” cells are matched as otherwise this could lead to a self-destructive autoimmune reaction. Classical immunology (Kuby 2006) stipulates that an immune response is triggered when the body encounters something nonself or foreign. It is not yet fully understood how this self–nonself discrimination is achieved, but many immunologists believe that the difference between them is learnt early in life. In particular it is thought that the maturation process plays an important role to achieve self-tolerance by eliminating those T- and B-cells that react to self. In addition, a “confirmation” signal is required; that is, for either B-cell or T (killer) cell activation, a T (helper) lymphocyte must also be activated. This dual activation is further protection against the chance of accidentally reacting to self.

Matzinger’s Danger Theory debates this point of view (for a good introduction, see [Matzinger 2002](#)). Technical overviews can be found in [Matzinger \(1994, 2001\)](#). She points out that there must be discrimination happening that goes beyond the self–nonself distinction described above. For instance:

1. There is no immune reaction to foreign bacteria in the gut or to the food we eat although both are foreign entities.

2. Conversely, some auto-reactive processes are useful, for example against self molecules expressed by stressed cells.
3. The definition of self is problematic—realistically, self is confined to the subset actually seen by the lymphocytes during maturation.
4. The human body changes over its lifetime and thus self changes as well. Therefore, the question arises whether defenses against nonself learned early in life might be autoreactive later.

Other aspects that seem to be at odds with the traditional viewpoint are autoimmune diseases and certain types of tumors that are fought by the immune system (both attacks against self) and successful transplants (no attack against nonself).

Matzinger concludes that the immune system actually discriminates “some self from some nonself”. She asserts that the Danger Theory introduces not just new labels, but a way of escaping the semantic difficulties with self and nonself, and thus provides grounding for the immune response. If we accept the Danger Theory as valid we can take care of *nonself but harmless* and of *self but harmful* invaders into our system. To see how this is possible, we have to examine the theory in more detail.

The central idea in the Danger Theory is that the immune system does not respond to nonself but to danger. Thus, just like the self–nonself theories, it fundamentally supports the need for discrimination. However, it differs in the answer to what should be responded to. Instead of responding to foreignness, the immune system reacts to danger. This theory is borne out of the observation that there is no need to attack everything that is foreign, something that seems to be supported by the counter-examples above. In this theory, danger is measured by damage to cells indicated by distress signals that are sent out when cells die an unnatural death (cell stress or lytic cell death, as opposed to programmed cell death, or apoptosis).

Figure 7.3 depicts how we might picture an immune response according to the Danger Theory ([Aickelin and Cayzer 2002](#)). A cell that is in distress sends out an alarm signal, whereupon antigens in the neighborhood are captured by antigen-presenting cells such as macrophages, which then travel to the local lymph node and present the antigens to lymphocytes. Essentially, the danger signal establishes a danger zone around itself. Thus B-cells producing antibodies that match antigens within the danger zone get stimulated and undergo the clonal expansion process. Those that do not match or are too far away do not get stimulated.

Matzinger admits that the exact nature of the danger signal is unclear. It may be a *positive* signal (for example heat shock protein release) or a *negative* signal (for example lack of synaptic contact with a dendritic antigen-presenting cell). This is where the Danger Theory shares some of the problems associated with traditional self–nonself discrimination (i.e. how to discriminate danger from non-danger). However, in this case, the signal is grounded rather than being some abstract representation of danger.

How could we use the Danger Theory in Artificial Immune Systems? The Danger Theory is not about the way Artificial Immune Systems represent data ([Aickelin and Cayzer 2002](#)). Instead, it provides ideas about which data the Artificial Immune Systems should represent and deal with. They should focus on dangerous

Table 7.2 Existing application areas of artificial immune systems

Major areas	Minor areas
Clustering/classification	Bio-informatics
Anomaly detection	Image processing
Computer security	Control
Numeric function optimization	Robotics
Combinatorial optimization	Virus detection
Learning	Web mining

(i.e. interesting) data. It could be argued that the shift from nonself to danger is merely a symbolic label change that achieves nothing. We do not believe this to be the case, since danger is a grounded signal, and nonself is (typically) a set of feature vectors with no further information about whether all or some of these features are required over time. The danger signal helps us to identify which subset of feature vectors is of interest. A suitably defined danger signal thus overcomes many of the limitations of self–nonself selection. It restricts the domain of nonself to a manageable size, removes the need to screen against all self, and deals adaptively with scenarios where self (or nonself) changes over time.

The challenge is clearly to define a suitable danger signal, a choice that might prove as critical as the choice of fitness function for an evolutionary algorithm. In addition, the physical distance in the biological system should be translated into a suitable proxy measure for similarity or causality in Artificial Immune Systems. This process is not likely to be trivial. Nevertheless, if these challenges are met, then future Artificial Immune System applications might derive considerable benefit, and new insights, from the Danger Theory, in particular Intrusion Detection Systems.

7.7 Promising Areas for Future Application

It seems intuitively obvious that Artificial Immune Systems should be most suitable for computer security problems. If the human immune system keeps our body alive and well, why can we not do the same for computers using Artificial Immune Systems? Table 7.2 shows a recent survey of application areas of Artificial Immune Systems (Hart and Timmis 2008), some which are described in the following sections.

Earlier, we outlined the traditional approach to do this. However, in order to provide viable Intrusion Detection Systems, Artificial Immune Systems must build a set of detectors that accurately match antigens. In current Artificial Immune System based Intrusion Detection Systems (Dasgupta and Gonzalez 2002; Esponda et al. 2004; Hofmeyr and Forrest 2000), both network connections and detectors are modeled as strings. Detectors are randomly created and then undergo a maturation

phase where they are presented with good, i.e. self, connections. If the detectors match any of these they are eliminated otherwise they become mature. These mature detectors start to monitor new connections during their lifetime. If these mature detectors match anything else, exceeding a certain threshold value, they become activated. This is then reported to a human operator who decides whether there is a true anomaly. If so, the detectors are promoted to memory detectors with an indefinite life span and minimum activation threshold (immunization) (Kim and Bentley 2002). An approach such as the above is known as negative selection as only those detectors (antibodies) that do not match live on Forrest et al. (1994). Earlier versions of negative-selection algorithms used a binary representation scheme; however, this scheme shows scaling problems when it is applied to real network traffic (Kim and Bentley 2001). As the systems to be protected grow larger so do self and nonself. Hence, it becomes more and more problematic to find a set of detectors that provides adequate coverage, whilst being computationally efficient. It is inefficient to map the entire self or nonself universe, particularly as they will be changing over time and only a minority of nonself is harmful, whilst some self might cause damage (e.g. internal attack). This situation is further aggravated by the fact that the labels self and nonself are often ambiguous and even with expert knowledge they are not always applied correctly (Kim and Bentley 2002).

How might this problem be overcome? One way could be to borrow ideas from the Danger Theory to provide a way of grounding the response and hence removing the necessity to map self or nonself. In our system, the correlation of low-level alerts (danger signals) will trigger a reaction. An important and recent research issue for Intrusion Detection Systems is how to find true intrusion alerts from thousands and thousands of false alerts generated (Hofmeyr and Forrest 2000). Existing Intrusion Detection Systems employ various types of sensors that monitor low-level system events. Those sensors report anomalies of network traffic patterns, unusual terminations of UNIX processes, memory usages, the attempts to access unauthorized files, etc. (Kim and Bentley 2001). Although these reports are useful signals of real intrusions, they are often mixed with false alerts and their unmanageable volume forces a security officer to ignore most alerts (Hoagland and Staniford 2002). Moreover, the low level of alerts makes it very hard for a security officer to identify advancing intrusions that usually consist of different stages of attack sequences. For instance, it is well known that computer hackers use a number of preparatory stages (raising low-level alerts) before actual hacking according to Hoagland and Staniford. Hence, the correlations between intrusion alerts from different attack stages provide more convincing attack scenarios than detecting an intrusion scenario based on low-level alerts from individual stages. Furthermore, such scenarios allow the Intrusion Detection Systems to detect intrusions early before damage becomes serious.

To correlate Intrusion Detection Systems alerts for detection of an intrusion scenario, recent studies have employed two different approaches: a probabilistic approach (Valdes and Skinner 2001) and an expert system approach (Ning et al. 2002). The probabilistic approach represents known intrusion scenarios as Bayesian networks. The nodes of Bayesian networks are Intrusion Detection Systems alerts and the posterior likelihood between nodes is updated as new alerts are collected.

The updated likelihood can lead to conclusions about a specific intrusion scenario occurring or not. The expert system approach initially builds possible intrusion scenarios by identifying low-level alerts. These alerts consist of prerequisites and consequences, and they are represented as hypergraphs. Known intrusion scenarios are detected by observing the low-level alerts at each stage, but these approaches have the following problems according to [Cuppens et al. \(2002\)](#):

1. Handling unobserved low-level alerts that comprise an intrusion scenario
2. Handling optional prerequisite actions
3. Handling intrusion scenario variations.

The common trait of these problems is that the Intrusion Detection Systems can fail to detect an intrusion when an incomplete set of alerts comprising an intrusion scenario is reported. In handling this problem, the probabilistic approach is somewhat more advantageous than the expert system approach because in theory it allows the Intrusion Detection Systems to correlate missing or mutated alerts. The current probabilistic approach builds Bayesian networks based on the similarities between selected alert features. However, these similarities alone can fail to identify a causal relationship between prerequisite actions and actual attacks if pairs of prerequisite actions and actual attacks do not appear frequently enough to be reported. Attackers often do not repeat the same actions in order to disguise their attempts. Thus, the current probabilistic approach fails to detect intrusions that do not show strong similarities between alert features but have causal relationships leading to final attacks. This limit means that such Intrusion Detection Systems fail to detect sophisticated intrusion scenarios.

We propose Artificial Immune Systems based on Danger Theory ideas that can handle the above Intrusion Detection Systems alert correlation problems. The Danger Theory explains the immune response of the human body by the interaction between antigen presenting cells and various signals. The immune response of each antigen presenting cell is determined by the generation of danger signals through cellular stress or cell death. In particular, the balance and correlation between different danger signals depending on different cell death causes would appear to be critical to the immunological outcome. The investigation of this hypothesis is the main research goal of the immunologists for this project. The wet experiments of this project focus on understanding how the antigen presenting cells react to the balance of different types of signals, and how this reaction leads to an overall immune response. Similarly, our Intrusion Detection Systems investigation will center on understanding how intrusion scenarios would be detected by reacting to the balance of various types of alerts. In the human immune system, antigen presenting cells activate according to the balance of apoptotic and necrotic cells and this activation leads to protective immune responses. Similarly, the sensors in Intrusion Detection Systems report various low-level alerts and the correlation of these alerts will lead to the construction of an intrusion scenario. A resulting product is the dendritic cell algorithm ([Greensmith 2007](#)), which is inspired by the function of the dendritic cells of the innate immune system and incorporates the principles of danger theory. An abstract model of natural dendritic cell behavior is used as the foundation of the

developed algorithm. It has been successfully applied to real-world problems, such as computer security (Al-Hammadi et al. 2008; Gu et al. 2008; Greensmith 2007), robotics (Oates et al. 2007) and fault detection of real-time embedded systems (Lay and Bate 2008).

7.8 Tricks of the Trade

Are Artificial Immune Systems suitable for pure optimization?

Depending on what is meant by optimization, the answer is probably no, in the same sense as “pure” genetic algorithms are not “function optimizers”. One has to keep in mind that although the immune system is about matching and survival, it is really a team effort where multiple solutions are produced all the time that together provide the answer. Hence, in our opinion Artificial Immune Systems are probably more suited as an optimizer where multiple solutions are of benefit, either directly, e.g. because the problem has multiple objectives, or indirectly, e.g. when a neighborhood of solutions is produced that is then used to generate the desired outcome. However, Artificial Immune Systems can be made into more focused optimizers by adding hill climbing or other functions that exploit local or problem-specific knowledge, similar to the idea of augmenting genetic algorithms to memetic algorithms.

What problems are Artificial Immune Systems most suitable for?

As mentioned in the previous paragraph, we believe that although using Artificial Immune Systems for pure optimization, e.g. the traveling salesman problem or job shop scheduling, can be made to work, this is probably missing the point. Artificial Immune Systems are powerful when a population of solution is essential either during the search or as an outcome. Furthermore, the problem has to have some concept of “matching”. Finally, because at their heart Artificial Immune Systems are evolutionary algorithms, they are more suitable for problems that change over time and need to be solved again and again, rather than one-off optimizations. Hence, the evidence seems to point to data mining in its wider meaning as the best area for Artificial Immune Systems.

How does one set the parameters?

Unfortunately, there is no short answer to this question. As with the majority of other heuristics that require parameters to operate, their setting is individual to the problem solved and universal values are not available. However, it is fair to say that along with other evolutionary algorithms Artificial Immune Systems are robust with respect to parameter values as long as they are chosen from a sensible range.

Why not use a genetic algorithm instead?

Because you may miss out on the benefits of the idiotypic network effects.

Why not use a neural network instead?

Because you may miss out on the benefits of a population of solutions and the evolutionary selection pressure and mutation.

Are Artificial Immune Systems learning classifier systems under a different name?

No, not quite. However, to our knowledge learning classifier systems are probably the most similar of the better known meta-heuristic, as they also combine some features of evolutionary algorithms and neural networks. However, these features are different. Someone who is interested in implementing Artificial Immune Systems or learning classifier systems is well advised to read about both approaches to see which one is most suited for the problem at hand.

7.9 Conclusions

The immune system is highly distributed, highly adaptive, self-organizing in nature, maintains a memory of past encounters, and has the ability to continually learn about new encounters. The Artificial Immune System is an example of a system developed around the current understanding of the immune system. It illustrates how an Artificial Immune System can capture the basic elements of the immune system and exhibit some of its chief characteristics.

Artificial Immune Systems can incorporate many properties of natural immune systems, including diversity, distributed computation, error tolerance, dynamic learning and adaptation and self-monitoring. The human immune system has motivated scientists and engineers to find powerful information processing algorithms that have solved complex engineering tasks. Artificial Immune Systems are a general framework for a distributed adaptive system and could, in principle, be applied to many domains. Artificial Immune Systems can be applied to classification problems, optimization tasks and other domains. Like many biologically inspired systems it is adaptive, distributed and autonomous. The primary advantages of the Artificial Immune Systems are that they only require positive examples, and the patterns learnt can be explicitly examined. In addition, because it is self-organizing, it does not require effort to optimize any system parameters.

To us, the attraction of the immune system is that if an adaptive pool of antibodies can produce *intelligent* behavior, can we harness the power of this computation to tackle the problem of preference matching, recommendation and intrusion detection? Our conjecture is that if the concentrations of those antibodies that provide a better match are allowed to increase over time, we should end up with a subset of good matches. However, we are not interested in optimizing, i.e. in finding the one best match. Instead, we require a set of antibodies that are a close match but which are at the same time distinct from each other for successful recommendation. This is where we propose to harness the idiotypic effects of binding antibodies to similar antibodies to encourage diversity. It is also advisable to develop hybrid Artificial Immune Systems by incorporating with other existing techniques, which may result better overall performance.

Sources of Additional Information

The following websites, books and proceedings should be an excellent starting point for those readers wishing to learn more about Artificial Immune Systems.

- Artificial Immune Systems and Their Applications, D. Dasgupta (ed), Springer, 1999.
- Artificial Immune Systems: A New Computational Intelligence Approach, L. de Castro, J. Timmis, Springer, 2002.
- Immunocomputing: Principles and Applications by Tarakanov et al., Springer, 2003.
- Proceedings of the International Conference on Artificial Immune Systems (ICARIS), Springer, 2003–2010.
- In Silico Immunology, D. R. Flower and J. and Timmis, Springer, 2007.
- Artificial Immune Systems Forum Webpage: <http://www.artificial-immune-systems.org/artist.htm>
- Artificial Immune Systems Bibliography: http://issr.cs.memphis.edu/ArtificialImmuneSystems/ArtificialImmuneSystems_bibliography.pdf

References

- Aickelin U, Cayzer S (2002) The danger theory and its application to artificial immune systems. Research Report HPL-2002-244
- Al-Hammadi Y, Aickelin U, Greensmith, J (2008) DCA for Bot detection. In: Proceedings of the IEEE WCCI, Hong Kong, pp 1807–1816
- Amazon (2003) Recommendations. <http://www.amazon.com/>
- Cayzer S, Aickelin U (2002a) A recommender system based on the immune network. In: Proceedings of the CEC 2002, Honolulu, pp 807–813
- Cayzer S, Aickelin U (2002b) On the effects of idiotypic interactions for recommendation communities in artificial immune systems. In: Proceedings of the 1st international conference on artificial immune systems, Canterbury, pp 154–160
- Cuppens F et al (2002) Correlation in an intrusion process. In: SECI 2002, TUNIS, TUNISIA
- Dasgupta, D (ed) (1999) Artificial immune systems and their applications. Springer, Berlin
- Dasgupta D, Gonzalez F (2002) An immunity-based technique to characterize intrusions in computer networks. IEEE Trans Evol Comput 6:1081–1088
- De Castro LN, Van Zuben FJ (1999) Artificial Immune Systems: Part 1 – basic theory and applications, Technical Report 1
- De Castro L N, Von Zuben FJ (2001) aiNet: An Artificial Immune Network for Data Analysis, pp. 231–259. Idea Group Publishing
- De Castro LN, Von Zuben FJ (2002) Learning and optimization using the clonal selection principle. EEE Trans Evol Comput 6:239–251

- Esponda F, Forrest S, Helman P (2004) A formal framework for positive and negative detection. *IEEE Trans Syst Man Cybern* 34:357–373
- Farmer JD, Packard NH, Perelson AS (1986) The immune system, adaptation, and machine learning. *Physica* 22:187–204
- Forrest S, Perelson, AS, Allen L, Cherukuri R (1994) Self–nonself discrimination in a computer. In: Proceedings of the IEEE symposium on research in security and privacy, Oakland, CA, USA, pp 202–212
- Goldsby R, Kindt T, Osborne B (2006) Kuby Immunology: International Edition, 6th edition, W. H. Freeman, San Francisco
- Greensmith J (2007) The dendritic cell algorithm. PhD thesis, University of Nottingham
- Gu F, Greensmith J, Aickelin U (2008) Further exploration of the dendritic cell algorithm: antigen multiplier and moving windows. In: Proceedings of the ICARIS, Phuket, pp 142–153
- Gu F, Greensmith J, Aickelin U (2009) Integrating real-time analysis with the dendritic cell algorithm through segmentation. In: GECCO 2009, Montreal, pp 1203–1210
- Hart E, Timmis J (2008) Application areas of AIS: the past, the present and the future. *Appl Soft Comput* 8:191–201
- Hightower RR, Forrest S, Perelson AS (1995) The evolution of emergent organization in immune system gene libraries. In: Proceedings of the 6th Conference on genetic algorithms, Pittsburgh, pp 344–350
- Hoagland J, Staniford S (2002) Viewing intrusion detection systems alerts: lessons from snortsnarf. <http://www.silicondefense.com/software/snortsnarf>
- Hofmeyr S, Forrest S (2000) Architecture for an artificial immune system. *Evol Comput* 7:1289–1296
- Jerne NK (1973) Towards a network theory of the immune system. *Ann Immunol* 125:373–389
- Kim J, Bentley P (2001) Evaluating negative selection in an artificial immune systems for network intrusion detection. In: GECCO 2001, San Francisco, pp 1330–1337
- Kim J, Bentley P (2002) Towards an artificial immune systems for network intrusion detection: an investigation of dynamic clonal selection. In: The Congress on Evolutionary Computation 2002, Honolulu, pp 1015–1020
- Kim J, Bentley P, Aickelin U, Greensmith J, Tedesco G, Twycross J (2007) Immune system approaches to intrusion detection—a review. *Nat Comput* 6:413–466
- Kindt T, Osborne B, Goldsby R (2006) Kuby immunology: international, 6th edn. W. H. Freeman, San Francisco
- Lay N, Bate I (2008) Improving the reliability of real-time embedded systems using innate immune techniques. *Evol Intell* 1:113–132
- Matzinger P (1994) Tolerance, danger and the extended family. *Ann Rev Immunol* 12:991–1045
- Matzinger P (2001) The danger model in its historical context. *Scand J Immunol* 54:4–9

- Matzinger P (2002) The danger model: a renewed sense of self. *Science* 296:301–305
- Ning P, Cui Y, Reeves S (2002) Constructing attack scenarios through correlation of intrusion alerts. In: Proceedings of the 9th ACM conference on computer and communications security, Washington, DC, pp 245–254
- Oates B, Greensmith J, Aickelin U, Garibaldi J, Kendall G (2007) The application of a dendritic cell algorithm to a robotic classifier. In: Proceedings of the ICARIS, Santos, Brazil, pp 204–215
- Perelson AS, Weisbuch G (1997) Immunology for physicists. *Rev Mod Phys* 69:1219–1267
- Resnick P, Varian HR (1997) Recommender systems. *Commun ACM* 40:56–58
- Valdes A, Skinner K (2001) Probabilistic alert correlation. In: Proceedings of the RAID 2001, Davis, pp 54–68

Chapter 8

Swarm Intelligence

Daniel Merkle and Martin Middendorf

8.1 Introduction

The complex and often coordinated behavior of swarms fascinates not only biologists but also computer scientists. Bird flocking and fish schooling are impressive examples of coordinated behavior that emerges without central control. Social insect colonies show complex problem-solving skills arising from the actions and interactions of nonsophisticated individuals.

Swarm intelligence is a field of computer science that designs and studies efficient computational methods for solving problems in a way that is inspired by the behavior of real swarms or insect colonies (e.g. see Bonabeau et al. 1999; Kennedy et al. 2001). Principles of self-organization and local or indirect communication are important for understanding the complex collective behavior (Sumpter 2009). Examples where insights into the behavior of natural swarms has influenced the design of algorithms and systems in computer science are:

- Models for the division of labor between members of an ant colony were used to regulate the joint work of robots and collective transport of ants has inspired the design of controllers of robots for doing coordinated work (e.g. Labella et al. 2006);
- Brood sorting behavior of ants motivated several clustering and sorting algorithms (e.g. Handl and Meyer 2002; Lumer and Faieta 1994);

D. Merkle

Department of Mathematics and Computer Science, University of South Denmark,
Odense, Denmark

e-mail: daniel@imada.sdu.dk

M. Middendorf (✉)

Department of Computer Science, University of Leipzig, Leipzig, Germany
e-mail: middendorf@informatik.uni-leipzig.de

- The synchronized flashing behavior observed in some species of fireflies has inspired algorithms for detecting non-operational robots in a swarm robotic system ([Christensen et al. 2009](#)).

In this chapter we focus on swarm intelligence methods for solving optimization and search problems. The two main areas of swarm intelligence that are relevant for such problems are ant colony optimization (ACO) and particle swarm optimization (PSO). A third emerging area is honey bee optimization (HBO).

ACO is a metaheuristic for solving combinatorial optimization problems. It is inspired by the way real ants find shortest paths from their nest to food sources. An essential aspect thereby is the indirect communication of the ants via pheromone, i.e. a chemical substance which is released into the environment and that influences the behavior or development of other individuals of the same species. Ants mark their paths to the food sources by laying trail pheromone along their way. The pheromone traces can be smelled by other ants and lead them to the food source.

PSO is a metaheuristic that is mainly used for finding maximum or minimum values of a function. PSO is inspired by the behavior of swarms of fishes or flocks of birds to find a good food place. The coordination of movements of the individuals in the swarm is the central aspect that inspires PSO.

HBO denotes a class of algorithms that are inspired by the collective behavior of honey bees. In particular, bee's mating behavior and related genetic principles, bee's foraging behavior, and their collective nest site selection behavior have been utilized for algorithm design. Depending on their type HBO algorithms are used for combinatorial optimization or function optimization. HBO is not covered in this tutorial. More information on HBO can be found in the overview articles by [Diwold et al. \(2011\)](#) and [Karaboga and Akay \(2009\)](#).

8.2 Ant Colony Optimization

A famous biological experiment called the double-bridge experiment was the inspiring source for the first ACO algorithm ([Dorigo 1992; Dorigo et al. 1991](#)). The double-bridge experiment ([Deneubourg et al. 1990; Goss et al. 1989](#)) was designed to investigate the pheromone trail laying and following behavior of the Argentine ant *Iridomyrmex humilis*. In the experiment a double bridge with two branches of different lengths connected the nest of this species with a food source (see Fig. 8.1). The long branch of the bridge was twice as long as the shorter branch. In most runs of this experiment it was found that after a few minutes nearly all ants use the shorter branch. This is interesting because Argentine ants cannot see very well. The explanation of this behavior has to do with the fact that the ants lay pheromone along their path. It is likely that ants which randomly choose the shorter branch arrive earlier at the food source. When they go back to the nest they smell some pheromone on the shorter branch and therefore prefer this branch. The pheromone on the shorter branch will accumulate faster than on the longer branch so that after some time the concentration of pheromone on the former is much higher and nearly all ants take the shorter branch. Similar to the experiment with branches

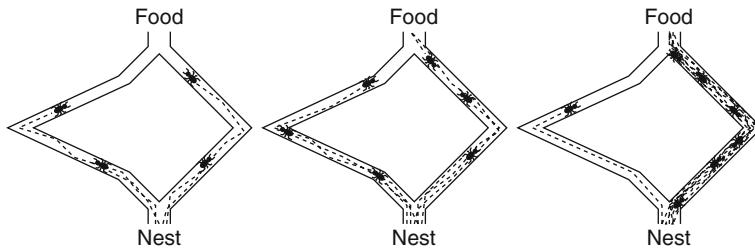


Fig. 8.1 Double-bridge experiment

of different lengths, when both branches have the same length, after some minutes nearly all ants use the same branch. But in several repetitions it is a random process which of the two branches will be chosen. The explanation is that when one branch has got a slightly higher pheromone concentration due to random fluctuations this branch will be preferred by the ants so that the difference in pheromone concentration will increase and after some time all ants take this branch.

Inspired by this experiment Dorigo and colleagues designed an algorithm for solving the traveling salesperson problem (TSP) (Dorigo 1992; Dorigo et al. 1991) and initiated the field of ACO. In recent years this field of research has become quite rich so that ACO algorithms have now been designed for various application problems and different types of combinatorial optimization problems including dynamic and multi-objective optimization problems. Some progress has been made in the last years on the theory of ACO algorithms (see Dorigo and Blum 2005; Gutjahr 2011 for an overview). An ACO metaheuristic has been formulated as a generic frame that contains most of the different ACO algorithms that have been proposed so far (see Dorigo and Di Caro 1999).

The idea of ACO is to let artificial ants construct solutions for a given combinatorial optimization problem. A prerequisite for designing an ACO algorithm is to have a constructive method which can be used by an ant to create different solutions through a sequence of decisions. Typically an ant constructs a solution by a sequence of probabilistic decisions where every decision extends a partial solution by adding a new solution component until a complete solution is derived. The sequence of decisions for constructing a solution can be viewed as a path through a corresponding decision graph (also called construction graph). Hence, an artificial ant that constructs a solution can be viewed as walking through the decision graph. The aim is to let the artificial ants find paths through the decision graph that correspond to good solutions. This is done in an iterative process where the good solutions found by the ants of an iteration should guide the ants of following iterations. Therefore, ants that have found good solutions are allowed to mark the edges of the corresponding path in the decision graph with artificial pheromone. This pheromone guides following ants of the next iteration so that they search near the paths to good solutions. In order that pheromone from older iterations does not influence the following iterations for too long, during an update of the pheromone values some percentage of the pheromone evaporates. Thus, an ACO algorithm is an iterative process where pheromone information is transferred from one iteration

to the next one. The process continues until some stopping criterion is met, e.g. a certain number of iterations has been done or a solution of a given quality has been found. A scheme of an ACO algorithm is given in the following:

ACO scheme:

```

Initialize pheromone values
repeat
for ant  $k \in \{1, \dots, m\}$ 
construct a solution
endfor
forall pheromone values do
decrease the value by a certain percentage {evaporation}
endfor
forall pheromone values corresponding to good solutions
do
increase the value {intensification}
endfor
until stopping criterion is met

```

We illustrate how the general ACO scheme can be applied to a broad class of optimization problems by means of three examples. In the first example a more detailed ACO scheme is described and applied to the TSP. An alternative approach is contained in the second example. The third example is an application of ACO to a scheduling problem which is used in comparison to the first example to discuss some additional aspects that have to be considered for designing ACO algorithms.

8.2.1 Example 1: Basic ACO and the TSP

The objective of ACO is to find good solutions for a given combinatorial optimization problem (Dorigo 1992; Dorigo and Di Caro 1999; Dorigo et al. 1991). For an easier description we restrict the following description to the broad class of optimization problems which have solutions that can be expressed as permutations of a set of given items. Such problems are called permutation problems and a famous example is the TSP. After a definition of the TSP we describe the elements of the ACO scheme that constitute an ACO algorithm, namely

- Pheromone information
- Solution construction
- Pheromone update: evaporation + intensification
- Stopping criterion

8.2.1.1 The TSP Problem

This problem is to find for a given set of n cities with distances d_{ij} between each pair of cities $i, j \in [1 : n]$ a shortest closed tour that visits every city exactly once.

Every such tour together with a start city can be characterized by the permutation of all cities as they are visited along the tour. Vice versa, each permutation of all cities corresponds to a valid solution, i.e. a closed tour.

8.2.1.2 Pheromone Information

An important part in the design of an ACO algorithm is to find a definition of the pheromone information so that it reflects the most relevant information for the solution construction. The pheromone information for permutation problems can usually be encoded in an $n \times n$ pheromone matrix $[\tau_{ij}]$, $i, j \in [1 : n]$. For the TSP problem pheromone value τ_{ij} expresses the desirability to assign city j after city i in the permutation. The pheromone matrix for the TSP problem is initialized so that all values τ_{ij} with $i \neq j$ are the same. Note that the values τ_{ii} are not needed because each city is selected only once.

TSP-ACO:

```

Initialize pheromone values
repeat
  for ant  $k \in \{1, \dots, m\}$  {solution construction}
     $S := \{1, \dots, n\}$  {set of selectable cities}
    choose city  $i$  with probability  $p_{0i}$ 
  repeat
    choose city  $j \in S$  with probability  $p_{ij}$ 
     $S := S - \{j\}$ 
     $i := j$ 
  until  $S = \emptyset$ 
endfor
forall  $i, j$  do
   $\tau_{ij} := (1 - \rho) \cdot \tau_{ij}$  {evaporation}
endfor
forall  $i, j$  in iteration best solution do
   $\tau_{i,j} := \tau_{ij} + \Delta$  {intensification}
endfor
until stopping criterion is met

```

8.2.1.3 Solution Construction

An iterative solution construction method that can be used by the ants is to start with a random item and then always choose the next item from the set S of selectable items that have not been selected so far until no item is left. Initially, the set of selectable items S contains all items; after each decision, the selected item is removed from S . Recall that in the case of the TSP the items are the cities. Every decision is made randomly where the probability equals the amount of pheromone relative to the sum of all pheromone values of items in the selection set S :

$$p_{ij} := \frac{\tau_{ij}}{\sum_{z \in S} \tau_{iz}} \quad \forall j \in S.$$

For most optimization problems additional problem-dependent heuristic information can be used to give the ants additional hints on which item to choose next. To each pheromone value τ_{ij} there is defined a corresponding heuristic value η_{ij} . For the TSP a suitable heuristic is to prefer a next city j that is near to the current city i , e.g. by setting $\eta_{ij} := 1/d_{ij}$. The probability distribution when using a heuristic is

$$p_{ij} := \frac{\tau_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum_{z \in S} \tau_{iz}^\alpha \cdot \eta_{iz}^\beta} \quad \forall j \in S, \quad (8.1)$$

where parameters α and β are used to determine the relative influence of pheromone values and heuristic values.

In order to more strongly exploit the pheromone information it has been proposed that the ant follows with some probability $q_0 \in (0, 1)$ the strongest trail, i.e. the edge in the decision graph with the maximal product of pheromone value and corresponding heuristic information (Dorigo and Gambardella 1997). For this case q_0 is a parameter of the algorithm and with probability q_0 an ant chooses next city j from the selectable cities in S which maximizes $\tau_{ij}^\alpha \cdot \eta_{ij}^\beta$. With probability $1 - q_0$ the next item is chosen according to the probability distribution determined by Eq. (8.1).

8.2.1.4 Pheromone Update

All m solutions that are constructed by the ants in one iteration are evaluated according to the respective objective function and the best solution π^* of the current iteration is determined. Then the pheromone matrix is updated in two steps:

1. Evaporation: All pheromone values are reduced by a fixed proportion $\rho \in (0, 1)$:

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} \quad \forall i, j \in [1 : n].$$

2. Intensification: All pheromone values corresponding to the best solution π^* are increased by an absolute amount $\Delta > 0$:

$$\tau_{i\pi^*(i)} := \tau_{i\pi^*(i)} + \Delta \quad \forall i \in [1 : n].$$

8.2.1.5 Stopping Criterion

The ACO algorithm executes a number of iterations until a specified stopping criterion has been met. The most commonly used stopping criteria are (possibly used in combination) that a predefined maximum number of iterations has been executed, a specific level of solution quality has been reached, or the best solution has not changed over a certain number of iterations.

Table 8.1 ACO variables and parameters

τ_{ij}	Pheromone value
η_{ij}	Heuristic value
m	Number of ants per iteration
\bar{m}	Number of ants per iteration allowed to increase pheromone
α	Influence of pheromone
β	Influence of heuristic
ρ	Evaporation rate
Δ	Amount of pheromone added during pheromone intensification
q_0	Probability to follow the strongest trail
π^*	Best solution in the actual iteration
π^e	Best solution found so far (elitist solution)

A good comparison of the optimization behavior of different ACO implementations for the TSP problem can be found in [Stützle and Hoos \(2000\)](#). The parameters and variables of ACO algorithms introduced in this section are summarized in Table 8.1.

8.2.2 Example 2: Population-Based ACO and TSP

In standard ACO algorithms the information that is transferred from one iteration to the next is the pheromone information—in the case of permutation problems this is the pheromone matrix. An alternative approach is population-based ACO (P-ACO) which was proposed by [Guntsch and Middendorf \(2002b\)](#). The idea of P-ACO is to transfer a small amount of only the most important information from one iteration to the next. This is done in the form of a small population of good solutions. An advantage of P-ACO is that the population of solutions makes it possible to apply operations from other metaheuristics to it, e.g. the crossover operation from genetic algorithms which builds a new solution by combining properties of two solutions that are already in the population. In this section we describe the differences between P-ACO and standard ACO for permutation problems. It has been shown that P-ACO is competitive to the state-of-the-art ACO algorithms with the advantage of finding good solution quality in a shorter computation time ([Guntsch and Middendorf 2002b; Oliveira et al. 2011](#)). A scheme of a P-ACO algorithm for the TSP is given in the following (compare with the scheme of ACO-TSP).

8.2.2.1 Information Transfer and Population Matrix

Instead of a complete pheromone matrix as in ACO, P-ACO transfers a small population P of the k best solutions that have been found in past iterations. Since each solution for a permutation problem is a permutation of n items, the population can be stored in an $n \times k$ matrix $P = [p_{ij}]$, where each column of P contains one solution.

P-ACO-TSP:

$P := \emptyset$

Initialize pheromone values

repeat

for ant $k \in \{1, \dots, m\}$ {solution construction}

$S := \{1, \dots, n\}$ {set of selectable cities}

 choose city i with probability p_{0i}

for $i = 1$ **to** n **do**

 choose city j with probability p_{ij}

$S := S - \{j\}$

$i := j$

endfor

endfor

If $|P| = k$ remove the oldest solution $\bar{\pi}$ from

the population: $P := P - \bar{\pi}$

Determine the best solution of the iteration and add it
to the population: $P := P + \pi^*$

Compute the new pheromone matrix from P

until stopping criterion is met

This matrix is called the population matrix. It contains the best solution of each of the preceding k iterations. When employing an elitism strategy, the best solution found so far in all iterations is—as in standard ACO—also always transferred to the next iteration. In that case the population matrix contains an additional column for the elitist solution.

8.2.2.2 Population Matrix Update

When the ants in an iteration have constructed their solutions the population (matrix) is updated. The best solution of the current iteration is added to P . If, afterwards, P contains $k + 1$ solutions the oldest solution is removed from P . The initial population is empty and after the first k iterations the population size remains k . Hence, for an update only one column in the population matrix has to be changed. Additionally, if elitist update is used and the best solution of the iteration is better than the elitist solution, the corresponding column is overwritten by the new solution. Note that each solution in the population has an influence on the decisions of the ants over exactly k subsequent iterations. Other schemes for deciding which solutions should enter/leave the population are discussed in [Guntsch and Middendorf \(2002a\)](#).

8.2.2.3 Construction of Pheromone Matrix

In P-ACO a pheromone matrix (τ_{ij}) is used by the ants for solution construction in the same way as in standard ACO. But differently, in P-ACO the pheromone matrix is derived in every iteration anew from the population matrix as follows.

Each pheromone value is set to an initial value $\tau_{init} > 0$ and is increased, if there are corresponding solutions in the population:

$$\tau_{ij} := \tau_{init} + \zeta_{ij} \cdot \Delta \quad (8.2)$$

with ζ_{ij} denoting the number of solutions $\pi \in P$ with $\pi(i) = j$, i.e. $\zeta_{ij} = |\{h : p_{ih} = j\}|$. Hence, in P-ACO a pheromone value is equal to one of the following possible values $\tau_{init}, \tau_{init} + \Delta, \dots, \tau_{init} + k \cdot \Delta$ (when using an elitist solution $\tau_{init} + (k+1) \cdot \Delta$ is also possible). An update of the pheromone values is done implicitly by a population update:

- A solution π entering the population, corresponds to a positive update:

$$\tau_{i\pi(i)} := \tau_{i\pi(i)} + \Delta.$$

- A solution σ leaving the population, corresponds to a negative update:

$$\tau_{i\sigma(i)} := \tau_{i\sigma(i)} - \Delta.$$

Note that a difference to the standard ACO algorithm is that no evaporation is used to reduce the pheromone values at the end of an iteration.

8.2.3 Example 3: ACO for a Scheduling Problem

In this section the ACO approach is applied to a scheduling permutation problem called the single machine total weighted tardiness problem (SMTWTP). The differences between the ACO algorithm for the SMTWTP and the TSP-ACO illuminate two important aspects for the design of ACO algorithms, namely the pheromone encoding and the pheromone evaluation (see [Blum and Sampels 2002b](#); [Merkle and Middendorf 2002, 2005](#) for more results on the importance of the pheromone model). Moreover, the proper adaptation of heuristics to be used for ACO is discussed. These aspects can be arranged as follows into the list of elements that constitute an ACO algorithm:

- Pheromone information
 - Pheromone encoding
- Solution construction
 - Pheromone evaluation
 - Adaptation of heuristics.

8.2.3.1 The SMTWTP Problem

For the SMTWTP n jobs are given that have to be scheduled onto a single machine. Every job $j \in [1 : n]$ has a due date d_j , a processing time p_j and a weight w_j . If C_j

denotes the completion time of job j in a schedule, then $L_j = C_j - d_j$ defines its lateness and $T_j = \max(0, L_j)$ its tardiness. The objective is to find a schedule that minimizes the total weighted tardiness of all jobs $\sum_{j=1}^n w_j T_j$.

8.2.3.2 Pheromone Encoding

When designing an ACO algorithm for an optimization problem it is important to encode the pheromone information in a way that is suitable for the problem. For the TSP it is relevant which cities are following each other in the permutation because the distance between the cities determines the quality of the solution. Therefore, pheromone values τ_{ij} are used to express the desirability that city j comes after i . For the SMTWTP the relative position of a job in the schedule is much more important than its direct predecessor or its direct successor in the schedule (see also [Blum and Sampels 2002a](#) for other scheduling problems). Therefore pheromone values for the SMTWTP are used differently than for the TSP. Pheromone value τ_{ij} expresses the desirability to assign item j at place i of the permutation. Thus, this pheromone matrix is of type place \times item whereas the pheromone matrix used for the TSP is of type item \times item. For SMTWTP an ant starts to decide which job is the first in the schedule and then always decides which job is on the next place. The pheromone matrix for the SMTWTP problem is initialized so that all values τ_{ij} , $i, j \in [1 : n]$ are the same.

8.2.3.3 Pheromone Evaluation

Another important aspect of ACO algorithms is how the pheromone information is used by the ants for their decisions. Real ants use trail pheromone only locally because they cannot smell it over long distances. The artificial ants in TSP-ACO also use the pheromone values locally which means that an ant at city i considers only the pheromone values τ_{ij} that lead to a possible next city $j \in S$. In principle, a local evaluation of the pheromone values is also possible for the SMTWTP (and has been used so—[Bauer et al. 1999](#)). An ant that has to decide which job is in the next place i in the permutation considers all values τ_{ij} , $j \in S$ which indicates how well the selectable jobs have performed in this place. But assume that for some selectable job $j \in S$ its highest pheromone value is τ_{lj} for an $l < i$. This indicates that for job j place l in the schedule is very good. But this also means that job j should not be placed much later than place l in order not to risk a due date violation. Therefore, even when the value τ_{ij} is small the ant should choose job l with high probability. Therefore, for SMTWTP a global pheromone evaluation rule has been proposed which is called summation evaluation because an ant that has to decide about place i of the permutation makes the selection probability for every selectable job dependent on the sum of all pheromone values for this job up to place i ([Merkle and Middendorf 2003b](#)):

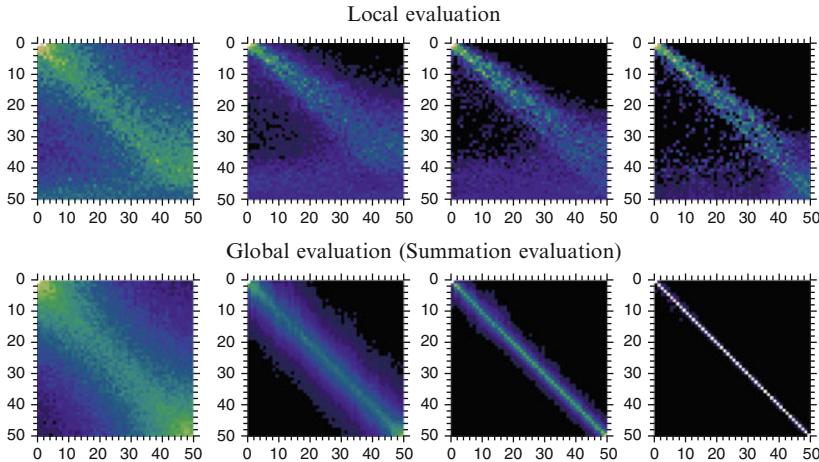


Fig. 8.2 Comparison between SMTWTP-ACO with local evaluation and summation evaluation pheromone matrices averaged over 25 runs in iterations 800, 1,500, 2,200 and 2,900 (left to right): brighter gray colors indicate higher pheromone values

$$p_{ij} = \frac{(\sum_{l=1}^i \tau_{lj})^\alpha \cdot \eta_{ij}^\beta}{\sum_{z \in S} (\sum_{l=1}^i \tau_{lz})^\alpha \cdot \eta_{iz}^\beta} \quad \forall j \in S. \quad (8.3)$$

To demonstrate the influence of the pheromone evaluation method and the change of pheromone values in ACO we show results of a very simple SMTWTP test instance (for more results and another global pheromone evaluation method see [Merkle and Middendorf 2003a](#)). It consists of 50 jobs where job $i \in [1 : 50]$ has processing time $p_i = 1$, due date $d_i = i$ and weight $w_i = 1$. Clearly, to place job i on place i is the only optimal solution with total weighted tardiness 0. Figure 8.2 shows the average change of pheromone values for ACO-SMTWTP with local and with global evaluation (no heuristic was used) for several runs with $m = 10$ ants per iteration. The figure shows clearly that for this problem summation evaluation performs much better than local evaluation. Compared to local evaluation the results of summation evaluation depicted in Fig. 8.2 show a very symmetric behavior and do not have the undesired property that some of the jobs with a small number are scheduled very late.

Table 8.2 Influence of global pheromone evaluation and adapted heuristic on solution quality for SMTWTP. Difference of total tardiness to total tardiness of best results from the literature average over 125 instances (see [Merkle and Middendorf 2003b](#) for details); Σ with summation evaluation; H with adapted heuristic (8.5)

ACO- Σ H	ACO- Σ	ACO-H	ACO
79.5	200.0	204.5	1198.6

8.2.3.4 Adaptation of Heuristics

For many (scheduling) problems there exist priority heuristics which can be used to decide which job is next when building a schedule. An example for the unweighted form of the SMTWTP is the modified due date rule (MDD), that is

$$\eta_{ij} = \frac{1}{\max\{\mathcal{T} + p_j, d_j\}}, \quad (8.4)$$

where \mathcal{T} is the total processing time of all jobs already scheduled. Observe, that the heuristic prefers jobs with a small due date from all jobs that would finish before their due date when scheduled next. Furthermore, of all those jobs that will finish after their due date, the jobs with short processing times are preferred. Some care has to be taken when using standard priority heuristics for scheduling problems in an ACO algorithm because the heuristic values might not properly reflect the relative influence they should have on the decisions of the ants. In the case of the MDD heuristic the problem occurs that the values of $\max\{\mathcal{T} + p_j, d_j\}$ become much larger—due to \mathcal{T} —when deciding about jobs to place further at the end of the schedule. As a consequence, the heuristic differences between the jobs are, in general, small at the end of the schedule. This means that the ants cannot really differentiate between the various alternatives. To avoid this effect an accordingly adapted heuristics should be used ([Merkle and Middendorf 2003b](#)), e.g.

$$\eta_{ij} = \frac{1}{\max\{\mathcal{T} + p_j, d_j\} - \mathcal{T}}. \quad (8.5)$$

To illustrate the effect of an adapted heuristic together with global pheromone evaluation some test results for benchmark problems with $n = 100$ jobs from the [OR-Library \(2012\)](#) are given. The ACO parameters used are $m = 20$ ants per generation $\alpha = 1$, $\beta = 1$, $\rho = 0.1$, $q_0 = 0.9$, and local optimization was applied to solutions found by the ants (see [Merkle and Middendorf 2003b](#) for more details). Table 8.2 compares the behavior of the algorithm using non-adapted heuristic (8.4) and local pheromone evaluation with the algorithms that use one or both of adapted heuristic (8.5) and global pheromone evaluation. The results clearly show that using an adapted heuristic (8.5) or the global pheromone evaluation improves the results significantly, and using both is best.

8.2.4 Advanced Features of ACO

In this section several variations and extension of the ACO algorithms are described that often lead to an increased search efficiency and better optimization results.

8.2.4.1 Variants of Pheromone Update

Several variations of pheromone update have been proposed in the literature:

- *Quality-dependent pheromone update.* In some ACO algorithms not only the best solution, but the $\bar{m} < m$ best solutions of each iteration are allowed to increase the pheromone values. In addition the amount of pheromone that is added can be made dependent on the quality of the solution so that the more pheromone added, the better the solution is ([Dorigo et al. 1996](#)). For the TSP this means that for shorter tours more pheromone is added.
- *Rank-based pheromone update.* Here the $\bar{m} \leq m$ best ants of an iteration are allowed to increase the pheromone. The amount of pheromone an ant is allowed to add depends on its rank within the \bar{m} best solutions and the quality of the solution ([Bullnheimer et al. 1999](#)).
- *Elitist solution pheromone update.* It can be advantageous to enforce the influence of the best solution that has been found so far over all iterations, called the elitist solution ([Dorigo et al. 1996](#)). This is done by adding pheromone during pheromone intensification also according to this solution. Several variations have been studied: e.g. to let randomly update either the iteration best or the elitist solution with increasing probability for an elitist update ([Stützle and Hoos 2000](#)) or to apply elitist pheromone update but to forget the elitist solution after several iterations by replacing it with the iteration best solution ([Merkle et al. 2002](#)).
- *Best-worst pheromone update.* This pheromone update method in addition to the standard pheromone update reduces the pheromone values according to the worst solution of an iteration provided that a pheromone value does not also correspond to the elitist solution ([Cordón et al. 2000](#)). A problem for this method is that often a decision which can lead to bad solutions can also lead to a very good solution.
- *Online step-by-step pheromone update* ([Dorigo and Gambardella 1997](#); [Dorigo et al. 1991](#)). This means that an ant adds or removes pheromone from an edge in the decision graph it has chosen immediately after the decision was made (see [Dorigo and Di Caro 1999](#) for more details). One motivation to use online step-by-step pheromone update in addition to the standard update is to remove pheromone to increase the variability in the choices of the ants during an iteration.
- *Moving-average pheromone update.* A pheromone update scheme where each constructed solution is allowed to update the pheromone ([Maniezzo 1999](#)). When the actual solution is better than the average quality of the last $k > 0$ solutions than it increases its corresponding pheromone values and otherwise it decreases them.
- *Minimum pheromone values.* The use of minimum pheromone values was proposed in order to guarantee that each possible choice always has a minimum probability to be chosen ([Stützle and Hoos 2000](#)).

8.2.4.2 Other ACO Variants

Several variants of ACO algorithms which do not use the pheromone update have also been proposed (see [Dorigo and Di Caro 1999](#) for an overview):

- *Candidate lists.* A candidate list defines for each decision a set of preferable choices ([Dorigo and Gambardella 1997](#)). For the TSP a candidate list can be defined for each city to determine the set of preferred successor cities. An ant then chooses, if possible, the next city only from cities that are in the selection set S and also in the candidate list.
- *Lower bounds.* The use of lower bounds on the cost of completing a partial solution was proposed in [Maniezzo \(1999\)](#). The lower bounds give additional heuristic information about the possible choices.
- *Lookahead.* A lookahead strategy was proposed by [Michels and Middendorf \(1999\)](#) where for each possible choice of an ant the maximum $\tau_{ij}^\alpha \cdot \eta_{ij}^\beta$ value that would result from this choice is evaluated and taken into account when actually making a decision.
- *Stagnation recovery.* For longer runs of an ACO algorithm there is the danger that after some time the search concentrates too much on a small search region. Several authors have proposed methods for modification of the pheromone information to counteract such stagnation behavior of ACO algorithms. When stagnation is detected the approach of [Gambardella et al. \(1999\)](#) is to reset all elements of the pheromone matrix to their initial values. [Stützle and Hoos \(1997\)](#) suggested increasing the pheromone values proportionately to their difference to the maximum pheromone value.
- *Changing α, β values.* [Merkle et al. \(2002\)](#) proposed reducing the value of β during a run to increase the influence of the pheromone at later stages of the algorithm. See item “Stagnation recovery” (above) for changing α values.
- *Repelling pheromone.* Some experiments with pheromone that allow the ants to avoid choosing an edge have been performed by [Kawamura et al. \(2000\)](#) and [Montgomery and Randall \(2002\)](#) in order to enforce ants (or different colonies of ants) to search in different regions of the search space.
- *Moving direction.* The use of ants that “move in different directions” can improve the optimization behavior ([Michels and Middendorf 1999](#)). For a permutation problem this could mean that some ants decide first which item is in place one of the permutation and other ants decide first which item is in the last place. In general, it can be said that the ants should make important decisions early ([Merkle and Middendorf 2005](#)).
- *Local improvement of solutions.* The use of local optimization strategies to improve the solutions that have been found by the ants has been applied so successfully (e.g. [Dorigo and Gambardella 1997; Stützle et al. 2000](#)) that most state-of-the-art ACO algorithms use local improvement strategies. Two variants of the use of local improvement strategies exist: (i) to determine how much pheromone is updated for a solution, the quality or rank of the solution is computed after the local improvement has been applied but the actual pheromone

update is done according to the original solution before the local improvement, (ii) the same as (i) but the pheromone update is done according to the solution after local improvement.

8.2.5 Promising Areas for Future Applications of ACO

An important area of research for practical applications of ACO is to create hybrid algorithms that combine properties of ACO with other search heuristics. Other promising fields like multiobjective optimization, dynamic and probabilistic optimization, hybrid algorithms, parallel and hardware algorithms, and theoretical aspects cannot be covered in this introductory tutorial.

8.2.5.1 Hybrid ACO

The aim of hybrid search heuristics is to combine the advantages of different types of search heuristics. In recent years hybrids between ACO and the following metaheuristics have been designed: genetic algorithms (GAs), particle swarm optimization (PSO), simulated annealing (SA), scatter search, path relinking, greedy randomized adaptive local search procedure (GRASP), tabu search, and others. The *International Workshop on Hybrid Metaheuristics* is particularly devoted to the study of this research field (see [Blesa et al. 2009](#) for the latest proceedings).

8.3 Particle Swarm Optimization

The roots of the metaheuristic that is described in this section lay in computing models that have been created by scientists in the last two decades to simulate bird flocking and fish schooling. The coordinated search for food which let a swarm of birds land at a certain place where food can be found was modeled with simple rules for information sharing between the individuals of the swarm. These studies inspired Kennedy and Eberhart to develop a method for function optimization that they called particle swarm optimization ([Kennedy and Eberhart 1995](#)). A PSO algorithm maintains a population of particles (the swarm), where each particle represents a location in a multidimensional search space (also called problem space). The particles start at random locations and search for the minimum (or maximum) of a given objective function by moving through the search space. The analogy to reality (in the case of a search for a maximum) is that the function measures the quality or amount of the food at each place and the particle swarm searches for the place with the best or most food. The movements of a particle depend only on its velocity and the locations where good solutions have already been found by the particle itself or other (neighbored) particles in the swarm. This is again in analogy to

bird flocking where each individual makes its decisions based on cognitive aspects (modeled by the influence of good solutions found by the particle itself) and social aspects (modeled by the influence of good solutions found by other particles). Note that, unlike many deterministic methods for continuous function optimization, PSO uses no gradient information.

In a typical PSO algorithm each particle keeps track of the coordinates in the search space which are associated with the best solution it has found so far. The corresponding value of the objective function (fitness value) is also stored. Another “best” value that is tracked by each particle is the best value obtained so far by any particle in its topological neighborhood. When a particle takes the whole population as its neighbors, the best value is a global best. At each iteration of the PSO algorithm the velocity of each particle is changed towards the personal and global best (or neighborhood best) locations. But also some random component is incorporated into the velocity update. A scheme for a PSO algorithm is given below.

PSO scheme:

```

Initialize location and velocity of each particle
repeat
  for each particle
    evaluate objective function  $f$  at the particles location
  endfor
  for each particle
    update the personal best position
  endfor
  update the global best position
  for each particle
    update the velocity
    compute the new location of the particle
  endfor
until stopping criterion is met

```

PSO has become a very active field of research with several hundreds of publications per year. The main use of PSO is as function optimizer that is often embedded into a more complex application context. An extensive overview on different fields of applications is given in [Poli \(2008\)](#). Most applications of PSO are in the fields of image and video analysis, control, distribution networks, power systems and plants ([Alrashidi and El-Hawary 2009; del Valle et al. 2008](#)), electronics and electromagnetics, design signal processing ([Merkle and Middendorf 2008](#)), biomedicine, communication networks, data mining, fuzzy systems and neural networks. Some works have also been done to apply PSO to discrete problems, e.g. scheduling problems.

In the following we describe in more detail how the PSO scheme can be applied to optimization problems. The first example considers the typical use of PSO for continuous optimization. A subset problem is addressed in the second example to illustrate how PSO can be applied to other types of optimization problems.

8.3.1 Example 1: Basic PSO and Continuous Function Optimization

In order to describe the PSO algorithm for function optimization we need some notation. Let f be a given objective function over a D -dimensional problem space. The location of a particle $i \in \{1, \dots, m\}$ is represented by a vector $\mathbf{x}_i = (x_{i1}, \dots, x_{iD})$ and the velocity of the particle by the vector $\mathbf{v}_i = (v_{i1}, \dots, v_{iD})$. Let l_d and u_d be lower and upper bounds for the particles coordinates in the d th dimension, $d \in [1 : D]$. The best previous position of a particle is recorded as $\mathbf{p}_i = (p_{i1}, \dots, p_{iD})$ and is called $pBest$. The index of the particle with the so far best found position in the swarm is denoted by g and \mathbf{p}_g is called $gBest$.

At each iteration of a PSO algorithm after the evaluation of function f the personal best position of each particle i is updated, i.e. if $f(\mathbf{x}_i) < f(\mathbf{p}_i)$ then set $\mathbf{p}_i = \mathbf{x}_i$. If $f(\mathbf{p}_i) < f(\mathbf{p}_g)$ then i becomes the new global best solution, i.e. set $g = i$. Then the new velocity of each particle i is determined during the update of velocity in every dimension $d \in [1 : D]$ as follows:

$$v_{id} = w \cdot v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (p_{gd} - x_{id}), \quad (8.6)$$

where

- Parameter w is called the *inertia weight*, it determines the influence of the old velocity; the higher the value of w the more the individuals tend to search in new areas; typical values for w are slightly smaller than 1.0;
- c_1 and c_2 are the *acceleration coefficients*, which are also called the cognitive and the social parameter respectively, because they are used to determine the influence of the local best position and the global best position respectively; typical values are $c_1 = c_2 = 2$;
- r_1 and r_2 are random values uniformly drawn from $[0, 1]$.

After velocity update the new position of particle i is determined by

$$x_{id} = x_{id} + v_{id}.$$

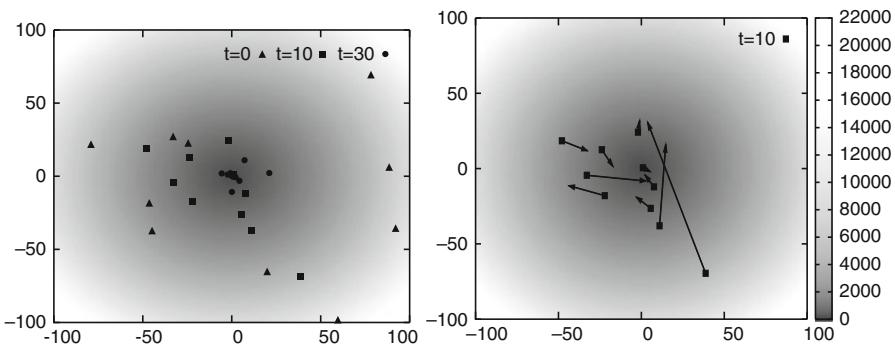
If there is a maximum range for the location in dimension d , i.e. $x_g \in [l_d, u_d]$, then the particle is reflected.

The behavior of PSO algorithms is usually studied and compared on a set of standard test functions. Examples of the most prominent test functions are given in Table 8.3. These functions represent different types of functions, e.g. the variables in Sphere and Rastrigin are uncorrelated which is not the case for the other functions in the table. Most of these functions are typically used for 10–100 dimensions.

As an example we consider a test run of the standard PSO with a swarm of size $m = 10$ on the two-dimensional Sphere function (the PSO parameters used are $w = 0.729$, $c_1 = c_2 = 1.494$). It can be seen from Fig. 8.3 (left) that the swarm proceeds from initial random positions at iteration $t = 0$ towards the single minimum value of the Sphere function. The velocity vectors of the particles at iteration $t = 10$ are shown in Fig. 8.3 (right).

Table 8.3 Test functions

Sphere
$f_1(x) = \sum_{i=1}^D x_i^2$
Rastrigin
$f_2(x) = \sum_{i=1}^D (x_i^2 - 10 \cos(2\pi x_i) + 10)$
Rosenbrock
$f_3(x) = \sum_{i=1}^{D-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$
Schaffer's f6
$f_4(x) = 0.5 - \frac{(\sin \sqrt{x_1^2 + x_2^2})^2 - 0.5}{(1 + 0.001(x_1^2 + x_2^2))^2}$
Griewank
$f_5(x) = \frac{1}{4000} \sum_{i=1}^D x_i^2 - \prod_{i=1}^D \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$

**Fig. 8.3** Swarm on the two-dimensional Sphere function; particle positions at iterations $t \in \{0, 10, 30\}$ (left); particle positions and velocity vectors at iteration $t = 10$ (right)

8.3.2 Example 2: Discrete Binary PSO for Subset Problems

Subset problems are a broad class of optimization problems where the aim is to find a good subset of a given set of items. For many practical problems additional restrictions will be given so that not all subsets of the given set are valid. A difference to many permutation problems like the TSP is that subset problems allow solutions of different sizes. As an example subset problem we consider a problem from the financial sector where the earnings of a company have to be forecast. The forecast is based on financial ratios that are generated from the companies results and other economic indicators from the last quarters. We assume that a forecast method is given that computes for each financial ratio a forecast and the final forecast is the average of the forecasts for all given values. Since many different financial ratios are in use, e.g. the book value per share or the total growth rate of a company, the problem is to select a not-too-large subset of these so that the forecast method gives good forecasts when applied to the selected financial ratios.

To solve a subset problem with PSO a particle can be encoded by a D -dimensional vector where $D = |M|$ equals the size of the given set M (in the example M is the

set of all considered financial ratios). Each dimension represents one binary bit that determines whether the corresponding item respectively the corresponding financial ratio is selected to be member of the subset. The crucial part in the design of the PSO algorithm is to connect the continuous movement of the particles to the discrete solution space.

In the so-called discrete binary PSO algorithm this is done as follows (Kennedy and Eberhart 1997). Similar as for the continuous PSO, the position of a particle corresponds to a solution and the velocity has an influence on the new position. But how the position is computed is different. Since the solution space is discrete and a particle should not stay at the same place, a random component is used in the computation of the new position. The idea is to let a high velocity in one dimension give a high probability that the corresponding bit of the position vector is one.

Formally, the velocity of a particle is determined exactly as in Eq. (8.6). In order to determine the probabilities for the computation of the position vector a function is used that maps a velocity value onto the interval $[0, 1]$. A function which is often used is $\text{sig}(v_{id}) = 1/(1 + \exp(-v_{id}))$. To determine the d th bit of the position vector of particle i a random number r_{id} is drawn from the interval $[0, 1]$ and the d th bit is set to one if $r_{id} < \text{sig}(v_{id})$ and otherwise it is set to zero.

The results of a comparative study between the discrete binary PSO and a GA for the financial ratio selection problem are presented in Ko and Lin (2004). It was shown for a problem of dimension $D = 64$ that PSO is faster and gives better results than the GA.

8.3.3 Advanced Features of PSO

Many variations of the velocity update in PSO and extensions of the standard PSO for a better control of the behavior of the swarm have been proposed in the literature (see Sedighizadeh and Masehian 2009 for a taxonomy of the different PSO variants). Only some extensions can reviewed in this section.

- *Adaptive PSO.* In Clerc (2002) a version of PSO was proposed where most values of the algorithms parameters are adapted automatically at run time. One example parameter is the swarm size that varied during execution. A particle is removed when it is the worst (with respect to the best solution found so far) of a neighborhood of particles and the best particle in its neighborhood has improved significantly since its creation. Other rules have been implemented for creating new particles. A PSO that combines ideas of various other adaptive PSOs and has several layers of adaptation has been proposed by Ritscher et al. (2010).
- *Neighborhood best velocity update.* Several PSO algorithms establish a neighborhood relation between particles. In that case, instead of using the global best position $gBest$ for velocity update for each particle the best position of the particles in its neighborhood is used. This position is called neighborhood best and is denoted by $lBest$. A PSO variant where all particles in the neighborhood of a particle have an influence on its velocity was proposed by Kennedy and Mendes (2003).

The following formula describes such an-all-neighborhood-velocity-update of particle i with neighborhood N_i in dimension $d \in [1 : D]$ (note that a particle is included in its own neighborhood):

$$v_{id} = w \cdot v_{id} + \sum_{j \in N_i} \frac{c \cdot r_j \cdot (p_{jd} - x_{id})}{|N_i|}. \quad (8.7)$$

A Hierarchical PSO where particles are particle within a hierarchy and neighborhood is defined by the place within the hierarchy has been proposed in [Janson and Middendorf \(2005\)](#).

- *Simplified PSO.* A PSO which does not use the personal best positions for velocity update was suggested by [Kennedy \(1997\)](#) and has been called *social only* PSO. A similar simplified PSO was investigated by [Pedersen and Chipperfield \(2010\)](#) and was shown to perform well for optimizing artificial neural network problems.
- *Maximum velocity.* A parameter v_{\max} is introduced for some PSO algorithms to restrict the size of the elements v_{id} of the velocity vector so that $v_{id} \in [-v_{\max}, v_{\max}]$. Hence, when the velocity of a particle becomes larger than v_{\max} during velocity update it is set to v_{\max} . A typical range for values of v_{\max} is $[0.1 \cdot x_{\max}, 1.0 \cdot x_{\max}]$. Observe that such values for v_{\max} do not restrict the possible locations of a particle to $[-x_{\max}, x_{\max}]$.
- *Queen particle.* The addition of a queen particle, which is always located at the swarm's actual gravity center was proposed in [Clerc \(1999\)](#). Since the gravity center of the swarm might often be near a possible optimum, it is reasonable to evaluate the objective function at this location. Experimental results have shown that it depends on the type of function whether the introduction of a queen particle is advantageous.
- *Convergence enforcement.* Several authors have considered the problem of how to improve the rate of convergence of PSO.
 - A constriction coefficient K was introduced in [Clerc and Kennedy \(2002\)](#) to reduce undesirable explosive feedback effects where the average distance between the particles grows during an execution. With the constriction coefficient as computed in [Kennedy and Eberhart \(1999\)](#) and [Kennedy et al. \(2001\)](#) the formula for velocity update becomes

$$v_{id} = K \cdot (v_{id} + c_1 \cdot r_1 \cdot (p_{id} - x_{id}) + c_2 \cdot r_2 \cdot (p_{gd} - x_{id})).$$

Note that the constriction factor is just another way of choosing parameters w , c_1 and c_2 . It can be shown that the swarm converges when parameter K is determined as ([Kennedy and Eberhart 1999](#))

$$K = \frac{2}{|2 - c - \sqrt{c^2 - 4c}|}$$

with $c = c_1 + c_2$, $c > 4$.

- Parameter w can be decreased over time during execution to diminish the diversity of the swarm and to reach faster a state of equilibrium. A linear decrease of w from a maximum value w_{\max} to a minimum value w_{\min} is used by several authors (e.g. Kennedy et al. 2001). Typical values are $w_{\max} = 0.9$ and $w_{\min} = 0.4$.
- In Vesterstrøm et al. (2002) the concept of division of labor and specialization was applied to PSO. Specialization to a task in this approach means for a particle to search near the global best position $gBest$. A particle that has not found a better solution for a longer time span is replaced by $gBest$ in order to start searching around it. To prevent too many particles searching around $gBest$ it was suggested to use a maximum number of particles that can switch to $gBest$. Note that a similar approach to let particles that have not found good solutions jump to the place of good particles is realized in the hybrid PSO, described below.
- *Controlling diversity.* To prevent the swarm from converging too early to a small area so that the particles become too similar, some methods have been proposed to maintain the diversity of the swarm. A common measure for the diversity of the swarm S in PSO is the “distance to average point”:

$$\text{diversity}(S) := \frac{1}{|S|} \cdot \sum_{i=1}^{|S|} \sqrt{\sum_{j=1}^D (p_{ij} - \bar{p}_j)^2}$$

where \bar{p} is the average vector of all $pBest$ vectors \mathbf{p}_i . In order to make the diversity measure independent of the range of the search space some authors use the measure $\text{diversity}(S)/|L|$ where $|L|$ is the length of the longest diagonal in the search space. Some methods to keep the diversity of the swarm high enough are described in the following.

- Xie et al. (2002) proposed adding an additional random element to the movement of the particles. In the new algorithm called dissipative PSO (DPSO), immediately after velocity update and determination of the new position of the particles the following computations are done to introduce additional “chaos” to the system:

```

if rand() <  $c_v$  then
     $v_{id} = \text{rand}() \cdot v_{\max,d}$  {chaos for velocity}
if rand() <  $c_l$  then
     $x_{id} = \text{rand}(l_d, u_d)$  {chaos for location}

```

where $c_v, c_l \in [0, 1]$ are parameters which control the probability to add chaos, $\text{rand}(a, b)$ is random number that is uniformly distributed in (a, b) ($\text{rand}()$ is a shortcut for $\text{rand}(0, 1)$) and l_d, u_d are lower and upper bounds for the location in dimension d . Observe, that if $\text{rand}() < c_l$ the d th dimension of the new position is a random location within the search area.

- The idea of using particles that have a spatial extension in the search space was established in [Krink et al. \(2002\)](#) to hinder particles from coming too close to each other and forming too dense clusters. In this variation of PSO particles that come too close to each other bounce away. Preliminary experimental results show that bouncing can be advantageous for complex objective functions. For objective functions with a single optimum, clustering is not a problem and bouncing is not advantageous. A similar approach to hinder the swarm from collapsing uses an analogy to electrostatic energy. In this approach, so-called charged particles experience a repulsive force when they come too close to each other ([Blackwell and Bentley 2002](#)). Swarms with different ratios of charged particles have been studied.
- A strategy to explicitly control the diversity is to have two different phases of the PSO algorithm that can increase (repulsion phase) or reduce (attraction phase) the diversity of the swarm ([Riget and Vesterstrøm 2002](#)). Two threshold values have been introduced that are used to determine when an exchange between the two phases should take place. When the diversity becomes lower than the threshold d_{low} the algorithm switches to the repulsion phase and when the diversity becomes larger than threshold $d_{\text{high}} > d_{\text{low}}$ the algorithm changes to the attraction phase. The only thing that happens when the phase of the algorithm changes is that every velocity vector is changed so that it points in the opposite direction. The authors have shown by experiments that nearly all improvements of the global best solutions were found during the attraction phases. Therefore, they propose to do no function evaluations during the repulsion phase to reduce the run time of the algorithm.
- *Stagnation recovery.* For multi-modal functions there is the danger of premature convergence of standard PSO which results in suboptimal solutions. Stagnation recovery means detecting such a situation and reacting accordingly.
 - Re-initialization of the swarm is proposed in [Clerc \(1999\)](#) when the diameter of the area that is actively searched by the swarm has become too small. The new swarm is initialized around the previous best position.

8.3.4 Promising Areas for Future Applications of PSO

Complex multimodal functions that possess multiple and possibly similarly good local optimal solutions occur in many applications. Often in such applications it is not enough to know just a single of these local optimal solutions but several or all of them are needed. Two areas of future PSO research that are relevant for optimizing such complex multimodal functions are: (i) to find additional operations on the particles that can improve the optimization efficiency and (ii) to investigate how several swarms can work cooperatively. We briefly review some works in both areas. Other interesting application and research areas like multiobjective and dynamic op-

timization, hybrid algorithms, and theoretical aspects of PSO cannot be covered in this introductory tutorial.

8.3.4.1 Operations on Particles

Several operations on particles other than velocity update have been proposed in recent years. Examples are: (i) mutation operators that change the position of a particle, (ii) interpolation operators that try to find a new good position between the positions of several other particles, or (iii) operations that try to make particles behave similarly to particles in quantum mechanics. Algorithms using the latter type of operations are called quantum PSO algorithms (QPSOs) ([Sun et al. 2004](#)). In a QPSO the state of a particle is described by a wavefunction, instead of a position and a velocity. The wavefunction determines the probability of a particle appearing at a certain position.

8.3.4.2 Cooperative Swarms

Cooperative swarms have been introduced in order to divide the work between several swarms. One motivation is that it can be very difficult for a single swarm to solve problems with large dimension D . An example is the Cooperative Swarm Optimizer (CPSO) or Split Swarm that uses a set of swarms and splits the work equally between them in the following way ([van den Bergh and Engelbrecht 2000](#)). The vector to be optimized is split across the swarms so that each swarm optimizes a different part of the vector, i.e. the swarms optimize with respect to different dimensions of the search space. Cooperation between the swarms is established in that for every evaluation of a new position of some particle its partial vector is combined with one partial vector from each of the other swarms so that the quality of the resulting position is best. Experiments with CPSO for function minimization and neural network learning have shown that it is good for problems where the dependencies between the component vectors are not too strong. Another motivation to use cooperative swarms is for solving multiobjective problems where several functions have to be optimized so that the swarms optimize with respect to different functions. A problem is then to find good methods for exchanging information about the best positions between the swarms (see for example [Parsopoulos et al. 2004](#)).

A PSO method that intends to form subswarms of particles searching for the same local minimum is proposed in [Kennedy \(2000\)](#). A standard k -means cluster method was used to divide the swarm into several clusters of individuals. For velocity update then each particle i uses the center of its cluster instead of its personal best position p_{id} (see Eq. (8.6)). Test results have shown that this velocity update modification can be advantageous, especially for multimodal functions like the Rastrigin function (see Table 8.3).

Niching techniques can also be used to promote the formation of subswarms around different local optima. Niching is a concept that is inspired by the famous

observation known from ecology that coexisting species can survive because they occupy different niches, which roughly means that they have different tasks. A niching technique for PSO that aims to find all good local minima was proposed by [Parsopoulos and Vrahatis \(2001\)](#). It uses a function *stretching* method that changes the objective function during execution as follows. Assume that a position x has been found where the objective function f to be minimized has a small value. Then f is transformed with the aim to remove local minima that are larger than $f(x)$ and a subswarm is created that searches for a local minimum near x on the transformed function. In addition, a second transformation is applied to f which increases the function values in the neighborhood of x . This function is then used by the main swarm which will be repelled from the area around x and searches for a different local minimum. Another niching approach for PSO was proposed by [Brits et al. \(2002\)](#). The niching PSO starts with particles that move according to the so-called cognition-only model where velocity update is done only according to the personal best position of an individual (i.e. $c_2 = 0$ in Eq. (8.6)). The particles then basically perform local search. When the quality of a particle has not changed significantly for several iterations it is assumed that it has reached the region of a local minimum. To search for this minimum a subswarm is formed. Several rules are used to define how other particles can enter a subswarm or how subswarms with intersecting regions are merged.

8.4 Tricks of the Trade

For newcomers to the field of swarm intelligence it is an advantage that ACO and PSO algorithms are relatively easy to implement so that one can devise one's own practical experience without too much initial effort. Often even the standard form of ACO and PSO algorithms that do not use many problem-specific features work reasonably well for different types of optimization problems. This is especially true for certain types of problems, e.g. scheduling problems in the case of ACO and continuous function optimization in case of PSO. Clearly, such an early success should not lead to the illusion that swarm intelligence is a field where good algorithms can be obtained more or less for free, because principles are used that have been inspired by successful strategies which occur in nature. The following hints might be helpful for the newcomer to get a deeper understanding of swarm intelligence:

- Read those papers where swarm intelligence methods have been applied to problems that are similar to the problem you want to solve. Not less important is the study of good papers where you can learn about specific aspects of swarm intelligence methods or where carefully designed state-of-the-art algorithms are described.
- Preferably, do not start with too complicated an algorithm that you do not understand. Critically evaluate every step of your algorithm.
- Investigate how your algorithm behaves on different types of problem instances and try to verify your explanations. Test your algorithm on benchmark instances,

if available, to make comparisons with the works of other researchers easier. Ideally, use random instances and real-world instances for the tests. Random instances have the advantage that their properties can be characterized by their generation method. A disadvantage is that they are often too artificial to reflect important characteristics of real-world problems. In addition, carefully designed artificial problem instances can sometimes help to study special aspects of the behavior of algorithms.

- Investigate how robust your algorithm is with respect to changes of the parameters (e.g. parameters α , β , and ρ for ACO and parameters w , c_2 and c_1 for PSO).
- Consider the optimization behavior of your algorithm at different numbers of iterations. Then you can find out for example whether the algorithm converges too early.

For ACO the following hints should be considered:

- It is important to use pheromone information so that the ants are guided to good solutions. Two connected aspects are important here: (i) the pheromone should be used to encode properties of a solution that are most relevant in the sense that they can be used to characterize the good solutions, and (ii) the pheromone information should be interpreted by the ants in the best possible way.
- Find a solution construction process so that the ants make important decisions early on and so that they can use a good (deterministic) heuristic. Such heuristics can be found in the literature for many problems.

For PSO the following hint should be considered:

- For function optimization it is important to understand the characteristics of the search landscape of the application functions. When there is basically a single valley in the search space, a single swarm where convergence is enforced might work. But for search landscapes with many valleys a more sophisticated approach might be necessary, where the diversity of the swarm is controlled, stagnation recovery mechanisms are introduced, or several cooperative swarm are used.

8.5 Conclusion

The field of swarm intelligence with the vision to learn from the behavior of natural swarms for the development of new methods in optimization has produced with ACO and PSO two successful metaheuristics that have found a increasing number of applications in the last few years. The basic principles of swarm intelligence methods and a selection of example applications have been explained in this tutorial. New algorithmic principles, e.g. HBO, arise for swarm intelligence and there are also a number of new application areas where swarm intelligence will play its part. One promising concept are hybrid methods where swarm intelligence algorithms work in line with other metaheuristics. Hopefully, this tutorial may be a starting point for the reader to further explore the field of swarm intelligence.

Sources Of Additional Information

- Good introductory books that cover various aspects of Swarm Intelligence are:
 (i) Bonabeau, Dorigo, and Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, 1999, Oxford University Press, (ii) Kennedy, Eberhart, and Shi, *Swarm Intelligence*, 2001, Morgan Kaufmann.
 Recent overview papers are [Blum \(2005\)](#) and [Dorigo and Blum \(2005\)](#) on ACO and [Banks et al. \(2007\)](#) and [Banks et al. \(2008\)](#) on PSO.
- The following book is the ultimate reference book for ACO: Dorigo and Stützle, *Ant Colony Optimization*, 2004, MIT Press. A good reference book on PSO which also covers other parts of Swarm Intelligence is: Engelbrecht, *Fundamentals of Computational Swarm Intelligence*, 2005, Wiley. Another recent book on PSO is: Parsopoulos and Vrahatis, *Particle Swarm Optimization and Intelligence: Advances and Applications*. 2010, Information Science Publishing (IGI Global).
- Two journals that are devoted entirely to the field of Swarm Intelligence are: *Swarm Intelligence*, Springer, and the *International Journal of Swarm Intelligence Research*, IGI Global.
- Recent special issues of journals that are devoted to Swarm Intelligence are
 - Special Issue on Swarm Intelligence Theory, *Theoretical Computer Science* 411(21), Bonabeau et al., guest editors, 2010.
 - Special Issue on Swarm Intelligence, *Natural Computing*, Bonabeau et al., guest editors, 2010.
 - Special Issue on Particle Swarm Optimization, *Swarm Intelligence* 3(4), Poli et al., guest editors, 2009.
 - Special Issue on Swarm Intelligence, *IEEE Transactions on Evolutionary Computation* 13(4), Engelbrecht et al., guest editors, 2009.
 - Special Issue on Ant Colony Optimization, *Swarm Intelligence* 3(1), Doerner et al., guest editors, 2009.
- A valuable source of recent research papers are the proceedings of the following conference series that focus on Swarm Intelligence: (i) International Conference on Ant Colony Optimization and Swarm Intelligence (ANTS), (ii) IEEE Swarm Intelligence Symposium (SIS) (the latest proceedings are [Dorigo et al. 2008](#) and [Kennedy and Shi 2009](#), respectively). Another new conference on the topic is the *International Conference on Swarm Intelligence* (ICSI).

References

- Alrashidi MR, El-Hawary ME (2009) A survey of particle swarm optimization applications in electric power systems. *IEEE Trans Evol Comput* 13:913–918
- Banks A, Vincent J, Anyakoha C (2007) A review of particle swarm optimization. Part I: background and development. *Nat Comput* 6:467–484

- Banks A, Vincent J, Anyakoha C (2008) A review of particle swarm optimization. Part II: hybridisation, combinatorial, multicriteria and constrained optimization, and indicative applications. *Nat Comput* 7:109–124
- Bauer A, Bullnheimer B, Hartl RF, Strauss C (1999) An ant colony optimization approach for the single machine total tardiness problem. In: Proceedings of the CEC 1999, Washington, DC. IEEE, Piscataway, pp 1445–1450
- Blackwell TM, Bentley PJ (2002) Dynamic search with charged swarms. In: GECCO 2002, New York. Morgan Kaufmann, San Mateo, pp 19–26
- Blesa MJ, Blum C, Di Gaspero L, Roli A, Sampels M, Schaefer A (eds) (2009) In: 6th international workshop hybrid metaheuristics, Udine. LNCS 5818. Springer, Berlin
- Blum C (2005) Ant colony optimization: introduction and recent trends. *Phys Life Rev* 2:353–373
- Blum C, Sampels M (2002a) Ant colony optimization for FOP shop scheduling: a case study on different pheromone representations. In: Proceedings of the CEC 2002, Honolulu, pp 1558–1563
- Blum C, Sampels M (2002b) When model bias is stronger than selection pressure. In: Proceedings of the PPSN VII, Granada. LNCS 2439. Springer, Berlin, pp 893–902
- Bonabeau E, Dorigo M, Theraulaz G (1999) Swarm intelligence: from natural to artificial systems. Oxford University Press, New York
- Brits R, Engelbrecht AP, van den Bergh F (2002) A niching particle swarm optimizer. In: Proceedings of the SEAL 2002, Singapore, pp 692–696
- Bullnheimer B, Hartl RF, Strauss CA (1999) New rank based version of the ant system: a computational study. *Cent Eur J Oper Res Econ* 7:25–38
- Christensen A, O’Grady R, Dorigo M (2009) From fireflies to fault tolerant swarms of robots. *IEEE Trans Evol Comput* 13:754–766
- Clerc M (1999) The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In: Proceedings of the CEC, Washington, DC. IEEE, Piscataway, pp 1951–1957
- Clerc M (2002) Think locally, act locally—a framework for adaptive particle swarm optimizers. *IEEE J Evol Comput* 3:1951–1957
- Clerc M, Kennedy J (2002) The particle swarm—explosion, stability, and convergence in a multidimensional complex space. *IEEE Trans Evol Comput* 6:58–73
- Cordón O, Fernandez I, Herrera F, Moreno L (2000) A new ACO model integrating evolutionary computation concepts: the best-worst ant system. In: Proceedings of the 2nd international workshop on ant algorithms, Brussels, pp 22–29
- del Valle Y, Venayagamoorthy GK, Mohagheghi S, Hernandez J-C, Harley RG (2008) Particle swarm optimization: basic concepts, variants and applications in power systems. *IEEE Trans Evol Comput* 12:171–195
- Deneubourg J-L, Aron S, Goss S, Pasteels JM (1990) The self-organizing exploratory pattern of the Argentine ant. *J Insect Behav* 32:159–168
- Diwold K, Beekman M, Middendorf M (2011) Honeybee optimisation. In: Panigrahi BK et al (eds) Handbook of swarm intelligence—concepts, principles and application. Springer, Berlin, pp 295–328

- Dorigo M (1992) Optimization, learning and natural algorithms (in Italian). PhD thesis, Dipartimento di Elettronica, Politecnico di Milano
- Dorigo M, Blum C (2005) Ant colony optimization theory: a survey. *Theor Comput Sci* 344:243–278
- Dorigo M, Di Caro G (1999) The ant colony optimization meta-heuristic. In: Corne D et al (eds) New ideas in optimization. McGraw-Hill, New York, pp 11–32
- Dorigo M, Gambardella LM (1997) Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Trans Evol Comput* 1:53–66
- Dorigo M, Maniezzo V, Colomi A (1991) Positive feedback as a search strategy. Technical report 91-016, Politecnico di Milano
- Dorigo M, Maniezzo V, Colomi A (1996) The ant system: optimization by a colony of cooperating agents. *IEEE Trans Syst Man Cybern B* 26:29–41
- Dorigo M, Birattari M, Blum C, Clerc M, Stützle T, Winfield AFT (eds) (2008) In: Proceedings of the ANTS 2008, Brussels. LNCS 5217. Springer, Berlin
- Gambardella LM, Taillard E, Dorigo M (1999) Ant colonies for the quadratic assignment problem. *J Oper Res Soc* 50:167–176
- Goss S, Aron S, Deneubourg JL, Pasteels JM (1989) Self-organized shortcuts in the Argentine ant. *Naturwissenschaften* 76:579–581
- Guntsch M, Middendorf M (2002a) Applying population based ACO to dynamic optimization problems. In: Proceedings of the 3rd international workshop ANTS 2002, Brussels. LNCS 2463. Springer, Berlin, pp 111–122
- Guntsch M, Middendorf M (2002b) A population based approach for ACO. In: Proceedings of the EvoWorkshops 2002 on applications of evolutionary computing, Kinsale. LNCS 2279. Springer, Berlin, pp 72–81
- Gutjahr WJ (2011) Ant colony optimization: recent developments in theoretical analysis. In: Auger A, Doerr B (eds) Theory of randomized search heuristics. World Scientific, Singapore, pp 225–254
- Handl J, Meyer B (2002) Improved ant-based clustering and sorting in a document retrieval interface. In: Merelo Guervos JJ et al (eds) Proceedings of the PPSN VII, Granada. LNCS 2439. Springer, Berlin, pp 913–923
- OR-Library (2012). <http://mscmga.ms.ic.ac.uk/jeb/orlib/wtinfo.html>
- Janson S, Middendorf M (2005) A hierarchical particle swarm optimizer and its adaptive variant. *IEEE Syst Man Cybern B* 32:1272–1282
- Karaboga D, Akay B (2009) A survey: algorithms simulating bee swarm intelligence. *Artif Intell Rev* 31:61–85
- Kawamura H, Yamamoto M, Suzuki K, Ohucke A (2000) Multiple ant colonies algorithm based on colony level interactions. *IEICE Trans Fundam* 83A:371–379
- Kennedy J (1997) The particle swarm: social adaptation of knowledge. In: Proceedings of the CEC, Indianapolis, pp 303–308
- Kennedy J (2000) Stereotyping: improving particle swarm performance with cluster analysis. In: Proceedings of the CEC, La Jolla, pp 1507–1512
- Kennedy J, Eberhart RC (1995) Particle swarm optimization. In: Proceedings of IEEE international conference on neural networks, Perth, pp 1942–1948
- Kennedy J, Eberhart RC (1997) A discrete binary version of the particle swarm algorithm. *Proc Conf Syst Man Cybern* 5:4104–4109. IEEE, Piscataway

- Kennedy J, Eberhart RC (1999) The particle swarm: social adaption in information processing systems. In: Corne D et al (eds) New ideas in optimization. McGraw-Hill, New York, pp 379–387
- Kennedy J, Mendes R (2003) Neighborhood topologies in fully-informed and best-of-neighborhood particle swarms. In: Proceedings of the IEEE international workshop on soft computing in industrial applications, New York
- Kennedy J, Shi Y (eds) (2009) In: Proceedings of the 2009 IEEE Swarm Intelligence Symposium, Nashville, IEEE
- Kennedy J, Eberhart RC, Shi Y (2001) Swarm intelligence. Morgan Kaufmann, San Francisco
- Ko P-C, Lin P-C (2004) A hybrid swarm intelligence based mechanism for earning forecast. In: Proceedings of the ICITA 2004, Harbin
- Krink T, Vesterstrøm JS, Riget J (2002) Particle swarm optimisation with spatial particle extension. In: Proceedings of the CEC 2002, Honolulu, pp 1474–1479
- Labella TH, Dorigo M, Deneubourg J-L (2006) Division of labour in a group of robots inspired by ants' foraging behaviour. ACM Trans Auton Adapt Syst 1:4–25
- Lumer ED, Faieta B (1994) Diversity and adaptation in populations of clustering ants. In: Proceedings of the SAB 1994, Brighton. MIT, Cambridge, pp 501–508
- Maniezzo V (1999) Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. Inf J Comput 11:358–369
- Merkle D, Middendorf M (2002) Ant colony optimization with the relative pheromone evaluation method. In: Proceedings of the EvoWorkshops 2001, Como. LNCS 2279. Springer, Berlin, pp 325–333
- Merkle D, Middendorf M (2003a) On the behavior of ACO algorithms: studies on simple problems. In: Resende MGC, Pinho de Sousa J (eds) Metaheuristics: computer decision-making. Kluwer, Dordrecht, pp 465–480
- Merkle D, Middendorf M (2003b) An ant algorithm with global pheromone evaluation for scheduling a single machine. Appl Intell 18:105–111
- Merkle D, Middendorf M (2005) On solving permutation scheduling problems with ant colony optimization. Int J Syst Sci 36:255–266
- Merkle D, Middendorf M (2008) Swarm intelligence and signal processing. IEEE Signal Process Mag 25:152–158
- Merkle D, Middendorf M, Schmeck H (2002) Ant colony optimization for resource-constrained project scheduling. IEEE Trans Evol Comput 6:333–346
- Michels R, Middendorf M (1999) An ant system for the shortest common supersequence problem. In: Corne D, Dorigo M, Glover F (eds) New ideas in optimization. McGraw-Hill, New York, pp 51–61
- Montgomery J, Randall M (2002) Anti-pheromone as a tool for better exploration of search space. In: Proceedings of the ANTS 2002, Brussels. LNCS 2463. Springer, Berlin, pp 100–110
- Oliveira SM, Hussin MS, Stützle T, Roli A, Dorigo M (2011) A detailed analysis of the population-based ant colony optimization algorithm for the TSP and the QAP. In: GECCO (Companion), Dublin, pp 13–14

- Parsopoulos KE, Vrahatis MN (2001) Modification of the particle swarm optimizer for locating all the global minima. In: Kurkova V et al (eds) Artificial neural networks and genetic algorithms. Springer, Berlin, pp 324–327
- Parsopoulos KE, Tasoulis DK, Vrahatis MN (2004) Multiobjective optimization using parallel vector evaluated particle swarm optimization. In: Proceedings of the IASTED international conference on artificial intelligence and applications, Innsbruck
- Pedersen MEH, Chipperfield AJ (2010) Simplifying particle swarm optimization. *Appl Soft Comput* 10:618–628
- Poli R (2008) Analysis of the publications on the applications of particle swarm optimisation. *J Artif Evol Appl* 1:1–10
- Riget J, Vesterstrøm JS (2002) A diversity-guided particle swarm optimizer—the ARPSO. Technical report no 2002-02, University of Aarhus
- Ritscher T, Helwig S, Wanka R (2010) Design and experimental evaluation of multiple adaptation layers in self-optimizing particle swarm optimization. In: Proceedings of the CEC 2010, Barcelona, pp 1–8
- Sedighizadeh D, Masehian E (2009) Particle swarm optimization methods, taxonomy and applications. *Int J Comput Theor Eng* 1:1793–8201
- Stützle T, Hoos H (1997) Improvements on the ant system: introducing MAX(MIN) ant system. In: Proceedings of the international conference on artificial neural networks and genetic algorithms. Springer, Berlin, pp 245–249
- Stützle T, Hoos H (2000) MAX-MIN ant system. *Future Gener Comput Syst* 16:889–914
- Stützle T, den Besten M, Dorigo M (2000) Ant colony optimization for the total weighted tardiness problem. In: Deb et al (eds) Proceedings of the PPSN-VI, Paris. LNCS 1917. Springer, Berlin, pp 611–620
- Sumpter DJT (2009) Collective animal behavior. Princeton University Press, Princeton
- Sun J, Feng B, Xu W (2004) Particle swarm optimization with particles having quantum behavior. In: IEEE Proceeding of the CEC, San Diego, pp 325–331
- van den Bergh F, Engelbrecht AP (2000) Cooperative learning in neural networks using particle swarm optimizers. *S Afr Comput J* 26:84–90
- Vesterstrøm JS, Riget J, Krink T (2002) Division of labor in particle swarm optimisation. In: Proceedings of the CEC 2002, Honolulu, pp 1570–1575
- Xie X-F, Zhang W-J, Yang Z-L (2002) A dissipative particle swarm optimization. In: Proceedings of the CEC 2002, Honolulu

Chapter 9

Tabu Search

Michel Gendreau and Jean-Yves Potvin

9.1 Introduction

Over the last 25 years, hundreds of papers presenting applications of tabu search, a heuristic method originally proposed by [Glover \(1986\)](#), to various combinatorial problems have appeared in the operations research literature (see for example [Glover and Laguna 1997](#); [Glover et al. 1993](#); [Jaziri 2008](#); [Pardalos and Resende 2002](#); [Rego and Alidaee 2005](#)). In several cases, the methods described provide solutions very close to optimality and are among the most effective, if not the best, to tackle the difficult problems at hand. These successes have made tabu search extremely popular among those interested in finding good solutions to the large combinatorial problems encountered in many practical settings. Several papers, book chapters, special issues and books have surveyed the rich tabu search literature (a list of some of the most important references is provided at the end). In spite of this abundant literature, there still seem to be many researchers who, while they are eager to apply tabu search to new problem settings, find it difficult to properly grasp the fundamental concepts of the method, its strengths and its limitations, and to come up with effective implementations. The purpose of this chapter is thus to focus on the fundamental concepts of tabu search. Throughout the chapter, two relatively straightforward, yet challenging and relevant, problems will be used to illustrate these concepts: the job shop scheduling problem and the capacitated plant location problem.

M. Gendreau

Département de mathématiques et de génie industriel, École Polytechnique de Montréal and CIRRELT, Montréal, Canada

J.-Y. Potvin (✉)

Département d'informatique et de recherche opérationnelle, Université de Montréal and CIRRELT, Montréal, Canada

e-mail: potvin@iro.umontreal.ca

9.2 Illustrative Problems

9.2.1 The Job-Shop Scheduling Problem

The job shop scheduling problem is one of the most studied problems in combinatorial optimization and a large number of papers and books deal with the numerous procedures that have been proposed to solve it, including several tabu search implementations. Although a large number of variants are found in the literature (and even more in the real world), the *classical* problem can be stated as follows. We first assume that n jobs must be scheduled on m machines. Each job corresponds to a fixed sequence of m operations, one per machine, where each operation must be processed on a specific machine for a specified duration. Note that the processing order on the machines does not need to be the same from one job to another. Each machine can process at most one operation at a time and, once started, an operation must be completed without interruption. The goal is to assign operations to time slots on the machines in order to minimize the maximum completion time of the jobs, which is also known as the makespan. A solution to this problem can be seen as a set of m permutations of the n jobs, one for each machine, with the associated machine schedules ([Anderson et al. 1997](#)).

9.2.2 The Capacitated Plant Location Problem

The capacitated plant location problem is one of the basic problems in location theory. It is encountered in many application settings that involve locating facilities with limited capacity to provide services. The problem can be formally described as follows. A set of customers I with demands $d_i, i \in I$, for some product are to be served from plants located in a subset of sites from a given set J of *potential sites*. For each site $j \in J$, the fixed cost of *opening* the plant at j is f_j and its capacity is K_j . The cost of transporting one unit of the product from site j to customer i is c_{ij} . The objective is to minimize the total cost, i.e. the sum of the fixed costs for open plants and the transportation costs.

Letting x_{ij} ($i \in I, j \in J$) denote the quantity shipped from site j to customer i (the x_{ij} are the so-called *flow variables*) and y_j ($j \in J$) be a 0–1 variable indicating whether or not the plant at site j is open (the y_j are the *location variables*), the problem can be formulated as the following mathematical program:

$$\begin{aligned} & \text{Minimize } z = \sum_{j \in J} f_j y_j + \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \\ & \text{subject to: } \sum_{j \in J} x_{ij} = d_i, \quad i \in I \\ & \qquad \qquad \qquad \sum_{i \in I} x_{ij} \leq K_j y_j, \quad j \in J \\ & \qquad \qquad \qquad x_{ij} \geq 0, \quad i \in I, j \in J \\ & \qquad \qquad \qquad y_j \in \{0, 1\}, \quad j \in J. \end{aligned}$$

Remark 9.1. For any vector \tilde{y} of location variables, optimal (w.r.t. to this plant configuration) values for the flow variables $x(\tilde{y})$ can be retrieved by solving the associated transportation problem:

$$\begin{aligned} \text{Minimize } z(\tilde{y}) &= \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \\ \text{subject to: } \sum_{j \in J} x_{ij} &= d_i, \quad i \in I \\ \sum_{i \in I} x_{ij} &\leq K_j \tilde{y}_j, \quad j \in J \\ x_{ij} &\geq 0, \quad i \in I, j \in J. \end{aligned}$$

If $\tilde{y} = y^*$, the optimal location variable vector, the optimal solution to the original problem is simply given by $(y^*, x(y^*))$.

Remark 9.2. An optimal solution of the original problem can always be found at an extreme point of the polyhedron of feasible flow vectors defined by the constraints

$$\begin{aligned} \sum_{j \in J} x_{ij} &= d_i, \quad i \in I \\ \sum_{i \in I} x_{ij} &\leq K_j, \quad j \in J \\ x_{ij} &\geq 0, \quad i \in I, j \in J. \end{aligned}$$

This property follows from the fact that the capacitated plant location problem can be interpreted as a fixed-charge problem defined in the space of the flow variables. This fixed-charge problem has a concave objective function that always admits an extreme point minimum. The optimal values for the location variables can easily be obtained from the optimal flow vector by setting y_j equal to 1 when $\sum_{i \in I} x_{ij} > 0$, and to 0 otherwise.

9.3 Basic Concepts

9.3.1 Historical Background

Before introducing the basic concepts of tabu search, we believe it is useful to go back in time to try to better understand the genesis of the method and how it relates to previous work.

Heuristics, i.e. approximate solution techniques, have been used since the beginnings of operations research to tackle difficult combinatorial problems. With the development of complexity theory in the early 1970s, it became clear that, since most of these problems were indeed NP-hard, there was little hope of ever finding efficient exact solution procedures for them. This realization emphasized the role of heuristics for solving the combinatorial problems that were encountered in real-life

applications and that needed to be tackled, whether or not they were NP-hard. While many different approaches were proposed and experimented with, the most popular ones were based on hill climbing. The latter can roughly be summarized as an iterative search procedure that, starting from an initial feasible solution, progressively improves it by applying a series of local modifications or moves (for this reason, hill climbing is in the family of local search methods). At each iteration, the search moves to an improving feasible solution that differs only slightly from the current one. In fact, the difference between the previous and the new solution amounts to one of the local modifications mentioned above. The search terminates when no more improvement is possible. At this point, we have a local optimum with regard to the local modifications considered by the hill climbing method. Clearly, this is an important limitation of the method: unless one is extremely lucky, this local optimum will often be a fairly mediocre solution. The quality of the solution obtained and computing times are usually highly dependent upon the *richness* of the set of transformations (moves) considered at each iteration.

In 1983, a new heuristic approach called simulated annealing ([Kirkpatrick et al. 1983](#)) was shown to converge to an optimal solution of a combinatorial problem, albeit in infinite computing time. Based on analogy with statistical mechanics, simulated annealing could be interpreted as a form of controlled random walk in the space of feasible solutions. The emergence of simulated annealing indicated that one could look for other ways to tackle combinatorial optimization problems and spurred the interest of the research community. In the following years, many other new approaches, mostly based on analogies with natural phenomena, were proposed such as tabu search, ant systems ([Dorigo 1992](#)) and threshold methods ([Dueck and Scheuer 1990](#)). Together with some older ones, in particular genetic algorithms ([Holland 1975](#)), they gained an increasing popularity. Now collectively known under the name of meta-heuristics, a term originally coined by [Glover \(1986\)](#), these methods have become, over the last 25 years, the leading edge of heuristic approaches for solving combinatorial optimization problems.

9.3.2 Tabu Search

In 1986 Fred Glover proposed a new approach, which he called tabu search, to allow hill climbing to overcome local optima. In fact, many elements of this first tabu search proposal, and some elements of later elaborations, had already been introduced ([Glover 1977](#)), including short-term memory to prevent the reversal of recent moves, and longer-term frequency memory to reinforce attractive components. The basic principle of tabu search is to pursue the search whenever a local optimum is encountered by allowing non-improving moves; cycling back to previously visited solutions is prevented by the use of memories, called tabu lists, that record the recent history of the search. The key idea to exploit information to guide the search can be linked to the informed search methods proposed in the late 1970s in the field of artificial intelligence ([Nilsson 1980](#)). It is important to note that Glover

did not see tabu search as a proper heuristic, but rather as a metaheuristic, i.e. a general strategy for guiding and controlling *inner* heuristics specifically tailored to the problems at hand.

9.3.3 Search Space and Neighborhood Structure

As just mentioned, tabu search extends hill climbing methods. In fact, the basic tabu search can be seen as simply the combination of hill climbing with short-term memories. It follows that the two first basic elements of any tabu search heuristic are the definition of its search space and its neighborhood structure.

The search space is simply the space of all possible solutions that can be considered (visited) during the search. For instance, in the job shop scheduling problem of Sect. 9.2.1, the search space could simply be the set of feasible solutions to the problem, where each point in the search space corresponds to a set of m machine schedules that satisfies all the specified constraints. While in that case the definition of the search space seems quite natural, it is not always so. Consider now the capacitated plant location problem of Sect. 9.2.2: the feasible space involves both integer location and continuous flow variables that are linked by strict conditions; moreover, as already indicated, for any feasible set of values for the location variables, one can fairly easily retrieve optimal values for the flow variables by solving the associated transportation problem. In this context, one could obviously use as a search space the full feasible space; this would involve manipulating both location and flow variables, which is not an easy task. A more attractive search space is the set of feasible vectors of location variables, i.e. feasible vectors in $\{0, 1\}^{|J|}$ (where $|J|$ is the cardinality of set J), any solution in that space being *completed* to yield a feasible solution to the original problem by computing the associated optimal flow variables. It is interesting to note that these two possible definitions are not the only ones. Indeed, on the basis of Remark 9.2, one could also decide to search instead the set of extreme points of the set of feasible flow vectors, retrieving the associated location variables by simply noting that a plant must be open whenever some flow is allocated to it. In fact, this type of approach was used successfully by Crainic et al. (2000) to solve the fixed-charge multi-commodity network design problem, which is a more general problem that includes the capacitated plant location problem as a special case. It is also important to note that it is not always a good idea to restrict the search space to feasible solutions. In many cases, allowing the search to move to infeasible solutions is desirable, and sometimes necessary (see Sect. 9.4.3 for further details).

Closely linked to the definition of the search space is that of the neighborhood structure. At each iteration of tabu search, the local transformations that can be applied to the current solution, denoted S , define a set of neighboring solutions in the search space, denoted $N(S)$ (the neighborhood of S). Formally, $N(S)$ is a subset of the search space defined by

$$N(S) = \{\text{solutions obtained by applying a single local transformation to } S\}.$$

In general, for any specific problem at hand, there are many more possible (and even, attractive) neighborhood structures than search space definitions. This follows from the fact that there may be several plausible neighborhood structures for a given definition of the search space. This is easily illustrated on our job shop scheduling problem. In order to simplify the discussion, we assume in the following that the search space is the feasible space.

Simple neighborhood structures for the job shop scheduling problem are obtained by considering the sequence of jobs associated with a machine schedule, where the position of a job in the sequence corresponds to its processing order on the machine. For example, one can move a job at another position in the sequence or interchange the position of two jobs. While these neighborhood structures involve only one or two jobs, the neighborhoods they define contain all the feasible schedules that can be obtained from the current one either by moving any single job at any other position or by interchanging any two jobs. Examining these neighborhoods can thus be fairly demanding. In practice, it is often possible to reduce the computational burden, by identifying a restricted subset of moves that are feasible and can lead to improvements. We refer the interested reader to [Vaessens et al. \(1996\)](#) and [Anderson et al. \(1997\)](#) for a more detailed discussion of these issues.

When different definitions of the search space are considered for a given problem, neighborhood structures will inevitably differ to a considerable degree. This can be illustrated on our capacitated plant location problem. If the search space is defined with respect to the location variables, neighborhood structures will usually involve the so-called Add/Drop and Swap moves that respectively change the status of one site (i.e. either opening a closed facility or closing an open one) and move an open facility from one site to another (this move amounts to performing simultaneously an Add move and a Drop move). If, however, the search space is the set of extreme points associated with feasible flow vectors, these moves become meaningless. One should instead consider moves defined by the application of pivots to the linear programming formulation of the transportation problem, where each pivot operation modifies the flow structure to move the current solution to an adjacent extreme point.

The preceding discussion should have clarified a major point: choosing a search space and a neighborhood structure is by far the most critical step in the design of any tabu search heuristic. It is at this step that one must make the best use of the understanding and knowledge he/she has of the problem at hand.

9.3.4 Tabus

Tabus are one of the distinctive elements of tabu search when compared to hill climbing. As already mentioned, tabus are used to prevent cycling when moving away from local optima through non-improving moves. The key realization here is that when this situation occurs, something needs to be done to prevent the search from tracing back its steps to where it came from. This is achieved by making certain

actions tabu. This might mean not allowing the search to return to a recently visited point in the search space or not allowing a recent move to be reversed. For example, in the job shop scheduling problem, if a job j has been moved to a new position in a machine schedule, one could declare tabu moving that job back to its previous position for some number of iterations (this number is called the tabu tenure of the move).

Tabus are stored in a short-term memory of the search (the tabu list) and usually only a fixed and fairly limited quantity of information is recorded. In any given context, there are several possibilities regarding the recorded information. One could record complete solutions, but this requires a lot of storage and makes it expensive to check whether a potential move is tabu or not; it is therefore seldom used. The most commonly used tabus involve recording the last few transformations performed on the current solution and prohibiting reverse transformations (as in the example above); others are based on key characteristics of the solutions themselves or of the moves.

To better understand how tabus work, let us go back to our reference problems. In the job shop scheduling problem, one could define tabus in several ways. To continue our example where a job j has just been moved from position p_1 to position p_2 , one could declare tabu specifically moving back j to position p_1 from position p_2 and record this in the short-term memory as the triplet (j, p_2, p_1) . Note that this type of tabu will not constrain the search much, but that cycling may occur if j is then moved to another position p_3 and then from p_3 to p_1 . A stronger tabu would involve prohibiting moving back j to p_1 (without consideration for its current position) and be recorded as (j, p_1) . An even stronger tabu would be to disallow moving j at all, and would simply be noted as (j) .

In the capacitated plant location problem, tabus on Add/Drop moves should prohibit changing the status of the affected location variable and can be recorded by noting its index. Tabus for Swap moves are more complex. They could be declared with respect to the site where the facility was closed, to the site where the facility was opened, to both locations (i.e. changing the status of both location variables is tabu), or to the specific swapping operation.

Multiple tabu lists can be used simultaneously and are sometimes advisable. For example, in the capacitated plant location problem, if one uses a neighborhood structure that contains both Add/Drop and Swap moves, it might be a good idea to keep a separate tabu list for each type of move.

Standard tabu lists are usually implemented as circular lists of fixed length. It has been shown, however, that fixed-length tabus cannot always prevent cycling, and some authors have proposed varying the tabu list length during the search (Glover 1989, 1990; Skorin-Kapov 1990; Taillard 1990, 1991). Another solution is to randomly generate the tabu tenure of each move within some specified interval. Using this approach requires a somewhat different scheme for recording tabus, which are usually stored as tags in an array. The entries in this array typically record the iteration number until which a move is tabu. More details are provided in Gendreau et al. (1994).

9.3.5 Aspiration Criteria

While central to the tabu search method, tabus are sometimes too powerful. They may prohibit attractive moves, even when there is no danger of cycling, or they may lead to an overall stagnation of the search process. It is thus necessary to use algorithmic devices that will allow one to revoke (cancel) tabus. These are called aspiration criteria. The simplest and most commonly used aspiration criterion, found in almost all tabu search implementations, allows a tabu move when it results in a solution with an objective value better than that of the current best-known solution (since the new solution has obviously not been previously visited). Much more complicated aspiration criteria have been proposed and successfully implemented (see, for example, [de Werra and Hertz 1989](#), and [Hertz and de Werra 1991](#)), but they are rarely used. The key rule is that if cycling cannot occur, tabus can be disregarded.

9.3.6 A Template for Simple Tabu Search

We are now in the position to give a general template for tabu search, integrating the elements we have seen so far. We suppose that we are trying to minimize a function $f(S)$ (sometimes known as an objective or evaluation function) over some domain and we apply the so-called *best improvement* version of tabu search, i.e. the version in which one chooses at each iteration the best available move (even if this results in an increase in the function $f(S)$). This is the most commonly used version of tabu search.

Notation

- S , the current solution
- S^* , the best-known solution
- f^* , value of S^*
- $N(S)$, the neighborhood of S
- $\tilde{N}(S)$, the *admissible* subset of $N(S)$ (i.e. non-tabu or allowed by aspiration)
- T , tabu list.

Initialization

Choose (construct) an initial solution S_0 .

Set $S \leftarrow S_0$, $f^* \leftarrow f(S_0)$, $S^* \leftarrow S_0$, $T \leftarrow \emptyset$.

Search

While *termination criterion not satisfied* do

- Select S in $\arg \min_{S' \in \tilde{N}(S)} [f(S')]$;
- If $f(S) < f^*$, then set $f^* \leftarrow f(S)$, $S^* \leftarrow S$
- Record tabu for the current move in T (delete oldest entry if necessary)

Endwhile.

In this algorithm, $\arg\min$ returns the subset of solutions in $\tilde{N}(S)$ that minimizes f .

9.3.7 Termination Criteria

Note that we have not specified in our template a termination criterion. In theory, the search could go on forever, unless the optimal value of the problem at hand is known beforehand. In practice, obviously, the search has to be stopped at some point. The most commonly used stopping criteria in tabu search are

- After a fixed number of iterations (or a fixed amount of CPU time)
- After some number of consecutive iterations without an improvement in the objective function value (the criterion used in most implementations)
- When the objective function reaches a pre-specified threshold value.

9.3.8 Probabilistic Tabu Search and Candidate Lists

Normally, one must evaluate the objective function for every element of the neighborhood $N(S)$ of the current solution. This can be extremely expensive from a computational standpoint. In probabilistic tabu search, only a random sample $N'(S)$ of $N(S)$ is considered, thus significantly reducing the computational overhead. Another attractive feature is that the added randomness can act as an anti-cycling mechanism. This allows one to use shorter tabu lists than would be necessary if a full exploration of the neighborhood was performed. On the negative side, it is possible to miss excellent solutions (see Sect. 9.6.3 for more detail). It is also possible to probabilistically select when to apply tabu criteria.

Another way to control the number of moves examined is by means of candidate list strategies, which provide more strategic ways of generating a useful subset $N'(S)$ of $N(S)$. In fact, the probabilistic approach can be considered to be one instance of a candidate list strategy, and may also be used to modify such a strategy. Failure to adequately address the issues involved in creating effective candidate lists is one of the more conspicuous shortcomings that differentiates a naive tabu

search implementation from one that is more solidly grounded. Relevant designs for candidate list strategies are discussed in [Glover and Laguna \(1997\)](#). We also discuss a useful type of candidate generation approach in Sect. 9.4.4.

9.4 Extensions to the Basic Concepts

Simple tabu search as described above can sometimes successfully solve difficult problems, but in most cases, additional elements have to be included in the search strategy to make it fully effective. We now briefly review the most important of these.

9.4.1 Intensification

The idea behind the concept of search intensification is that, as an intelligent human being would probably do, one should explore more thoroughly the portions of the search space that seem *promising* in order to make sure that the best solutions in these areas are found. In general, intensification is based on some intermediate-term memory, such as a recency memory, in which one records the number of consecutive iterations that various *solution components* have been present in the current solution without interruption. For instance, in the capacitated plant location problem, one could record how long each site has had an open facility. A typical approach to intensification is to restart the search from the best currently known solution and to *freeze* (fix) in it the components that seem more attractive. To continue with our capacitated plant location problem, one could freeze a number of facilities in sites that have been often selected in previous iterations and perform a restricted search on the other sites. Another technique that is often used consists of changing the neighborhood structure to one allowing more powerful or more diverse moves. In the capacitated plant location problem, if Add/Drop moves were used, Swap moves could be added to the neighborhood structure. In probabilistic tabu search, one could increase the sample size or switch to searching without sampling. Intensification is used in many tabu search implementations, although it is not always necessary. This is because there are many situations where the normal search process is thorough enough.

9.4.2 Diversification

One of the main problems of all methods based on local search, and this includes tabu search in spite of the beneficial impact of tabus, is that they tend to be too *local* (as their name implies), i.e. they tend to spend most, if not all, of their time in a

restricted portion of the search space. The negative consequence of this fact is that, although good solutions may be obtained, one may fail to explore the most interesting parts of the search space and thus end up with solutions that are still far from the optimal ones. Diversification is an algorithmic mechanism that tries to alleviate this problem by forcing the search into previously unexplored areas of the search space. It is usually based on some form of long-term memory of the search, such as a frequency memory, in which one records the total number of iterations (since the beginning of the search) that various *solution components* have been present in the current solution or have been involved in the selected moves. For instance, in the capacitated plant location problem, one could record the number of iterations during which each site has had an open facility. In the job shop scheduling problem, one could note how many times each job has been moved. In cases where it is possible to identify useful *regions* of the search space, the frequency memory can be refined to track the number of iterations spent in these different regions.

There are two major diversification techniques. The first, called restart diversification, involves introducing a few rarely used components in the current solution (or the best known solution) and restarting the search from this point. In the capacitated plant location problem, one could thus open one or more facilities at locations that have seldom been used up to that point and resume searching from that plant configuration (one could also close facilities at locations that have been used the most frequently). In the job shop scheduling problem, a job that has not occupied a particular position in a machine schedule can be forced to that position. The second diversification method, called continuous diversification, integrates diversification considerations directly into the regular searching process. This is achieved by biasing the evaluation of possible moves by adding to the objective a small term related to component frequencies. An extensive discussion on these two techniques is provided by [Soriano and Gendreau \(1996\)](#). A third way of achieving diversification is strategic oscillation, which is discussed in the next section.

We would like to stress that ensuring proper search diversification is possibly the most critical issue in the design of tabu search heuristics. It should be addressed with extreme care fairly early in the design phase and revisited if the results obtained are not up to expectations.

9.4.3 Allowing Infeasible Solutions

Accounting for all problem constraints in the definition of the search space often restricts the searching process too much and can lead to mediocre solutions. In such cases, constraint relaxation is an attractive strategy, since it creates a larger search space that can be explored with *simpler* neighborhood structures. Constraint relaxation is easily implemented by dropping selected constraints from the search space definition and adding to the objective, weighted penalties for constraint violations. In the capacitated plant location problem, this can be done by allowing solutions with flows that exceed the capacity of one or more plants. This, however, raises

the issue of finding correct weights for constraint violations. An interesting way of circumventing this problem is to use self-adjusting penalties, i.e. weights are adjusted dynamically on the basis of the recent history of the search. Weights are increased if only infeasible solutions were encountered in the last few iterations, and decreased if all recent solutions were feasible; see [Gendreau et al. \(1994\)](#) for further details. Penalty weights can also be modified systematically to drive the search to cross the feasibility boundary of the search space and thus induce diversification. This technique, known as strategic oscillation, was introduced in [Glover \(1977\)](#) and used since in several successful tabu search procedures. An important early variant oscillates among alternative types of moves, hence neighborhood structures, while another oscillates around a selected value for a critical function.

9.4.4 Surrogate and Auxiliary Objectives

There are many problems for which the true objective function is quite costly to evaluate, a typical example being the capacitated plant location problem when one searches the space of location variables. Remember that, in this case, computing the objective value for any potential solution entails solving the associated transportation problem. When this occurs, the evaluation of moves may become prohibitive, even if sampling is used. An effective approach to handle this issue is to evaluate neighbors using a surrogate objective, i.e. a function that is correlated to the true objective, but is less computationally demanding, in order to identify a small set of promising candidates (potential solutions achieving the best values for the surrogate). The true objective is then computed for this small set of candidate moves and the best one selected to become the new current solution. An example of this approach is found in [Crainic et al. \(1993\)](#).

Another frequently encountered difficulty is that the objective function may not provide enough information to effectively drive the search to more interesting areas of the search space. A typical illustration of this situation is observed when the fixed costs for open plants in the capacitated plant location problem are much larger than the transportation costs. In this case, it is indicated to open as few plants as possible. It is thus important to define an auxiliary objective function to orient the search. Such a function must measure in some way the desirable attributes of the solutions. In our example, one could use a function that would favor, for the same number of open plants, solutions with plants having just a small amount of flow, thus increasing the likelihood of closing them in subsequent iterations. It should be noted that developing an effective auxiliary objective is not always easy and may require a lengthy trial and error process. In some other cases, fortunately, the auxiliary objective is obvious for anyone familiar with the problem at hand (for an illustration, see the work of [Gendreau et al. 1993](#)).

9.5 Promising Areas for Future Applications

The concepts and techniques described in the previous sections are sufficient to design effective tabu search heuristics for many combinatorial problems. Most early tabu search implementations, several of which were extremely successful, relied indeed almost exclusively on these algorithmic components (Friden et al. 1989; Hertz and de Werra 1987; Skorin-Kapov 1990; Taillard 1991). Nowadays, however, most leading-edge research in tabu search makes use of more advanced concepts and techniques. While it is clearly beyond the scope of an introductory tutorial such as this to review this type of advanced material, we would like to give readers some insight into it by briefly describing some current trends. Readers who wish to learn more about this topic should read our survey paper (Gendreau 2002) and some of the other references provided here.

A large part of the recent research in tabu search deals with various techniques for making the search more effective. These include methods for better exploitation of the information that becomes available during search and creating better starting points, as well as more powerful neighborhood operators and parallel search strategies. For more details, see the taxonomy of Crainic et al. (1997), the survey of Cung et al. (2002) and the book of Alba (2005). The numerous techniques for utilizing the information are of particular significance since they can lead to dramatic performance improvements. Many of these rely on elite solutions (the best solutions previously encountered) or on parts of these to create new solutions, the rationale being that *fragments* of excellent solutions are often identified quite early in the search process. However, the challenge is to complete these fragments or to recombine them (Glover 1992; Glover and Laguna 1993, 1997; Rochat and Taillard 1995). Other methods, such as the reactive tabu search of Battiti and Tecchiolli (1994), are aimed at finding ways to move the search away from local optima that have already been visited.

Another important trend is hybridization which is, in fact, a pervasive trend in the whole metaheuristics field (for example, a series of workshops on hybrid metaheuristics has been held each year since 2004, see Blum et al. 2008). Tabu search has been used in conjunction with many different solution approaches such as genetic algorithms (Crainic and Gendreau 1999; Fleurent and Ferland 1996), Lagrangean relaxation (Grünert 2002), constraint programming (Pesant and Gendreau 1999), column generation (Crainic et al. 2000) and neural networks (Ateme-Nguema and Dao 2009; Wang et al. 2009). A whole chapter on this topic is found in Glover and Laguna (1997). Problem-specific information and simple heuristics can also be used in conjunction with different components of tabu search. For example, in Burke et al. (1998), problem-specific heuristics are used to realize diversification.

The literature on tabu search has also started moving away from its traditional application areas (graph theory problems, scheduling, vehicle routing) to new ones: continuous optimization (Rolland 1996), multi-objective optimization (Gandibleux et al. 2000), stochastic programming (Lokketangen and Woodruff 1996), mixed integer programming (Crainic et al. 2000; Lokketangen and Woodruff 1996), real-time

decision problems (Gendreau et al. 2006), etc. These new areas confront researchers with new challenges that, in turn, call for novel and original extensions of the method.

9.6 Tricks of the Trade

9.6.1 Getting Started

Newcomers to tabu search, trying to apply the method to a problem that they wish to solve, are often confused about what they need to do to come up with a successful implementation. Basically, they do not know where to start. We believe that the following step-by-step procedure will help and provides a useful framework for getting started.

A Step-by-Step Procedure

1. *Read one or two good introductory papers* to gain some knowledge of the concepts and of the vocabulary (see the references provided in *Sources of Additional Information*).
2. *Read several papers describing in detail applications in various areas* to see how the concepts have been actually implemented by other researchers (see the references provided in *Sources of Additional Information*).
3. *Think a lot* about the problem at hand, focusing on the definition of the *search space* and the *neighborhood structure*.
4. *Implement a simple version* based on this search space definition and this neighborhood structure.
5. *Collect statistics* on the performance of this simple heuristic. It is usually useful at this point to introduce a variety of *memories*, such as frequency and recency memories, to really track down what the heuristic does.
6. *Analyze results and adjust* the procedure accordingly. It is at this point that one should eventually introduce mechanisms for search intensification and diversification or other intermediate features. Special attention should be paid to *diversification*, since this is often where simple tabu search procedures fail.

9.6.2 More Tips

In spite of carefully following the procedure outlined above, it is possible to end up with a heuristic that produces mediocre results. If this occurs, the following tips may prove useful:

1. If there are *constraints*, consider *penalizing the violation* of them. Letting the search move to infeasible solutions is often necessary in highly constrained problems to allow for a meaningful exploration of the search space (see Sect. 9.4.3).
2. Reconsider the *neighborhood structure* and change it if necessary. Many tabu search implementations fail because the neighborhood structure is too simple. In particular, one should make sure that the chosen neighborhood structure allows for a sensible evaluation of possible moves (i.e. the moves that seem intuitively to move the search in the *right* direction should be the ones that are likely to be selected); it might also be a good idea to introduce a *surrogate objective* (see Sect. 9.4.4) to achieve this.
3. *Collect more statistics.* For example, recording the number and quality of previously visited local optima can be useful to find a good trade-off between intensification and diversification
4. *Follow the execution of the algorithm step by step* on some reasonably sized instances. (For example: Is the algorithm behaving as expected on particular solution configurations? Is the algorithm converging prematurely?)
5. Reconsider *diversification*. As mentioned earlier, this is a critical feature in most tabu search implementations.
6. *Experiment with parameter settings.* Many tabu search procedures are extremely sensitive to parameter settings; it is not unusual to see the performance of a procedure dramatically improve after changing the value of one or two key parameters (unfortunately, it is not always obvious to determine which parameters are the key ones in a given procedure).

9.6.3 Additional Tips for Probabilistic Tabu Search

While probabilistic tabu search is an effective way of tackling many problems, it creates difficulties of its own that need to be carefully addressed. The most important of these occurs because, more often than not, the best solutions returned by probabilistic tabu search will not be local optima with respect to the neighborhood structure being used. This is particularly annoying since, when it happens, better solutions can be easily obtained, sometimes even manually. An easy way to address this is to simply perform a local improvement phase (using the same neighborhood operator) from the best found solution at the end of the tabu search itself. One could alternately switch to tabu search without sampling (again from the best found solution) for a short duration before completing the algorithm. A possibly more effective technique is to add, throughout the search, an intensification step without sampling. This will mean that the best solutions available in the various regions of the space explored by the method will be found and recorded. This is similar to the method proposed by [Glover and Laguna \(1993\)](#). They employed special aspiration criteria for allowing the search to reach local optima at useful junctures.

9.6.4 Parameter Calibration and Computational Testing

Parameter calibration and computational experiments are key steps in the development of any algorithm. This is particularly true in the case of tabu search, since the number of parameters required by most implementations is fairly large and the performance of a given procedure can vary quite significantly when parameter values are modified. The first step in any serious computational experimentation is to select a good set of benchmark instances (either by obtaining them from other researchers or by constructing them), preferably with some reasonable measure of their difficulty and with a wide range of size and difficulty. This set should be split into two subsets, the first one being used at the algorithmic design and parameter calibration steps, and the second reserved for performing the final computational tests that will be reported in the paper(s) describing the heuristic under development. The reason for doing so is quite simple: when calibrating parameters, one always runs the risk of overfitting, i.e. finding parameter values that are excellent for the instances at hand, but poor in general, because these values provide too good a *fit* (from the algorithmic standpoint) to these instances. Methods with several parameters should thus be calibrated on much larger sets of instances than ones with few parameters to ensure a reasonable degree of robustness. The calibration process itself should proceed in several stages:

1. Perform exploratory testing to find good ranges of parameters. This can be done by running the heuristic with a variety of parameter settings.
2. Fix the value of the parameters that appear to be *robust*, i.e. which do not seem to have a significant impact on the performance of the algorithm.
3. Perform systematic testing for the other parameters. It is usually more efficient to test values for only a single parameter at a time, the others being fixed at what appear to be reasonable values. One must be careful, however, for cross-effects between parameters. For example, assume that value x_1 for parameter p_1 leads to good results when the other parameters are fixed at their default values, and that value x_2 for parameter p_2 leads to good results when the other parameters are fixed at their default values. Then, it might happen that value x_1 for parameter p_1 and value x_2 for parameter p_2 lead to poor results. Where such effects exist, it can be important to jointly test pairs or triplets of parameters, which can be an extremely time-consuming task.

The paper by [Crainic et al. \(1993\)](#) provides a detailed description of the calibration process for a fairly complex tabu search procedure and can be used as a guideline for this purpose.

9.7 Conclusions

Tabu search is a powerful algorithmic approach that has been applied with great success to many difficult combinatorial problems. A particularly nice feature of tabu search is that it can quite easily handle the *dirty* complicating constraints that

are typically found in real-life applications. It is thus a really practical approach. It is not, however, a panacea: every reviewer or editor of a scientific journal has seen more than his/her share of failed tabu search heuristics. These failures stem from two major causes: an insufficient understanding of fundamental concepts of the method (and we hope that this tutorial may help in alleviating this shortcoming), but also, more often than not, a crippling lack of understanding of the problem at hand. One cannot develop a good tabu search heuristic for a problem that one does not know well! This is because significant problem knowledge is absolutely vital to perform the most basic steps of the development of any tabu search procedure, namely the choice of a search space and the choice of an effective neighborhood structure. If the search space and/or the neighborhood structure are inadequate, no amount of tabu search expertise will be sufficient to save the day. A last word of caution: to be successful, all metaheuristics need to achieve both depth and breadth in their searching process; depth is usually not a problem for tabu search, which is quite aggressive in this respect (it generally finds pretty good solutions very early in the search), but breadth can be a critical issue. To handle this, it is extremely important to develop an effective diversification scheme.

Sources of Additional Information

- Good introductory papers on tabu search may be found in [Glover and Laguna \(1993\)](#), [Glover et al. \(1993\)](#), [Hertz and de Werra \(1991\)](#), [Hindsberger and Vidal \(2000\)](#), [de Werra and Hertz \(1989\)](#) and, in French, in [Soriano and Gendreau \(1997\)](#).
- The book by [Glover and Laguna \(1997\)](#) is the ultimate reference on tabu search. Apart from the fundamental concepts of the method, it presents a considerable amount of advanced material, as well as a variety of applications. It is interesting to note that this book contains several ideas applicable to tabu search that yet remain to be fully exploited.
- Two issues of *Annals of Operations Research* devoted respectively to *Tabu Search* ([Glover et al. 1993](#)) and *Metaheuristics in Combinatorial Optimization* ([Laporte and Osman 1996](#)) provide a good sample of applications of tabu search.
- The books made up from selected papers presented at the Metaheuristics International Conferences (MIC) are also extremely valuable. At this time, the books for the following conferences are available: Breckenridge 1995 ([Osman and Kelly 1996](#)), Sophia-Antipolis, 1997 ([Voss et al. 1999](#)), Angra dos Reis, 1999 ([Ribeiro and Hansen 2002](#)), Porto 2001 ([Resende and de Sousa 2004](#)), Kyoto, 2003 ([Ibaraki et al. 2005](#)) and Vienna, 2005 ([Doerner et al. 2007](#)). A book for the 2009 conference in Hamburg is also planned. Finally, a special issue of *Journal of Heuristics* was devoted to the 2007 conference in Montreal ([Crainic et al. 2010](#)).
- Three books of interest have also been published. The first one, edited by [Rego and Alidaee \(2005\)](#), deals with tabu search and scatter search. The one edited by

[Glover and Kochenberger \(2003\)](#) addresses metaheuristics in general (a second edition of this book, edited by Gendreau and Potvin, was published in 2010). The third book, edited by [Pardalos and Resende \(2002\)](#), has a broader scope but contains a nice chapter on metaheuristics.

References

- Alba E (ed) (2005) Parallel metaheuristics: a new class of algorithms. Wiley, Hoboken
- Anderson EJ, Glass CA, Potts CN (1997) Machine scheduling, in local search in combinatorial optimization. In: Aarts EHL, Lenstra JK (eds). Wiley, New York, pp 361–414
- Ateme-Nguema B, Dao T-M (2009) Quantized Hopfield networks and tabu search for manufacturing cell formation problems. *Int J Product Econ* 121:88–98
- Battini R, Tecchiolli G (1994) The reactive tabu search. *ORSA J Comput* 6:126–140
- Blum C, Blesa Aguilera MJ, Roli A, Sampels M (eds) (2008) Hybrid metaheuristics: an emerging approach to optimization. Springer, Berlin
- Burke E, De Causmaecker P, Vanden Berghe G (1998) A Hybrid Tabu Search Algorithm for the Nurse Rostering Problem. In: Selected papers from the 2nd Asia Pacific conference on simulated evolution and learning, LNAI 1585. Springer, Berlin, Canberra, Australia, pp 187–194
- Crainic TG, Gendreau M (1999) Towards an evolutionary method—cooperative multi-thread parallel tabu search heuristic hybrid. In: Voss S et al (eds) Metaheuristics: advances and trends in local search paradigms for optimization. Kluwer, Norwell, pp 331–344
- Crainic TG, Gendreau M, Soriano P, Toulouse M (1993) A tabu search procedure for multicommodity location/allocation with balancing requirements. *Ann Oper Res* 41:359–383
- Crainic TG, Toulouse M, Gendreau M (1997) Toward a taxonomy of parallel tabu search heuristics. *INFORMS J Comput* 9:61–72
- Crainic TG, Gendreau M, Farvolden JM (2000) Simplex-based tabu search for the multicommodity capacitated fixed charge network design problem. *INFORMS J Comput* 12:223–236
- Crainic TG, Gendreau M, Rousseau L-M (eds) (2010) *J Heuristics* 16:235–535 (Special issue: Recent advances in metaheuristics)
- Cung V-D, Martins SL, Ribeiro CC, Roucairol C (2002) Strategies for the parallel implementation of metaheuristics. In: Ribeiro CC, Hansen P (eds) Essays and surveys in metaheuristics. Kluwer, Norwell, pp 263–308
- de Werra D, Hertz A (1989) Tabu search techniques: a tutorial and an application to neural networks. *OR Spektrum* 11:131–141
- Doerner KF, Gendreau M, Greistorfer P, Gutjahr WJ, Hartl RF, Reimann M (eds) (2007) Metaheuristics: progress in complex systems optimization. Springer, New York

- Dorigo M (1992) Optimization, learning and natural algorithms. PhD Dissertation, Dipartimento di Elettronica, Politecnico di Milano
- Dueck G, Scheuer T (1990) Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *J Comput Phys* 90:161–175
- Fleurent C, Ferland JA (1996) Genetic and hybrid algorithms for graph colouring. *Ann Oper Res* 63:437–461
- Friden C, Hertz A, de Werra D (1989) STABULUS: a technique for finding stable sets in large graphs with tabu search. *Computing* 42:35–44
- Gandibleux X, Jaszkiewicz A, Freville A, Slowinski R (eds) (2000) *J Heuristics* 6:291–431 (Special issue: Multiple objective metaheuristics)
- Gendreau M (2002) Recent advances in tabu search. In: Ribeiro CC, Hansen P (eds) *Essays and surveys in metaheuristics*. Kluwer, Norwell, pp 369–377
- Gendreau M, Soriano P, Salvail L (1993) Solving the maximum clique problem using a tabu search approach. *Ann Oper Res* 41:385–403
- Gendreau M, Hertz A, Laporte G (1994) A tabu search heuristic for the vehicle routing problem. *Manage Sci* 40:1276–1290
- Gendreau M, Guertin F, Potvin J-Y, Séguin R (2006) Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries. *Transp Res C* 14:157–174
- Glover F (1977) Heuristics for integer programming using surrogate constraints. *Decis Sci* 8:156–166
- Glover F (1986) Future paths for integer programming and links to artificial intelligence. *Comput Oper Res* 13:533–549
- Glover F (1989) Tabu search I. *ORSA J Comput* 1:190–206
- Glover F (1990) Tabu search II. *ORSA J Comput* 2:4–32
- Glover F (1992) Ejection chains, reference structures and alternating path methods for traveling salesman problems. University of Colorado Report (Shortened version published in *Discret Appl Math* 65:223–253, 1996)
- Glover F, Kochenberger GA (eds) (2003) *Handbook of metaheuristics*. Kluwer, Norwell
- Glover F, Laguna M (1993) Tabu search. In: Reeves CR (ed) *Modern heuristic techniques for combinatorial problems*. Halsted Press, New York, pp 70–150
- Glover F, Laguna M (1997) Tabu search. Kluwer, Norwell
- Glover F, Laguna M, Taillard ED, de Werra D (eds) (1993) Tabu search. *Ann Oper Res* 41, Baltzer Science, Basel, pp 1–490
- Glover F, Taillard ED, de Werra D (1993) A user's guide to tabu search. *Ann Oper Res* 41:3–28
- Grünert T (2002) Lagrangean tabu search. In: Ribeiro CC, Hansen P (eds) *Essays and surveys in metaheuristics*. Kluwer, Norwell, pp 379–397
- Hertz A, de Werra D (1987) Using tabu search for graph coloring. *Computing* 39:345–351
- Hertz A, de Werra D (1991) The tabu search metaheuristic: how we used it. *Ann Math Artif Intell* 1:111–121
- Hindsberger M, Vidal RVV (2000) Tabu search—a guided tour. *Control Cybern* 29:631–651

- Holland JH (1975) *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor
- Ibaraki T, Nonobe K, Yagiura M (eds) (2005) *Metaheuristics: progress as real problem solvers*. Springer, New York
- Jaziri W (2008) Local search techniques: focus on tabu search. In-Teh, Croatia
- Kirkpatrick S, Gelatt CD Jr, Vecchi MP (1983) Optimization by simulated annealing. *Science* 220:671–680
- Laporte G, Osman IH (eds) (1996) *Metaheuristics in combinatorial optimization*. Ann Oper Res 63, Baltzer Science, Basel, pp 1–630
- Lokketangen A, Glover F (1996) Probabilistic move selection in tabu search for 0/1 mixed integer programming problems. In: Osman IH, Kelly JP (eds) *Metaheuristics: theory and applications*. Kluwer, Norwell, pp 467–488
- Lokketangen A, Woodruff DL (1996) Progressive hedging and tabu search applied to mixed integer (0,1) multistage stochastic programming. *J Heuristics* 2:111–128
- Nilsson NJ (1980) *Principles of artificial intelligence*. Morgan Kaufmann, Los Altos
- Osman IH, Kelly JP (eds) (1996) *Meta-heuristics: theory and applications*. Kluwer, Norwell
- Pardalos PM, Resende MGC (eds) (2002) *Handbook of applied optimization*. Oxford University Press, New York
- Pesant G, Gendreau M (1999) A constraint programming framework for local search methods. *J Heuristics* 5:255–280
- Rego C, Alidaee B (eds) (2005) *Metaheuristic optimization via memory and evolution: tabu search and scatter search*. Kluwer, Norwell
- Resende MGC, de Sousa JP (eds) (2004) *Metaheuristics: computer decision-making*, Kluwer, Norwell
- Ribeiro CC, Hansen P (eds) (2002) *Essays and surveys in metaheuristics*. Kluwer, Norwell
- Rochat Y, Taillard ED (1995) Probabilistic diversification and intensification in local search for vehicle routing. *J Heuristics* 1:147–167
- Rolland E (1996) A tabu search method for constrained real-number search: applications to portfolio selection, Working Paper, The Gary Anderson Graduate School of Management, University of California, Riverside
- Skorin-Kapov J (1990) Tabu search applied to the quadratic assignment problem. *ORSA J Comput* 2:33–45
- Soriano P, Gendreau M (1996) Diversification strategies in tabu search algorithms for the maximum clique problems. *Ann Oper Res* 63:189–207
- Soriano P, Gendreau M (1997) Fondements et applications des méthodes de recherche avec tabous. *RAIRO—Recherche Opérationnelle* 31:133–159
- Taillard ED (1990) Some efficient heuristic methods for the flow shop sequencing problem. *Eur J Oper Res* 47:65–74
- Taillard ED (1991) Robust taboo search for the quadratic assignment problem. *Parallel Comput* 17:443–455
- Vaessens RJM, Aarts EHL, Lenstra JK (1996) Job shop scheduling by local search. *INFORMS J Comput* 8:302–317

- Voss S, Martello S, Osman IH, Roucairol C (eds) (1999) Meta-heuristics: advances and trends in local search paradigms for optimization. Kluwer, Norwell
- Wang Y, Li L, Ni J, Huang S (2009) Feature selection using tabu search with long-term memories and probabilistic neural networks. Pattern Recognit Lett 30:661–670

Chapter 10

Simulated Annealing

Emile Aarts, Jan Korst and Wil Michiels

10.1 Introduction

Many problems in engineering, planning and manufacturing can be modeled as that of minimizing or maximizing a cost function over a finite set of discrete variables. This class of so-called combinatorial optimization problems has received much attention over the years and major achievements have been made in its analysis ([Ausiello et al. 1999](#)). One of these achievements is the separation of this class into two subclasses. The first one contains the problems that can be efficiently solved, i.e. problems for which algorithms are known that solve each instance to optimality in polynomial time. Examples are linear programming, matching and network problems. The second subclass contains the problems that are notoriously hard—formally referred to as NP-hard—and for which it is generally believed that no algorithms exist that solve each instance in polynomial time. Consequently, there are instances that require superpolynomial or exponential time to be solved to optimality. Many known problems belong to this class and probably the best known example is the traveling salesman problem (TSP). The above-mentioned distinction is supported by a general discipline in computer science called complexity theory; for a detailed introduction and an extensive listing of provably hard problems see [Garey and Johnson \(1979\)](#) and [Arora and Barak \(2009\)](#).

E. Aarts
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: e.h.l.aarts@tue.nl

J. Korst
Philips Research Laboratories, Eindhoven, The Netherlands
e-mail: jan.korst@philips.com

W. Michiels (✉)
NXP, Eindhoven, The Netherlands
e-mail: wil.michiels@philips.com

Clearly, also hard problems must be handled in practice. Roughly speaking, this can be done by two types of algorithms of inherently different nature: either one may use *optimization algorithms* that find optimal solutions possibly using large amounts of computation time or one may use *heuristic algorithms* that find approximate solutions in relatively small amounts of computation time. Local search algorithms are of the latter type (Aarts and Lenstra 2003; Michiels et al. 2007). Simulated annealing, the subject of this chapter, is among the best known local search algorithms, since it performs quite well and is widely applicable. In this chapter we present the basics of simulated annealing. The chapter summarizes the treatment of simulated annealing contained in Michiels et al. (2007). First, we introduce some elementary local search concepts. We introduce basic simulated annealing as an approach following directly from the strong analogy with the physical process of the annealing of solids. We analyze the asymptotic performance of basic simulated annealing. Next, we present some cooling schedules that allow for a finite-time implementation. Finally, we discuss some issues related to the practical use of simulated annealing and conclude with some suggestions for further reading.

10.2 Local Search

Local search algorithms constitute a widely used, general approach to hard combinatorial optimization problems. They are typically instantiations of various general search schemes, but all have the same feature of an underlying neighborhood function, which is used to guide the search for a good solution. To make this more precise, we introduce in this section some notation and definitions.

An instance of a combinatorial optimization problem consists of a set S of feasible solutions and a non-negative cost function f . The problem is to find a *globally optimal solution* $i^* \in S$, i.e. a solution with optimal cost f^* . A *neighborhood function* is a mapping $N : S \rightarrow 2^S$, which defines for each solution $i \in S$ a set $N(i) \subseteq S$ of solutions that are in some sense close to i . The set $N(i)$ is called the *neighborhood* of solution i , and each $j \in N(i)$ is called a *neighbor* of i . The simplest form of local search is called *iterative improvement*. An iterative improvement algorithm starts with an initial solution and then continuously explores neighborhoods for a solution with lower cost. If such a solution is found, then the current solution is replaced by this better solution. The procedure is continued until no better solutions can be found in the neighborhood of the current solution. By definition, iterative improvement terminates in a *local optimum*, i.e. a solution $\hat{i} \in S$ that is at least as good as all its neighbors with regard to the cost. Note that the concept of local optimality depends on the neighborhood function that is used.

For many combinatorial optimization problems one can represent solutions as sequences or collections of subsets of elements; examples are tours in the TSP, partitions in the graph partitioning problem (GPP), and schedules in the job shop scheduling problem (JSSP). These solution representations enable the use of k -change neighborhoods, where the k -change neighborhood $N(i)$ of a solution i is defined as the set of solutions that can be obtained from i by exchanging at most k ele-

ments. These k -change neighborhoods are widely applied; see Lin (1965) and Lin and Kernighan (1973) for the TSP, Kernighan and Lin (1970) for the GPP, and van Laarhoven et al. (1992) for the JSSP.

As an example we discuss the TSP. In an instance of TSP we are given n cities and an $n \times n$ -matrix (d_{pq}) , whose elements denote the distance from city p to city q for each pair p, q of cities. A tour is defined as a closed path visiting each city exactly once. The problem is to find a tour of minimal length. For this problem a solution can be written as a permutation $\pi = (\pi(1), \dots, \pi(n))$ as each permutation corresponds uniquely to a tour. The solution space is given by

$$S = \{\text{all permutations } \pi \text{ on } n \text{ cities}\}.$$

The cost function is defined as

$$f(\pi) = \sum_{i=1}^{n-1} d_{\pi(i), \pi(i+1)} + d_{\pi(n), \pi(1)}$$

that is, $f(\pi)$ gives the length of the tour corresponding to π . Furthermore, we have $|S| = (n - 1)!$

For a TSP instance, the k -change neighborhood function N_k defines for each solution i a neighborhood N_k consisting of the set of solutions that can be obtained from the given solution i by removing $k' \leq k$ edges from the tour corresponding to solution i , replacing them with k' other edges such that again a tour is obtained, and choosing the direction of the tour arbitrarily (Lin 1965; Lin and Kernighan 1973). The simplest non-trivial version of this is the 2-change neighborhood. In that case we have

$$N_2(\pi) = \{\pi' \in S \mid \pi' \text{ is obtained from } \pi \text{ by a 2-exchange}\}$$

and

$$|N_2(\pi)| = 2 + n(n - 3), \text{ for all } \pi \in S.$$

In general, local search can be viewed as a walk in a *neighborhood graph*. The node set of the neighborhood graph is given by the set of solutions and there is an arc from node i to node j if and only if j is a neighbor of i . The sequence of nodes visited by the search process defines the walk. Note that, as for the TSP, each solution j can be obtained from any other solution i by at most $n - 2$ successive 2-changes, so the 2-change neighborhood graph is strongly connected. Roughly speaking, the two main issues of a local search algorithm are the choice of the neighborhood function and the search strategy that is used. Good neighborhoods often take advantage of the combinatorial structure of the problem at hand, and are therefore typically problem dependent. A disadvantage of using iterative improvement as a search strategy is that it easily gets trapped in poor local minima. To avoid this disadvantage—while maintaining the basic principle of local search algorithms, i.e. iteration among neighboring solutions—one can consider the extension of accepting in a limited way neighboring solutions corresponding to a deterioration in the value of the cost function. This in fact is the basic idea underlying simulated annealing.

10.3 Basic Simulated Annealing

In the early 1980s Kirkpatrick et al. (1983) and independently Černý (1985) introduced the concept of annealing in combinatorial optimization. Originally this concept was heavily inspired by an analogy between the physical annealing process of solids and the problem of solving large combinatorial optimization problems. Since this analogy is quite appealing we use it here as a background for introducing simulated annealing.

In condensed matter physics, annealing is known as a thermal process for obtaining low-energy states of a solid in a heat bath. The process consists of the following two steps (Kirkpatrick et al. 1983):

- Increase the temperature of the heat bath to a maximum value at which the solid melts.
- Decrease *carefully* the temperature of the heat bath until the particles arrange themselves in the ground state of the solid.

In the liquid phase all particles arrange themselves randomly, whereas in the ground state of the solid, the particles are arranged in a highly structured lattice, for which the corresponding energy is minimal. The ground state of the solid is obtained only if the maximum value of the temperature is sufficiently high and the cooling is sufficiently slow. Otherwise the solid will be frozen into a meta-stable state rather than into the true ground state.

As far back as 1953, Metropolis et al. (1953) introduced a simple algorithm for simulating the evolution of a solid in a heat bath to thermal equilibrium. Their algorithm is based on Monte Carlo techniques (Binder 1978) and generates a sequence of states of the solid in the following way. Given a current state i of the solid with energy E_i , a subsequent state j is generated by applying a perturbation mechanism which transforms the current state into a next state by a small distortion, for instance by displacement of a particle. The energy of the next state is E_j . If the energy difference, $E_j - E_i$, is less than or equal to 0, the state j is accepted as the current state. If the energy difference is greater than 0, the state j is accepted with a probability given by

$$\exp\left(\frac{E_i - E_j}{k_B T}\right),$$

where T denotes the temperature of the heat bath and k_B a physical constant known as the Boltzmann constant. The acceptance rule described above is known as the Metropolis criterion and the algorithm that goes with it is known as the Metropolis algorithm. It is known that, if the lowering of the temperature is sufficiently slow, the solid can reach thermal equilibrium at each temperature. In the Metropolis algorithm this is achieved by generating a large number of transitions at a given temperature value. Thermal equilibrium is characterized by the Boltzmann distribution, which gives the probability of the solid of being in a state i with energy E_i at temperature T , and which is given by

$$\mathbb{P}_T\{\mathbf{X} = i\} = \frac{\exp(-E_i/k_B T)}{\sum_j \exp(-E_j/k_B T)}, \quad (10.1)$$

where \mathbf{X} is a random variable denoting the current state of the solid and the summation extends over all possible states. As we show below, the Boltzmann distribution plays an essential role in the analysis of the convergence of simulated annealing.

Returning to simulated annealing, the Metropolis algorithm can be used to generate a sequence of solutions of a combinatorial optimization problem by assuming the following equivalences between a physical many-particle system and a combinatorial optimization problem:

- Solutions in a combinatorial optimization problem are equivalent to states of a physical system.
- The cost of a solution is equivalent to the energy of a state.

Furthermore, we introduce a *control parameter* c which plays the role of the temperature. In this way simulated annealing can be viewed as an iteration of Metropolis algorithms, evaluated at decreasing values of the control parameter.

We now let go of the physical analogy and formulate simulated annealing in terms of a local search algorithm. To simplify the presentation, we assume in the remainder of this paper that we are dealing with a minimization problem. The discussion easily translates to maximization problems. Figure 10.1 describes simulated annealing in pseudo-code for an instance (S, f) of a combinatorial optimization problem and a neighborhood function N .

The meaning of the four functions in the procedure SIMULATED_ANNEALING is obvious: INITIALIZE computes a start solution and initial values of the parameters c and L , where L denotes the number of iterations at a given value of the control parameter c ; GENERATE selects a solution from the neighborhood of the current solution; CALCULATE_LENGTH and CALCULATE_CONTROL compute new values for the parameters L and c , respectively.

As already mentioned, a typical feature of simulated annealing is that, besides accepting improvements in cost, it also to a limited extent accepts deteriorations in cost. Initially, at large values of c , large deteriorations will be accepted; as c decreases, only smaller deteriorations will be accepted and finally, as the value of c approaches 0, no deteriorations will be accepted at all. Furthermore, there is no limitation on the size of a deterioration with respect to its acceptance. In simulated annealing, arbitrarily large deteriorations are accepted with positive probability; for these deteriorations the acceptance probability is small, however. This feature means that simulated annealing, in contrast to iterative improvement, can escape from local minima while it still exhibits the favorable features of iterative improvement, i.e. simplicity and general applicability.

Note that the probability of accepting deteriorations is implemented by comparing the value of $\exp((f(i) - f(j))/c)$ with a random number generated from a uniform distribution on the interval $[0,1]$. Furthermore, it should be obvious that the speed of convergence of the algorithm is determined by the choice of the parameters L_k and c_k with $k = 0, 1, \dots$, where L_k and c_k denote the values of L and

```

procedure SIMULATED_ANNEALING;
begin
    INITIALIZE ( $i_{start}, c_0, L_0$ );
     $k := 0$ ;
     $i := i_{start}$ ;
    repeat
        for  $l := 1$  to  $L_k$  do
            begin
                GENERATE ( $j$  from  $S_l$ );
                if  $f(j) \leq f(i)$  then  $i := j$ 
                else
                    if  $\exp\left(\frac{f(i)-f(j)}{c_k}\right) > \text{random}[0, 1]$  then  $i := j$ 
                end;
                 $k := k + 1$ ;
                CALCULATE_LENGTH ( $L_k$ );
                CALCULATE_CONTROL ( $c_k$ );
            until stopcriterion
        end;

```

Fig. 10.1 The simulated annealing algorithm in pseudo-code

c in iteration k of the algorithm. In the next section we will argue that under certain mild conditions on the choice of the parameters simulated annealing converges asymptotically to a globally optimal solution, and that it exhibits an equilibrium behavior from which some performance characteristics can be derived. In the subsequent section we present more practical, implementation-oriented choices of the parameter values that lead to a finite-time execution of the algorithm.

Comparing simulated annealing to iterative improvement, it is evident that simulated annealing can be viewed as a generalization. Simulated annealing becomes identical to iterative improvement in the case where the value of the control parameter is taken equal to zero. With respect to a comparison between the performance of both algorithms we mention that for most problems simulated annealing performs better than iterative improvement, repeated for a number of different initial solutions such that both algorithms have used the same computation time.

Figure 10.2 shows four solutions in the evolution of simulated annealing running on a TSP instance with 100 cities on the positions of a 10×10 grid. The initial solution at the top left is given by a random sequence among 100 cities, which is far from optimal evidently. It looks very chaotic; the corresponding value of the tour length is large. In the course of the optimization process the solutions become less

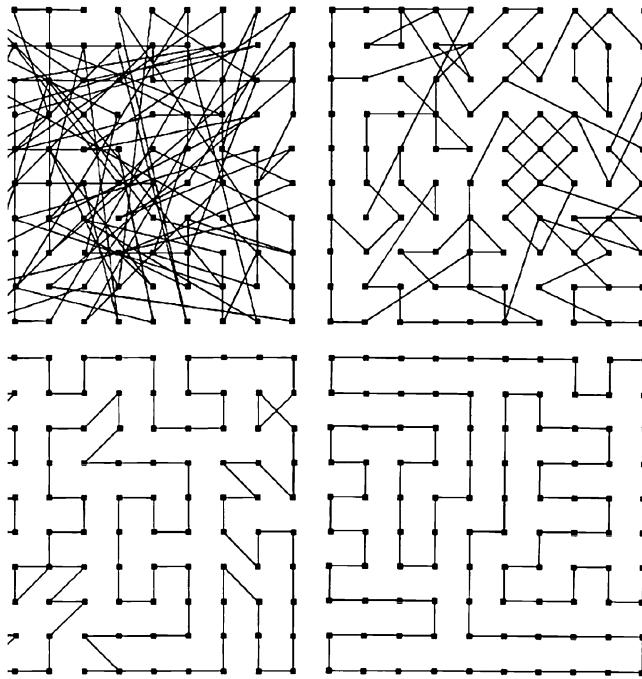


Fig. 10.2 Evolution of simulated annealing for an instance with 100 cities on a regular grid

and less chaotic (top right and bottom left), and the tour length decreases. Finally, the optimal solution shown at the bottom right is obtained. This solution has a highly regular pattern for which the tour length is minimal.

10.4 Mathematical Modeling

Simulated annealing can be mathematically modeled by means of Markov chains ([Feller 1950](#); [Isaacson and Madsen 1976](#); [Seneta 1981](#)). In this model, we view simulated annealing as a process in which a sequence of Markov chains is generated. For each Markov chain the value of the control parameter is constant, but it decreases for successive Markov chains. Each chain consists of a sequence of trials, where the outcomes of the i th trial corresponds to the solution generated in the i th iteration at the considered value of the control parameter.

Let (S, f) be a problem instance, N be a neighborhood function, and $\mathbf{X}(k)$ be a stochastic variable denoting the outcome of the k th trial. Then the *transition probability* $P_{i,j}(k)$ is the probability to make a transition from solution i to solution j at the k th trial and it is given by

$$P_{ij}(k) = \mathbb{P}\{\mathbf{X}(k) = j | \mathbf{X}(k-1) = i\}$$

$$= \begin{cases} G_{ij}(c_k)A_{ij}(c_k) & \text{if } i \neq j \\ 1 - \sum_{l \in S, l \neq i} G_{il}(c_k)A_{il}(c_k) & \text{if } i = j, \end{cases} \quad (10.2)$$

where $G_{ij}(c_k)$ denotes the *generation probability*, i.e. the probability of generating a solution j from the neighborhood of a solution i , and $A_{ij}(c_k)$ denotes the *acceptance probability*, i.e. the probability of accepting the solution j , once it is generated from solution i . The most frequently used choice for these probabilities is the following (Aarts and Korst 1989):

$$G_{ij}(c_k) = \begin{cases} |N(i)|^{-1} & \text{if } j \in S_i \\ 0 & \text{if } j \notin S_i \end{cases} \quad (10.3)$$

and

$$A_{ij}(c_k) = \begin{cases} 1 & \text{if } f(j) \leq f(i) \\ \exp((f(i) - f(j))/c) & \text{if } f(j) > f(i). \end{cases} \quad (10.4)$$

For fixed values of c , the probabilities do not depend on k , in which case the resulting Markov chain is *time-independent* or *homogeneous*. Using the theory of Markov chains it is fairly straightforward to show that, under the condition that the neighborhoods are strongly connected and not all solutions have the same cost—in which case the Markov chain is *irreducible* and *aperiodic*—there exists a unique stationary distribution of the outcomes. This distribution is the probability distribution of the solutions after an infinite number of trials. If $G_{i,j}(c) = G_{j,i}(c)$, then the distribution takes the following form (Aarts and Korst 1989).

Theorem 10.1. *Given an instance (S, f) of a combinatorial optimization problem in which not all solutions are optimal and a neighborhood function that induces a strongly connected neighborhood graph, then, after a sufficiently large number of transitions at a fixed value of c , applying the transition probabilities of (10.2)–(10.4), simulated annealing will find a solution $i \in S$ with a probability equal to*

$$\mathbb{P}_c\{\mathbf{X} = i\} \stackrel{\text{def}}{=} q_i(c) = \frac{1}{N_0(c)} \exp\left(-\frac{f(i)}{c}\right), \quad (10.5)$$

where \mathbf{X} is a stochastic variable denoting the current solution obtained by simulated annealing and

$$N_0(c) = \sum_{j \in S} \exp\left(-\frac{f(j)}{c}\right) \quad (10.6)$$

denotes a normalization constant.

A proof of this theorem is considered beyond the scope of this chapter. For those interested we refer to Michiels et al. (2007). The probability distribution of Eq. (10.5)

is called the stationary or equilibrium distribution and it is the equivalent of the Boltzmann distribution of Eq. (10.1). Next we can formulate the following important result.

Corollary 10.1. *Given an instance (S, f) of a combinatorial optimization problem and a suitable neighborhood function, and furthermore let the stationary distribution be given by Eq. (10.5), then*

$$\lim_{c \downarrow 0} q_i(c) \stackrel{\text{def}}{=} q_i^* = \frac{1}{|S^*|} \chi_{(S^*)}(i), \quad (10.7)$$

where S^* denotes the set of globally optimal solutions.¹

Proof. Using the fact that for all $a \leq 0$, $\lim_{x \downarrow 0} e^{\frac{a}{x}} = 1$ if $a = 0$, and 0 otherwise, we obtain

$$\begin{aligned} \lim_{c \downarrow 0} q_i(c) &= \lim_{c \downarrow 0} \frac{\exp\left(-\frac{f(i)}{c}\right)}{\sum_{j \in S} \exp\left(-\frac{f(j)}{c}\right)} \\ &= \lim_{c \downarrow 0} \frac{\exp\left(\frac{f^* - f(i)}{c}\right)}{\sum_{j \in S} \exp\left(\frac{f^* - f(j)}{c}\right)} \\ &= \lim_{c \downarrow 0} \frac{1}{\sum_{j \in S} \exp\left(\frac{f^* - f(j)}{c}\right)} \chi_{(S^*)}(i) \\ &\quad + \lim_{c \downarrow 0} \frac{\exp\left(\frac{f^* - f(i)}{c}\right)}{\sum_{j \in S} \exp\left(\frac{f^* - f(j)}{c}\right)} \chi_{(S \setminus S^*)}(i) \\ &= \frac{1}{|S^*|} \chi_{(S^*)}(i) + \frac{0}{|S^*|} \chi_{(S \setminus S^*)}(i), \end{aligned}$$

which completes the proof. \square

As already mentioned, the result of this corollary is important since it guarantees asymptotic convergence of the simulated annealing algorithm to the set of globally optimal solutions under the condition that the stationary distribution of Eq. (10.5) is attained at each value of c . More specifically, it implies that asymptotically optimal solutions are obtained which can be expressed as

$$\lim_{c \downarrow 0} \lim_{k \rightarrow \infty} \mathbb{P}_c \{ \mathbf{X}(k) \in S^* \} = 1.$$

¹ Let A and $A' \subset A$ be two sets. Then the characteristic function $\chi_{(A')} : A \rightarrow \{0, 1\}$ of the set A' is defined as $\chi_{(A')}(a) = 1$ if $a \in A'$, and $\chi_{(A')}(a) = 0$ otherwise.

We end this section with some remarks:

- We can also prove asymptotic convergence to optimality in the case that the constraints on the generation probabilities are weakened to the extent that they only need to induce a symmetric neighborhood graph.
- The simulated annealing algorithm can also be formulated as an *inhomogeneous algorithm*, namely as a single inhomogeneous Markov chain, where the value of the control parameter c is decreased between subsequent trials. In this case, asymptotic convergence can again be proved. However, an additional condition on the sequence $\{c_k\}$ of values of the control parameter is needed, namely

$$c_k \geq \frac{\Gamma}{\log(k+2)}, \quad k=0,1,\dots$$

for some constant Γ that can be related to the neighborhood function that is applied.

- Asymptotic estimates of the rate of convergence show that the stationary distribution of simulated annealing can only be approximated arbitrarily closely if the number of transitions is proportional to $|S|^2$. For hard problems, $|S|$ is necessarily exponential in the size of the problem instance, thus implying that approximating the asymptotic behavior arbitrarily close results in an exponential-time execution of simulated annealing. Similar results have been derived for the asymptotic convergence of the inhomogeneous algorithm.

Summarizing, simulated annealing can find optimal solutions with probability one if it is allowed an infinite number of transitions and it can get arbitrarily close to an optimal solution if at least an exponential amount of transitions is allowed. In Sect. 10.6 we show how a more efficient finite-time implementation of simulated annealing can be obtained. Evidently, this will be at the cost of the guarantee of obtaining optimal solutions. Nevertheless, practice shows that high-quality solutions can be obtained in this way.

10.5 Equilibrium Statistics

In order to enhance our understanding of the algorithm, we discuss some characteristic features of simulated annealing under the assumption that we are at equilibrium, i.e. at the stationary distribution given by Eq. (10.5). The expected cost $\mathbb{E}_c(f)$ at equilibrium is defined as

$$\begin{aligned} \mathbb{E}_c(f) &\stackrel{\text{def}}{=} \langle f \rangle_c \\ &= \sum_{i \in S} f(i) \mathbb{P}_c\{\mathbf{X} = i\} \\ &= \sum_{i \in S} f(i) q_i(c). \end{aligned} \tag{10.8}$$

Similarly, the expected squared cost $\mathbb{E}_c(f^2)$ is defined as

$$\begin{aligned}\mathbb{E}_c(f^2) &\stackrel{\text{def}}{=} \langle f^2 \rangle_c \\ &= \sum_{i \in S} f^2(i) \mathbb{P}_c\{\mathbf{X} = i\} \\ &= \sum_{i \in S} f^2(i) q_i(c).\end{aligned}\tag{10.9}$$

Using the above definitions, the variance $\text{Var}_c(f)$ of the cost is given by

$$\begin{aligned}\text{Var}_c(f) &\stackrel{\text{def}}{=} \sigma_c^2 \\ &= \sum_{i \in S} (f(i) - \mathbb{E}_c(f))^2 \mathbb{P}_c\{\mathbf{X} = i\} \\ &= \sum_{i \in S} (f(i) - \langle f \rangle_c)^2 q_i(c) \\ &= \langle f^2 \rangle_c - \langle f \rangle_c^2.\end{aligned}\tag{10.10}$$

The notation $\langle f \rangle_c$, $\langle f^2 \rangle_c$ and σ_c^2 is introduced as shorthand notation for the remainder of this paper.

Corollary 10.2. *Let the stationary distribution be given by Eq. (10.5), then the following relation holds:*

$$\frac{\partial}{\partial c} \langle f \rangle_c = \frac{\sigma_c^2}{c^2}.\tag{10.11}$$

Proof. The relation can be straightforwardly verified by using the definition of Eq. (10.8) and substituting the expression for the stationary distribution given by Eq. (10.5). \square

Corollary 10.3. *Let the stationary distribution be given by (10.5). Then we have*

$$\lim_{c \rightarrow \infty} \langle f \rangle_c \stackrel{\text{def}}{=} \langle f \rangle_\infty = \frac{1}{|S|} \sum_{i \in S} f(i)\tag{10.12}$$

$$\lim_{c \downarrow 0} \langle f \rangle_c = f^*\tag{10.13}$$

$$\lim_{c \rightarrow \infty} \sigma_c^2 \stackrel{\text{def}}{=} \sigma_\infty^2 = \frac{1}{|S|} \sum_{i \in S} (f(i) - \langle f \rangle_\infty)^2\tag{10.14}$$

and

$$\lim_{c \downarrow 0} \sigma_c^2 = 0.\tag{10.15}$$

Proof. The relations can be easily verified by using the definitions of the expected cost (10.8) and the variance (10.10), and by substituting the stationary distribution of Eq. (10.5). \square

Since $\frac{\partial}{\partial c} \langle f \rangle_c$ is strictly positive, as follows from Eq.(10.11), we get that during execution of simulated annealing the expected cost decreases monotonically—provided equilibrium is reached at each value of the control parameter—to its final value, i.e. f^* . The dependence of the stationary distribution of Eq. (10.5) on the control parameter c is the subject of the following corollary.

Corollary 10.4. *Let (S, f) denote an instance of a combinatorial optimization problem with $S^* \neq S$, and let $q_i(c)$ denote the stationary distribution associated with simulated annealing and given by (10.5). Then we have*

$$(i) \quad \forall i \in S^*$$

$$\frac{\partial}{\partial c} q_i(c) < 0$$

$$(ii) \quad \forall i \in S \setminus S^*, f(i) \geq \langle f \rangle_\infty$$

$$\frac{\partial}{\partial c} q_i(c) > 0$$

$$(iii) \quad \forall i \in S \setminus S^*, f(i) < \langle f \rangle_\infty, \exists \tilde{c}_i > 0$$

$$\begin{aligned} \frac{\partial}{\partial c} q_i(c) &< 0 & \text{if } c > \tilde{c}_i \\ &= 0 & \text{if } c = \tilde{c}_i \\ &> 0 & \text{if } c < \tilde{c}_i. \end{aligned}$$

Proof. From (10.6) we can derive the following expression:

$$\frac{\partial}{\partial c} N_0(c) = \sum_{j \in S} \frac{f(j)}{c^2} \exp\left(\frac{-f(j)}{c}\right).$$

Hence, we obtain

$$\begin{aligned} \frac{\partial}{\partial c} q_i(c) &= \frac{\partial}{\partial c} \frac{\exp\left(\frac{-f(i)}{c}\right)}{N_0(c)} \\ &= \left\{ \frac{f(i)}{c^2} \frac{\exp\left(\frac{-f(i)}{c}\right)}{N_0(c)} - \frac{\exp\left(\frac{-f(i)}{c}\right)}{N_0^2(c)} \frac{\partial}{\partial c} N_0(c) \right\} \\ &= \frac{q_i(c)}{c^2} f(i) - \frac{q_i(c)}{c^2} \frac{\sum_{j \in S} f(j) \exp\left(\frac{-f(j)}{c}\right)}{N_0(c)} \\ &= \frac{q_i(c)}{c^2} (f(i) - \langle f \rangle_c). \end{aligned} \tag{10.16}$$

Thus, the sign of $\frac{\partial}{\partial c} q_i(c)$ is determined by the sign of $f(i) - \langle f \rangle_c$ since $\frac{q_i(c)}{c^2} > 0$, for all $i \in S$ and $c > 0$.

From Eqs. (10.11) to (10.13) we have that $\langle f \rangle_c$ increases monotonically from f^* to $\langle f \rangle_\infty$ with increasing c , provided $S^* \neq S$. The remainder of the proof is now straightforward.

If $i \in S^*$ and $S \neq S^*$, then $f(i) < \langle f \rangle_c$. Hence, $\frac{\partial}{\partial c} q_i(c) < 0$ (see Eq. (10.16)), which completes the proof of part (i).

If $i \notin S^*$, then the sign of $\frac{\partial}{\partial c} q_i(c)$ depends on the value of $\langle f \rangle_c$. Hence, using (10.16), we have that $\forall i \in S \setminus S^* : \frac{\partial}{\partial c} q_i(c) > 0$ if $f(i) \geq \langle f \rangle_\infty$, whereas $\forall i \in S \setminus S^*$, where $f(i) < \langle f \rangle_\infty$, there exists a $\tilde{c}_i > 0$ at which $f(i) - \langle f \rangle_c$ changes sign. Consequently, we have

$$\begin{aligned}\frac{\partial}{\partial c} q_i(c) &< 0 \text{ if } c > \tilde{c}_i \\ &= 0 \text{ if } c = \tilde{c}_i \\ &> 0 \text{ if } c < \tilde{c}_i.\end{aligned}$$

This completes the proofs of parts (ii) and (iii). \square

From Corollary 10.4 it follows that the probability of finding an optimal solution increases monotonically with decreasing c . Furthermore, for each solution, not being an optimal one, there exists a positive value of the control parameter \tilde{c}_i , such that for $c < \tilde{c}_i$, the probability of finding that solution decreases monotonically with decreasing c .

We conclude this section with some results that illustrate some of the elements discussed in the analysis presented above. For this we need the definition of the *acceptance ratio* $\omega(c)$ which is defined as

$$\omega(c) = \left. \frac{\text{number of accepted transitions}}{\text{number of proposed transitions}} \right|_c. \quad (10.17)$$

Figure 10.3 shows the behavior of the acceptance ratio as a function of the value of the control parameter for typical implementations of simulated annealing. The figure illustrates the behavior as it would be expected from the acceptance criterion given in Eq. (10.4). At large values of c , virtually all proposed transitions are accepted. As c decreases, ever fewer proposed transitions are accepted, and finally, at very small values of c , no proposed transitions are accepted at all.

Figure 10.4 shows the typical behavior of (a) the normalized average cost and (b) the normalized spread of the cost for simulated annealing as a function of the control parameter c . The typical behavior shown in this figure is observed for many different problem instances and is reported in the literature by a number of authors (Aarts et al. 1988; Hajek 1985; Kirkpatrick et al. 1983; van Laarhoven and Aarts 1987; White 1984).

From the figures we can deduce some characteristic features of the expected cost $\langle f \rangle_c$ and the variance σ_c^2 of the cost. First, it is observed that for large values of c the average and the spread of the cost are about constant and equal to $\langle f \rangle_\infty$ and σ_∞ ,

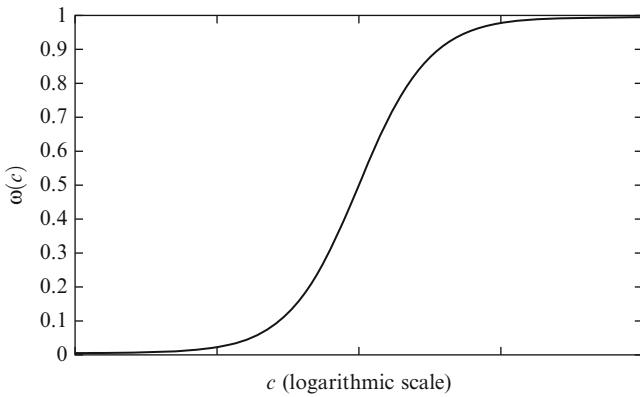


Fig. 10.3 Acceptance ratio as function of the control parameter

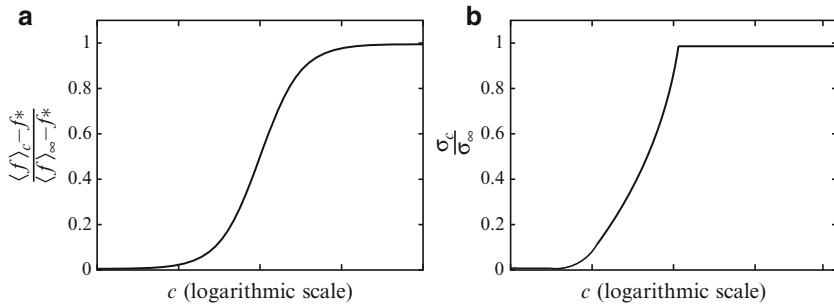


Fig. 10.4 (a) Normalized average value $|\langle f \rangle_c - f^*| / |\langle f \rangle_\infty - f^*|$, and (b) normalized spread σ_c / σ_∞ of the cost function, as a function of the control parameter

respectively. This behavior is directly explained by Eqs. (10.12) and (10.14), from which it follows that both the average value and the spreading of the cost function are constant at large c -values.

Secondly, we observe that there exists a threshold value c_t of the control parameter for which

$$\langle f \rangle_{c_t} \approx \frac{1}{2}(\langle f \rangle_\infty + f^*) \quad (10.18)$$

and

$$\begin{aligned} \sigma_c^2 &\approx \sigma_\infty^2 & \text{if } c \geq c_t \\ &< \sigma_\infty^2 & \text{if } c < c_t. \end{aligned} \quad (10.19)$$

Moreover, we mention that c_t is roughly the value of c for which $\omega(c) \approx 0.5$.

10.6 Practical Application

A finite-time implementation of simulated annealing is obtained by generating a sequence of homogeneous Markov chains of finite length at descending values of the control parameter. A *cooling schedule* specifies a finite sequence of values of the control parameter, and a finite number of transitions at each value of the control parameter. More precisely, it is specified by

- An *initial value* of the control parameter c_0
- A *decrement function* for lowering the value of the control parameter
- A *final value* of the control parameter specified by a *stop criterion* and
- A finite *length* of each homogeneous Markov chain.

The search for adequate cooling schedules has been the subject of many studies over the years. Reviews are given by [van Laarhoven and Aarts \(1987\)](#), [Collins et al. \(1988\)](#), and [Romeo and Sangiovanni-Vincentelli \(1991\)](#). Below we discuss some results.

Most of the existing work on cooling schedules presented in the literature deals with heuristic schedules. We distinguish between two broad classes: static and dynamic schedules. In a static cooling schedule the parameters are fixed; they cannot be changed during execution of the algorithm. In a dynamic cooling schedule the parameters are adaptively changed during execution of the algorithm. Below we present some examples.

10.6.1 Static Cooling Schedules

The following simple schedule is known as the geometric schedule. It originates from the early work on cooling schedules by [Kirkpatrick et al. \(1983\)](#), and is still used in many practical situations.

10.6.1.1 Initial Value of the Control Parameter

To ensure a sufficiently large value of $\omega(c_0)$, one may choose $c_0 = \Delta f_{\max}$, where Δf_{\max} is the maximal difference in cost between any two neighboring solutions. Exact calculation of Δf_{\max} is quite time consuming in many cases. However, one often can give simple estimates of its value.

10.6.1.2 Lowering the Control Parameter Value

A frequently used decrement function is given by

$$c_{k+1} = \alpha \cdot c_k, \quad k = 0, 1, \dots$$

where α is a positive constant smaller than but close to 1. Typical values lie between 0.8 and 0.99.

10.6.1.3 Final Value of the Control Parameter

The final value is fixed at some small value, which may be related to the smallest possible difference in cost between two neighboring solutions.

10.6.1.4 Markov Chain Length

The length of Markov chains is fixed by some number that may be related to the size of the neighborhoods in the problem instance at hand.

10.6.2 Dynamic Cooling Schedules

There exist many extensions of the simple static schedule presented above that lead to a dynamic schedule. For instance, a sufficiently large value of c_0 may be obtained by requiring that the initial acceptance ratio $\omega(c_0)$ is close to 1. This can be achieved by starting off at a small positive value of c_0 and multiplying it with a constant factor, larger than 1, until the corresponding value of $\omega(c_0)$, which is calculated from a number of generated transitions, is close to 1. Typical values of $\omega(c_0)$ lie between 0.9 and 0.99. An adaptive calculation of the final value of the control parameter may be obtained by terminating the execution of the algorithm at a c_k -value for which the value of the cost function of the solution obtained in the last trial of a Markov chain remains unchanged for a number of consecutive chains. Clearly such a value exists for each local minimum that is found. The length of Markov chains may be determined by requiring that at each value c_k , a minimum number of transitions is accepted. However, since transitions are accepted with decreasing probability, one would obtain $L_k \rightarrow \infty$ for $c_k \downarrow 0$. Therefore, L_k is usually bounded by some constant L_{\max} to avoid extremely long Markov chains for small values of c_k .

In addition to this basic dynamic schedule the literature presents a number of more elaborate schedules. Most of these schedules are based on a statistical analysis of simulated annealing using the equilibrium statistics of the previous section.

10.7 Tricks of the Trade

To apply simulated annealing in practice, three basic ingredients are needed: a concise problem representation, a neighborhood and a cooling schedule. The algorithm is usually implemented as a sequence of homogeneous Markov chains of finite length, generated at descending values of the control parameter. This is specified by the cooling schedule. As for the choice of the cooling schedule, we have seen in the previous section that there exist some general guidelines. However, for the other ingredients no general rules are known that guide their choice. The way

they are handled is still a matter of experience, taste and skill left to the annealing practitioner, and we expect that this will not change in the near future.

Ever since its introduction in 1983, simulated annealing has been applied to a large number of different combinatorial optimization problems in areas as diverse as operations research, VLSI design, code design, image processing and molecular physics. The success of simulated annealing can be characterized by the following elements:

- Performance, i.e. running time and solution quality
- Ease of implementation and
- Applicability and flexibility.

With respect to the last two items we make the following remarks. It is apparent that simulated annealing is conceptually simple and quite easy to implement. Implementation of the algorithm typically takes only a few hundred lines of computer code. Experience shows that implementations for new problems often take only a few days and in most cases existing programs, written for another problem, can be efficiently used.

With respect to applicability and flexibility it has become obvious as a result of the overwhelming amount of practical experience that has been gathered over the past 30 years that simulated annealing can be considered as one of the most flexible and applicable algorithms that exist. However, one must bear in mind that it is not always trivial to apply the algorithm to a given problem. Finding appropriate neighborhoods requires problem insight, and sometimes it is necessary to reformulate the problem or transform it into an equivalent or similar problem, before simulated annealing can be applied successfully; an example is graph coloring ([Michiels et al. 2007](#)).

With respect to performance, one typically trades solution quality against running time. Performance analyses of simulated annealing algorithms have been the subject of many studies. Despite numerous studies it is still difficult to judge simulated annealing on its true merits. This is predominantly due to the fact that many of these studies lack the depth required to draw reliable conclusions; for example, results are often limited to one single run of the algorithm, instead of taking the average over a number of runs; the applied cooling schedules are often too simple, and do not get the best out of the algorithm; results are often not compared to the results obtained with other (tailored) algorithms.

We conclude this section with two remarks.

Comparing simulated annealing to time-equivalent iterative improvement using the same neighborhood function, i.e. repeating iterative improvement with different initial solutions for an equally long time as the running time of simulated annealing and keeping the best solution, reveals that simulated annealing performs substantially better (smaller error). This difference becomes even more pronounced for larger problem instances ([van Laarhoven et al. 1992; van Laarhoven 1988](#)).

Finally, experience shows that the performance of simulated annealing depends as much on the skill and effort that is applied to the implementation as on the algorithm itself. For instance, the choice of an appropriate neighborhood function,

of an efficient cooling schedule, and of sophisticated data structures allowing fast manipulations can substantially reduce the error as well as the running time. Thus, in view of this and considering the simple nature of annealing, there lies a challenge in constructing efficient and effective implementations of simulated annealing.

10.8 Conclusions

Since its introduction in 1983, simulated annealing has been applied to many different problems in many different areas. Thirty years of experience has led to the following general observations:

- High-quality solutions can be obtained but sometimes at the cost of large amounts of computation time.
- In many practical situations, where no tailored algorithms are available, the algorithm is a real boon due to its general applicability and its ease of implementation.

So, simulated annealing is an algorithm that every practical mathematician and computer scientist should have in his toolbox.

Sources of Additional Information

Introductory textbooks describing both theoretical and practical issues of simulated annealing are given by [Aarts and Korst \(1989\)](#), [van Laarhoven and Aarts \(1987\)](#), and [Michiels et al. \(2007\)](#). [Salamon et al. \(2002\)](#) present a basic textbook on simulated annealing with improvements for practical implementations and references to software tools. [Azencott \(1992\)](#) presents a theoretical textbook on parallelization techniques for simulated annealing for the purpose of speeding up the algorithm through effective parallel implementations.

Early proofs of the asymptotic convergence of the homogeneous Markov model for simulated annealing are presented by [Aarts and van Laarhoven \(1985\)](#) and [Lundy and Mees \(1986\)](#). Proofs for the inhomogeneous algorithm have been published by [Connors and Kumar \(1987\)](#), [Gidas \(1985\)](#), and [Mitra et al. \(1986\)](#). [Hajek \(1988\)](#) was the first to present necessary and sufficient conditions for asymptotic convergence of the inhomogeneous model. [Anily and Federgruen \(1987\)](#) present theoretical results on the convergence of simulated annealing for a set of acceptance probabilities that are much more general than the classical Metropolis acceptance probabilities. [Villalobos-Arias et al. \(2006\)](#) prove asymptotic convergence of simulated annealing when applied to multi-objective optimization problems. A comprehensive review of the theory of simulated annealing is given by [Romeo and Sangiovanni-Vincentelli \(1991\)](#).

[Strenski and Kirkpatrick \(1991\)](#) present an early analysis of the finite-time behavior of simulated annealing for various cooling schedules. [Steinhöfel et al.](#)

(1998) present a comparative study in which they investigate the performance of simulated annealing for different cooling schedules when applied to job shop scheduling. Nourani and Andersen (1998) present a comparative study in which they investigate the performance of simulated annealing with cooling schedules applying different types of decrement functions for lowering the value of the control parameter. Andersen (1996) elaborates on the thermodynamical analysis of finite-time implementations of simulated annealing. Orosz and Jacobson (2002) study the finite-time behavior of a special variant of simulated annealing in which the values of the control parameters are kept constant during the annealing process. Park and Kim (1998) present a systematic approach to the problem of choosing appropriate values for the parameters in a cooling schedule.

Vidal (1993) presents an edited collection of papers on practical aspects of simulated annealing, ranging from empirical studies of cooling schedules up to implementation issues of simulated annealing for problems in engineering and planning. Eglese (1990) presents a survey of the application of simulated annealing to problems in operations research. Collins et al. (1988) present an annotated bibliography with more than a thousand references to papers on simulated annealing. It is organized in two parts; one on theory, and the other on applications. The applications range from graph-theoretic problems to problems in engineering, biology and chemistry. Fox (1993) discusses the integration of simulated annealing with other local search heuristics such as tabu search and genetic algorithms.

References

- Aarts EHL, Korst JHM (1989) Simulated annealing and Boltzmann machines. Wiley, Chichester
- Aarts EHL, van Laarhoven PJM (1985) Statistical cooling: a general approach to combinatorial optimization problems. Philips J Res 40:193–226
- Aarts EHL, Lenstra JK (eds) (2003) Local search in combinatorial optimization. Princeton University Press, Princeton
- Aarts EHL, Korst JHM, van Laarhoven PJM (1988) A quantitative analysis of the simulated annealing algorithm: a case study for the traveling salesman problem. J Stat Phys 50:189–206
- Andersen B (1996) Finite-time thermodynamics and simulated annealing. In: Shiner JS (ed) Entropy and entropy generation. Kluwer, Dordrecht, pp 111–127
- Anily S, Federgruen A (1987) Simulated annealing methods with general acceptance probabilities. J Appl Probab 24:657–667
- Arora S, Barak B (2009) Computational complexity: a modern approach. Cambridge University Press, Cambridge/New York
- Ausiello G, Crescenzi P, Gambosi G, Kann V, Marchetti-Spaccamela A, Protasi M (1999) Complexity and approximation: combinatorial optimization problems and their approximability properties. Springer, Berlin

- Azencott R (1992) Simulated annealing: parallelization techniques. Wiley, Chichester
- Binder K (1978) Monte Carlo methods in statistical physics. Springer, Berlin
- Černý V (1985) Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *J Optim Theory Appl* 45:41–51
- Collins NE, Eglese RW, Golden BL (1988) Simulated annealing: an annotated bibliography. *Am J Math Manage Sci* 8:209–307
- Connors DP, Kumar PR (1987) Simulated annealing and balance of recurrence order in time-inhomogeneous Markov chains. In: Proceedings of the 26th IEEE conference on decision and control, Los Angeles, pp 2261–2263
- Eglese RW (1990) Simulated annealing: a tool for operational research. *Eur J Oper Res* 46:271–281
- Feller W (1950) An introduction to probability theory and its applications, vol 1. Wiley, New York
- Fox BL (1993) Integrating and accelerating tabu search, simulated annealing, and genetic algorithms. In: Glover F et al (eds) Tabu search. Annals of operations research, vol 41. Baltzer, Basel, pp 47–67
- Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco
- Gidas B (1985) Nonstationary Markov chains and convergence of the annealing algorithm. *J Stat Phys* 39:73–131
- Hajek B (1985) A tutorial survey of the theory and application of simulated annealing. In: Proceedings of the 24th IEEE conference on decision and control, Fort Lauderdale, pp 755–760
- Hajek B (1988) Cooling schedules for optimal annealing. *Math Oper Res* 13:311–329
- Isaacson D, Madsen R (1976) Markov chains. Wiley, New York
- Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. *Bell Syst Tech J* 49:291–307
- Kirkpatrick S, Gelatt CD Jr, Vecchi MP (1983) Optimization by simulated annealing. *Science* 220:671–680
- Lin S (1965) Computer solutions of the traveling salesman problem. *Bell Syst Tech J* 44:2245–2269
- Lin S, Kernighan BW (1973) An effective heuristic algorithm for the traveling salesman problem. *Oper Res* 21:498–516
- Lundy M, Mees A (1986) Convergence of an annealing algorithm. *Math Program* 34:111–124
- Metropolis N, Rosenbluth A, Rosenbluth M, Teller A, Teller E (1953) Equation of state calculations by fast computing machines. *J Chem Phys* 21:1087–1092
- Michiels W, Aarts E, Korst J (2007) Theoretical aspects of local search. Springer, Berlin
- Mitra D, Romeo F, Sangiovanni-Vincentelli AL (1986) Convergence and finite-time behavior of simulated annealing. *Adv Appl Probab* 18:747–771
- Nourani Y, Andersen B (1998) A comparison of simulated annealing cooling strategies. *J Phys A* 31:8373–8385

- Orosz JE, Jacobson SH (2002) Finite-time performance analysis of static simulated annealing algorithms. *Comput Optim Appl* 21:21–53
- Park M-W, Kim Y-D (1998) A systematic procedure for setting parameters in simulated annealing algorithms. *Comput Oper Res* 25:207–217
- Romeo F, Sangiovanni-Vincentelli A (1991) A theoretical framework for simulated annealing. *Algorithmica* 6:302–345
- Salamon P, Sibani P, Frost R (2002) Facts, conjectures, and improvements for simulated annealing. SIAM Monographs, Philadelphia
- Seneta E (1981) Non-negative matrices and Markov chains, 2nd edn. Springer, New York
- Steinhöfel K, Albrecht A, Wong CK (1998) On various cooling schedules for simulated annealing applied to the job shop problem. In: Luby M et al (eds) Randomization and approximation techniques in computer science. Lecture notes in computer science, vol 1518. Springer, Berlin, pp 260–279
- Strenski PN, Kirkpatrick S (1991) Analysis of finite length annealing schedules. *Algorithmica* 6:346–366
- van Laarhoven PJM (1988) Theoretical and computational aspects of simulated annealing. PhD thesis, Erasmus University Rotterdam
- van Laarhoven PJM, Aarts EHL (1987) Simulated annealing: theory and applications. Reidel, Dordrecht
- van Laarhoven PJM, Aarts EHL, Lenstra JK (1992) Job shop scheduling by simulated annealing. *Oper Res* 40:185–201
- Vidal RVV (ed) (1993) Applied simulated annealing. Lecture notes in economics and mathematical systems, vol 396. Springer, Berlin
- Villalobos-Arias M, Coello CA, Hernandez-Lerma O (2006) Asymptotic convergence of a simulated annealing algorithm for multiobjective optimization problems. *Math Methods Oper Res* 64:353–362
- White SR (1984) Concepts of scale in simulated annealing. In: Proceedings of the IEEE international conference on computer design, New York, pp 646–651

Chapter 11

GRASP: Greedy Randomized Adaptive Search Procedures

Mauricio G.C. Resende and Celso C. Ribeiro

11.1 Introduction

Metaheuristics are general high-level procedures that coordinate simple heuristics and rules to find good-quality solutions to computationally difficult combinatorial optimization problems. Among them, we find simulated annealing (see Chap. 10), tabu search (see Chap. 9), genetic algorithms (Chap. 4), scatter search (Chap. 5), variable neighborhood search (Chap. 12), ant colonies (Chap. 8), and others. The method described in this chapter represents another example of such a technique. Metaheuristics are based on distinct paradigms and offer different mechanisms to escape from locally optimal solutions. They are among the most effective solution strategies for solving combinatorial optimization problems in practice and have been applied to a wide array of academic and real-world problems. The customization (or instantiation) of a metaheuristic to a given problem yields a *heuristic* for that problem.

In this chapter, we consider the combinatorial optimization problem of minimizing $f(S)$ over all solutions $S \in X$, which is defined by a finite set $E = \{e_1, \dots, e_n\}$ (called the ground set), by a set of feasible solutions $X \subseteq 2^E$ and by an objective function $f : 2^E \rightarrow \mathbb{R}$. The ground set E , the objective function f , and the constraints defining the set of feasible solutions X are specific for each problem. We seek an optimal solution $S^* \in X$ such that $f(S^*) \leq f(S)$, $\forall S \in X$.

GRASP, which stands for greedy randomized adaptive search procedures (Feo and Resende 1989, 1995), is a multistart, or iterative, metaheuristic in which

M.G.C. Resende (✉)

Algorithms and Optimization Research Department, AT&T Labs Research,
Florham Park, NJ, USA

e-mail: mgcr@research.att.com

C.C. Ribeiro

Department of Computer Science, Universidade Federal Fluminense, Niterói, RJ, Brazil
e-mail: cels0@ic.uff.br

each iteration consists of two phases: construction and local search. The construction phase builds a solution. If this solution is not feasible, a repair procedure should be applied to attempt to achieve feasibility. If feasibility cannot be reached, it is discarded and a new solution is created. Once a feasible solution is obtained, its neighborhood is investigated until a local minimum is found during the local search phase. The best overall solution is kept as the result.

The principles and building blocks of GRASP, which are also common to other metaheuristics, are reviewed in Sect. 11.2. A template for the basic GRASP algorithm is described in Sect. 11.3. The GRASP with path-relinking heuristic is considered in Sect. 11.4, where different strategies for the efficient implementation of path-relinking are discussed. Hybridizations of GRASP with data mining and other metaheuristics are reviewed in Sect. 11.5. Recommendations and good problem-solving practices leading to more efficient implementations are presented in Sect. 11.6. Finally, we suggest sources of additional information, with references and links to literature surveys, annotated bibliographies and source codes, tools and software for algorithm evaluation and comparison, and accounts of applications and parallel implementations of GRASP.

11.2 Principles and Building Blocks

Several principles and building blocks appear as components common to GRASP and other metaheuristics. They are often blended using different strategies and additional features that distinguish one metaheuristic from another.

11.2.1 Greedy Algorithms

In a greedy algorithm, solutions are progressively built from scratch. At each iteration, a new element from the ground set E is incorporated into the partial solution under construction, until a complete feasible solution is obtained. The selection of the next element to be incorporated is determined by the evaluation of all candidate elements according to a *greedy evaluation function*. This greedy function usually represents the incremental increase in the cost function due to the incorporation of this element into the partial solution under construction. The greediness criterion establishes that an element with the smallest incremental increase is selected, with ties being arbitrarily broken. Figure 11.1 provides a template for a greedy algorithm for a minimization problem.

The solutions obtained by greedy algorithms are not necessarily optimal. Greedy algorithms are often used to build initial solutions to be explored by local search or metaheuristics.

```

procedure GreedyAlgorithm
1.  $S \leftarrow \emptyset$ ;
2. Initialize the candidate set:  $C \leftarrow E$ ;
3. Evaluate the incremental cost  $c(e)$  for all  $e \in C$ ;
4. while  $C \neq \emptyset$  do
5.     Select an element  $s \in C$  with the smallest incremental cost  $c(s)$ ;
6.     Incorporate  $s$  into the current solution:  $S \leftarrow S \cup \{s\}$ ;
7.     Update the candidate set  $C$ ;
8.     Reevaluate the incremental cost  $c(e)$  for all  $e \in C$ ;
9. end;
10. return  $S$ ;
end.

```

Fig. 11.1 Greedy algorithm for minimization

```

procedure GreedyRandomizedAlgorithm(Seed)
1.  $S \leftarrow \emptyset$ ;
2. Initialize the candidate set:  $C \leftarrow E$ ;
3. Evaluate the incremental cost  $c(e)$  for all  $e \in C$ ;
4. while  $C \neq \emptyset$  do
5.     Build a list with the candidate elements having the smallest incremental costs;
6.     Select an element  $s$  from the restricted candidate list at random;
7.     Incorporate  $s$  into the solution:  $S \leftarrow S \cup \{s\}$ ;
8.     Update the candidate set  $C$ ;
9.     Reevaluate the incremental cost  $c(e)$  for all  $e \in C$ ;
10. end;
11. return  $S$ ;
end.

```

Fig. 11.2 Greedy randomized algorithm for minimization

11.2.2 Randomization and Greedy Randomized Algorithms

Randomization plays a very important role in algorithm design. Metaheuristics such as simulated annealing, GRASP and genetic algorithms rely on randomization to sample the search space. Randomization can also be used to break ties, enabling different trajectories to be followed from the same initial solution in multistart methods, or sampling different parts of large neighborhoods. One particularly important use of randomization appears in the context of greedy algorithms.

Greedy randomized algorithms are based on the same principle guiding pure greedy algorithms. However, they make use of randomization to build different solutions at different runs. Figure 11.2 illustrates the pseudo-code of a greedy randomized algorithm for minimization. At each iteration, the set of candidate elements is formed by all elements that can be incorporated into the partial solution under construction without destroying feasibility. As before, the selection of the next element is determined by the evaluation of all candidate elements according to a greedy evaluation function. The evaluation of the elements by this function leads

to the creation of a restricted candidate list (RCL) formed by the best elements, i.e. those whose incorporation into the current partial solution results in the smallest incremental costs. The element to be incorporated into the partial solution is randomly selected from those in the RCL. Once the selected element has been incorporated into the partial solution, the set of candidate elements is updated and the incremental costs are re-evaluated.

Greedy randomized algorithms are used for a variety of purposes. For example, they are used in the construction phase of GRASP heuristics or to create initial solutions for population-based metaheuristics such as genetic algorithms or scatter search. Randomization is also a major component of metaheuristics, such as simulated annealing and VNS, in which a solution in the neighborhood of the current solution is randomly generated at each iteration.

11.2.3 Neighborhoods

A *neighborhood* of a solution S is a set $N(S) \subseteq X$. Each solution $S' \in N(S)$ is reached from S by an operation called a *move*. Normally, two neighbor solutions S and $S' \in N(S)$ differ by only a few elements. Neighborhoods may also eventually contain infeasible solutions not in X .

A solution S^* is a local optimum with respect to a given neighborhood N if $f(S^*) \leq f(S), \forall S \in N(S^*)$. Local search methods are based on the exploration of solution neighborhoods in an iterative fashion by successively searching for improving solutions until a local optimum is found.

The definition of a neighborhood is not unique. Some implementations of metaheuristics make use of multiple neighborhood structures. A metaheuristic may also modify the neighborhood, by excluding some of the possible moves and introducing others. Such modifications might also require changes in the nature of solution evaluation. The strategic oscillation approach (Glover 1996) illustrates this intimate relationship between changes in neighborhood and changes in evaluation.

11.2.4 Local Search

Solutions generated by greedy algorithms are not necessarily optimal, even with respect to simple neighborhoods. A local search technique attempts to improve solutions in an iterative fashion, by successively replacing the current solution by a better solution in a neighborhood of the current solution. It terminates when no better solution is found in the neighborhood. The pseudo-code of a basic local search algorithm for a minimization problem is given in Fig. 11.3. It starts from a solution S and makes use of a neighborhood structure N .

```

procedure LocalSearch( $S$ )
1. while  $S$  is not locally optimal do
2.   Find  $S' \in N(S)$  with  $f(S') < f(S)$ ;
3.    $S \leftarrow S'$ ;
4. end;
5. return  $S$ ;
end.

```

Fig. 11.3 Local search algorithm for minimization

The effectiveness of a local search procedure depends on several aspects, such as the neighborhood structure, the neighborhood search technique, the speed of evaluation of the cost function, and the starting solution. The neighborhood search may be implemented using either a best-improving or a first-improving strategy. In the case of a *best-improving* strategy, all neighbors are investigated and the current solution is replaced by the best neighbor. In the case of a *first-improving* strategy, the current solution moves to the first neighbor whose cost function value is smaller than that of the current solution.

11.2.5 Restricted Neighborhoods and Candidate Lists

Glover and Laguna (1997) point out that the use of strategies to restrict neighborhoods and to create candidate lists is essential to restrict the number of solutions examined in a given iteration in situations where the neighborhoods are very large or their elements are expensive to evaluate.

Their goal consists in attempting to isolate regions of the neighborhood containing desirable features and inserting them into a list of candidates for close examination. The efficiency of candidate list strategies can be enhanced by the use of memory structures for efficient updates of move evaluations from one iteration to another. The effectiveness of a candidate list strategy should be evaluated in terms of the quality of the best solution found in some specified amount of computation time. Strategies such as aspiration plus, elite candidate list, successive filtering, sequential fan candidate list, and bounded change candidate list are reviewed in Glover and Laguna (1997).

Ribeiro and Souza (2000) used a candidate list strategy, based on quickly computed estimates of move values, to significantly speed up the search for the best neighbor in their tabu search heuristic for the Steiner problem in graphs. Moves with bad estimates were discarded. Restricted neighborhoods based on filtering out unpromising solutions with high evaluations are discussed, for example, in Martins et al. (1999) and Resende and Ribeiro (2003b).

11.2.6 Intensification and Diversification

Two important components of metaheuristics are intensification and diversification:

- *Intensification* strategies encourage move combinations and solution features historically found to be good or to return to explore attractive regions of the solution space more thoroughly. The implementation of intensification strategies enforces the investigation of neighborhoods of elite solutions and makes use of explicit memory to do so. Intensification is often implemented in GRASP heuristics by using path-relinking, as described below.
- *Diversification* strategies encourage the search to examine unvisited regions of the solution space or to generate solutions that significantly differ from those previously visited. Penalty and incentive functions are often used in this context. Diversification is often implemented by means of perturbations which destroy the structure of the current solution. In the context of GRASP, they are used, for example, within hybridizations with the iterated local search (ILS) metaheuristic, as described in Sect. 11.5.

11.2.7 Path-Relinking

Path-relinking was originally proposed by [Glover \(1996\)](#) as an intensification strategy exploring trajectories connecting elite solutions obtained by tabu search or scatter search ([Glover et al. 2000](#)). Starting from one or more elite solutions, paths in the solution space leading toward other elite solutions are generated and explored in the search for better solutions. To generate paths, moves are selected to introduce attributes in the current solution that are present in the elite guiding solution. Path-relinking may be viewed as a strategy that seeks to incorporate attributes of high-quality solutions, by favoring these attributes in the selected moves.

The algorithm in Fig. 11.4 illustrates the pseudo-code of the path-relinking procedure applied to a pair of solutions S_s (starting solution) and S_t (target solution). The procedure starts by computing the symmetric difference $\Delta(S_s, S_t)$ between the two solutions, i.e. the set of elements of the ground set E that appear in one of them but not in the other. The symmetric difference also defines the set of moves that have to be successively applied to S_s until S_t is reached. At each step, the procedure examines all moves $m \in \Delta(S_s, S_t)$ from the current solution S and selects the one which results in the least cost solution, i.e. the one which minimizes $f(S \oplus m)$, where $S \oplus m$ is the solution resulting from applying move m to solution S . The best move m^* is made, producing solution $S \oplus m^*$. The set of available moves is updated. If necessary, the best solution \bar{S} is updated. The procedure terminates when S_t is reached, i.e. when $\Delta(S, S_t) = \emptyset$. A path of solutions is thus generated linking S_s to S_t and \bar{S} is the best solution in this path. Since there is no guarantee that \bar{S} is a local minimum, local search can be applied to it and the resulting local minimum is returned by the algorithm.

```

procedure PathRelinking( $S_s, S_t$ )
1. Compute the symmetric difference  $\Delta(S_s, S_t)$ ;
2.  $\bar{f} \leftarrow \min\{f(S_s), f(S_t)\}$ ;
3.  $\bar{S} \leftarrow \operatorname{argmin}\{f(S_s), f(S_t)\}$ ;
4.  $S \leftarrow S_s$ ;
5. while  $\Delta(S, S_t) \neq \emptyset$  do
6.    $m^* \leftarrow \operatorname{argmin}\{f(S \oplus m) : m \in \Delta(S, S_t)\}$ ;
7.    $\Delta(S \oplus m^*, S_t) \leftarrow \Delta(S, S_t) \setminus \{m^*\}$ ;
8.    $S \leftarrow S \oplus m^*$ ;
9.   if  $f(S) < \bar{f}$  then
10.     $\bar{f} \leftarrow f(S)$ ;
11.     $\bar{S} \leftarrow S$ ;
12.   end_if;
13. end_while;
14.  $\bar{S} \leftarrow \text{LocalSearch}(\bar{S})$ ;
15. return  $\bar{S}$ ;
end.

```

Fig. 11.4 Path-relinking procedure for minimization

Path-relinking may also be viewed as a constrained local search strategy applied to the initial solution S_s , in which only a limited set of moves can be performed and uphill moves are allowed. Several alternatives have been considered and combined in successful implementations of path-relinking in conjunction with GRASP and other metaheuristics. They are reviewed in Sect. 11.4.

11.3 A Template for GRASP

Each iteration of the original GRASP metaheuristic proposed in Feo and Resende (1989) may be divided in two main phases: construction and local search (see also Feo and Resende 1995; Resende 2008; Resende and Ribeiro 2003a,b, 2005a, 2010 for other surveys on GRASP and its extensions). These steps are repeated many times, characterizing a multistart metaheuristic. The construction phase builds a solution. If this solution is not feasible, it is either discarded or a repair heuristic is applied to achieve feasibility (examples of repair procedures can be found in Duarte et al. 2007a,b; Mateus et al. 2011; Nascimento et al. 2010). Once a feasible solution is obtained, its neighborhood is investigated until a local minimum is found during the local search phase. The best solution found over all iterations is returned.

The pseudo-code in Fig. 11.5 illustrates the main blocks of a GRASP procedure for minimization, in which `MaxIterations` iterations are performed and `Seed` is used as the initial seed for the pseudo-random number generator.

An especially appealing characteristic of GRASP is the ease with which it can be implemented. Few parameters need to be set and tuned, and therefore development can focus on implementing efficient data structures to assure quick iterations. Basic implementations of GRASP rely exclusively on two parameters: the number

```

procedure GRASP(MaxIterations,Seed)
1. Set  $f^* \leftarrow \infty$ ;
2. for  $k = 1, \dots, \text{MaxIterations}$  do
3.    $S \leftarrow \text{GreedyRandomizedAlgorithm}(\text{Seed})$ ;
4.   if  $S$  is not feasible then
5.      $S \leftarrow \text{RepairSolution}(S)$ ;
6.   end;
7.    $S \leftarrow \text{LocalSearch}(S)$ ;
8.   if  $f(S) < f^*$  then
9.      $S^* \leftarrow S$ ;
10.     $f^* \leftarrow f(S)$ ;
11.   end;
12. end;
13. return  $S^*$ ;
end.

```

Fig. 11.5 Template of a GRASP heuristic for minimization

MaxIterations of iterations and the parameter used to limit the size of the RCL within the greedy randomized algorithm used by the construction phase. In spite of its simplicity and ease of implementation, GRASP is a very effective metaheuristic and produces the best known solutions for many problems, see [Festa and Resende \(2002, 2009a,b\)](#) for extensive surveys of applications of GRASP.

For the construction of the RCL used in the first phase, we consider, without loss of generality, a minimization problem such as the one formulated in Sect. 11.1. As before, we denote by $c(e)$ the incremental cost associated with the incorporation of element $e \in E$ into the solution under construction. At any GRASP iteration, let c^{\min} and c^{\max} be, respectively, the smallest and the largest incremental costs.

The RCL is made up of the elements $e \in E$ with the best (i.e. the smallest) incremental costs $c(e)$. This list can be limited either by the number of elements (cardinality-based) or by their quality (value-based). In the first case, it is made up of the p elements with the best incremental costs, where p is a parameter. In this chapter, the RCL is associated with a threshold parameter $\alpha \in [0, 1]$. The RCL is formed by all *feasible* elements $e \in E$ which can be inserted into the partial solution under construction without destroying feasibility and whose quality is superior to the threshold value, that is $c(e) \in [c^{\min}, c^{\min} + \alpha(c^{\max} - c^{\min})]$. The case $\alpha = 0$ corresponds to a pure greedy algorithm, while $\alpha = 1$ is equivalent to a random construction. The pseudo-code in Fig. 11.6 is a refinement of the greedy randomized construction algorithm, whose pseudo-code appears in Fig. 11.2.

GRASP may be viewed as a repetitive sampling technique. Each iteration produces a sample solution from an unknown distribution, whose mean value and variance are functions of the restrictive nature of the RCL. The pseudo-code in Fig. 11.6 shows that the parameter α controls the amounts of greediness and randomness in the algorithm. [Resende and Ribeiro \(2003b, 2010\)](#) have shown that what often leads to good solutions are relatively low average solution values (i.e. close to the value of

```

procedure GreedyRandomizedConstruction( $\alpha$ , Seed)
1.  $S \leftarrow \emptyset$ ;
2. Initialize the candidate set:  $C \leftarrow E$ ;
3. Evaluate the incremental cost  $c(e)$  for all  $e \in C$ ;
4. while  $C \neq \emptyset$  do
5.    $c^{min} \leftarrow \min\{c(e) \mid e \in C\}$ ;
6.    $c^{max} \leftarrow \max\{c(e) \mid e \in C\}$ ;
7.   Build the restricted candidate list:  $RCL \leftarrow \{e \in C \mid c(e) \leq c^{min} + \alpha(c^{max} - c^{min})\}$ ;
8.   Choose  $s$  at random from  $RCL$ ;
9.   Incorporate  $s$  into solution:  $S \leftarrow S \cup \{s\}$ ;
10.  Update the candidate set  $C$ ;
11.  Reevaluate the incremental cost  $c(e)$  for all  $e \in C$ ;
12. end;
13. return  $S$ ;
end.

```

Fig. 11.6 Refined pseudo-code of the construction phase using parameter α for defining a quality threshold

the purely greedy solution obtained with $\alpha = 0$) in the presence of a relatively large variance (i.e. solutions obtained with a larger degree of randomness as α increases), such as is often the case for $\alpha = 0.2$.

[Prais and Ribeiro \(2000a\)](#) showed that using a single fixed value for the value of the RCL parameter α often hinders finding a high-quality solution, which eventually could be found if another value was used. An alternative is to use a different value of α , chosen uniformly at random in the interval $[0, 1]$, at each GRASP iteration. [Prais and Ribeiro \(2000a\)](#) proposed another alternative, the *Reactive* GRASP extension of the basic procedure, in which the parameter α is self-tuned and its value is periodically modified according with the quality of the solutions previously obtained. Applications to other problems (see e.g. [Festa and Resende 2009a; Resende and Ribeiro 2010](#)) have shown that Reactive GRASP outperforms the basic algorithm. These results motivated the study of the behavior of GRASP for different strategies for the variation of the value of the RCL parameter α . The experiments reported in [Prais and Ribeiro \(2000a\)](#) show that implementation strategies based on the variation of α are likely to be more affective than one using a single fixed value for this parameter.

Two other randomized greedy approaches, with smaller worst-case complexities than that depicted in the pseudo-code of Fig. 11.6, were proposed by [Resende and Werneck \(2004\)](#). Instead of combining greediness and randomness at each step of the construction procedure, the *random plus greedy* scheme applies randomness during the first p construction steps to produce a random partial solution. Next, the algorithm completes the solution with one or more pure greedy construction steps. By changing the value of the parameter p , one can control the balance between greediness and randomness in the construction: larger values of p correspond to solutions that are more random, with smaller values corresponding to greedier solutions. The *sampled greedy* construction provides a different way to combine randomness and greediness. This procedure is also controlled by a parameter p . At each step

of the construction process, the procedure builds a RCL by sampling $\min\{p, |C|\}$ elements of the candidate set C . Each of the sampled elements is evaluated by the greedy function and an element with the smallest greedy function value is added to the partial solution. These steps are repeated until there are no more candidate elements. As before, the balance between greediness and randomness can be controlled by changing the value of the parameter p , i.e. the number of candidate elements that are sampled. Small sample sizes lead to more random solutions, while large sample sizes lead to more greedy solutions.

11.4 GRASP with Path-Relinking

GRASP, as originally proposed, is a memoryless procedure in which each iteration does not make use of information gathered in previous iterations. Path-relinking is a major enhancement used for search intensification with GRASP. By adding memory structures to the basic procedure described above, path-relinking leads to significant improvements in solution time and quality.

The basic principles of path-relinking were described in Sect. 11.2.7. The use of path-relinking within a GRASP procedure was proposed in [Laguna and Martí \(1999\)](#) and followed by extensions, improvements, and successful applications (see Sect. 11.7.2). Surveys of GRASP with path-relinking can be found in [Resende and Ribeiro \(2003a, 2005a, 2010\)](#). Different schemes have been proposed for the implementation of path-relinking. In essence, it has been applied as a post-optimization phase (between every pair of elite solutions in the pool of elite solutions) and as an intensification strategy (between every local optimum obtained after the local search phase and one or more elite solutions in the pool of elite solutions).

In this last context, path-relinking is applied to pairs of solutions, one of which is a locally optimal solution and the other is randomly chosen from a pool with a limited number `MaxElite` of elite solutions found along the search. A simple strategy is to assign equal probabilities of being selected to each elite solution. Another strategy assigns probabilities proportional to the cardinality of the symmetric difference between the elite solution and the locally optimal solution. This strategy favors elite solutions that result in longer paths. One of these solutions is called the *initial solution*, while the other is the *guiding solution*. One or more paths in the solution space graph connecting these solutions may be explored in the search for better solutions. The pool of elite solutions is originally empty. Since we wish to maintain a pool of good but diverse solutions, each locally optimal solution obtained by local search is considered as a candidate to be inserted into the pool if it is sufficiently different from every other solution currently in the pool. If the pool already has `MaxElite` solutions and the candidate is better than the worst of them, then a simple strategy is to have the candidate replace the worst elite solution. This strategy improves the quality of the elite set. Another strategy is to have the candidate replace an elite solution with worse objective function value that is most similar to it. This strategy improves the diversity of the elite set as well as its quality.

```

procedure GRASPwithPathRelinking(MaxIterations, Seed)
1. Set  $f^* \leftarrow \infty$ ;
2. Set Pool  $\leftarrow \emptyset$ ;
3. for  $k = 1, \dots, \text{MaxIterations}$  do
4.    $S \leftarrow \text{GreedyRandomizedAlgorithm}(\text{Seed})$ ;
5.   if  $S$  is infeasible then
6.      $S \leftarrow \text{RepairSolution}(S)$ ;
7.   endif;
8.    $S \leftarrow \text{LocalSearch}(S)$ ;
9.   if  $k > 1$  then
10.     Randomly select a solution  $S' \in \text{Pool}$ ;
11.      $S \leftarrow \text{PathRelinking}(S', S)$ ;
12.   endif;
13.   if  $f(S) < f^*$  then
14.      $S^* \leftarrow S$ ;
15.      $f^* \leftarrow f(S)$ ;
16.   end_if;
17.   Update Pool with  $S$  if it satisfies the membership conditions;
18. end_for;
19. return  $S^*$ ;
end.

```

Fig. 11.7 Template of a GRASP with path-relinking heuristic for minimization

The pseudo-code in Fig. 11.7 illustrates the main steps of a GRASP procedure using path-relinking to implement a memory-based intensification strategy.

Several alternatives for applying path-relinking to a pair of solutions S and S' have been considered and combined in the literature. These include forward, backward, back and forward, mixed, truncated, greedy randomized adaptive, and evolutionary path-relinking. All these alternatives involve trade-offs between computation time and solution quality.

In *forward* path-relinking, the GRASP local optimum S is designated as the initial solution and the pool solution S' is made the guiding solution. The roles of S and S' are interchanged in *backward* path-relinking. This scheme was originally proposed in Aix *et al.* (2005), Ribeiro *et al.* (2002), and Resende and Ribeiro (2003a). The main advantage of this approach over forward path-relinking comes from the fact that, in general, there are more high-quality solutions near pool elements than near GRASP local optima. Backward path-relinking explores more thoroughly the neighborhood around the pool solution, whereas forward path-relinking explores more thoroughly the neighborhood around the GRASP local optimum. Experiments in Aix *et al.* (2005) and Resende and Ribeiro (2003a) have confirmed that backward path-relinking usually outperforms forward path-relinking. *Back and forward* path-relinking combines forward and backward path-relinking, exploring two different paths. It finds solutions at least as good as forward path-relinking or backward path-relinking, but at the expense of taking about twice as long to run. *Mixed* path-relinking shares the benefits of back and forward path-relinking, in about the same time as forward or backward path-relinking alone.

This is achieved by interchanging the roles of the initial and guiding solutions at each step of the path-relinking procedure. [Ribeiro and Rosseti \(2007\)](#) have shown experimentally that it outperforms forward, backward, and back and forward path-relinking (see also [Resende and Ribeiro 2010](#)).

Other strategies have been proposed more recently. *Truncated* path-relinking can be applied to either forward, backward, back and forward, or mixed path-relinking. Instead of exploring the entire path, it takes only a fraction of those steps and consequently takes a fraction of the time to run. Since high-quality solutions tend to be near the initial or guiding solutions, exploring part of the path near the extremities may produce solutions about as good as those found by exploring the entire path. Indeed, [Resende et al. \(2010\)](#) showed experimentally that this is the case for instances of the max–min diversity problem. *Greedy randomized adaptive* path-relinking, introduced by [Faria et al. \(2005\)](#), is a semi-greedy version of path-relinking. Instead of taking the best move in the symmetric difference still not performed, a RCL of good moves still not performed is set up and a randomly selected move from the RCL is applied. By applying this strategy several times between the initial and guiding solutions, several alternative paths can be explored. [Resende and Werneck \(2004, 2006\)](#) described an *evolutionary* path-relinking scheme applied to pairs of elite solutions and used as a post-optimization phase, in which the pool resulting from the GRASP with path-relinking iterations progressively evolves as a population. Similar schemes were also used by [Aix et al. \(2005\)](#) and [Resende et al. \(2010\)](#).

11.5 Extensions

Hybridizations of GRASP with metaheuristics such as tabu search, simulated annealing, variable neighborhood search, iterated local search and genetic algorithms have been reported in the literature.

Almost all the randomization effort in GRASP involves the construction phase, since the local search always stops at the first local optimum. Variable neighborhood search (VNS), see Chap. 12, relies almost entirely on the randomization of the local search to escape from local optima. Thus GRASP and VNS may be considered as complementary and potentially capable of leading to effective hybrid methods. [Festa et al. \(2002\)](#) studied different variants and combinations of GRASP and VNS for the MAX-CUT problem, finding and improving the best known solutions for some open instances in the literature. Other examples of hybrids of GRASP with VNS include [Beltrán et al. \(2004\)](#) and [Canuto et al. \(2001\)](#).

GRASP has also been used in conjunction with genetic algorithms. Basically, the greedy randomized strategy used in the construction phase of a GRASP is applied to generate the initial population for a genetic algorithm. We may cite, for example, the genetic algorithm of [Ahuja et al. \(2000\)](#) for the quadratic assignment problem, which makes use of the GRASP proposed by [Li et al. \(1994\)](#) to create the initial

population of solutions. A similar approach was used by [Armony et al. \(2000\)](#), with the initial population made up of both randomly generated solutions and those built by a GRASP.

The hybridization of GRASP with tabu search was first studied by [Laguna and González-Velarde \(1991\)](#). [Delmaire et al. \(1999\)](#) considered two approaches. In the first, GRASP is applied as a powerful diversification strategy in the context of a tabu search procedure. The second approach is an implementation of the Reactive GRASP algorithm, in which the local search phase is strengthened by tabu search. Two two-stage heuristics are proposed in [Abdinnour-Helm and Hadley \(2000\)](#) for solving the multi-floor facility layout problem. GRASP/TS applies a GRASP to find the initial layout and tabu search to refine it. [Souza et al. \(2004\)](#) used a short-term tabu search procedure as a substitute for the standard local search in a GRASP heuristic for the capacitated minimum spanning tree problem.

Iterated local search (ILS) iteratively builds a sequence of solutions generated by the repeated application of local search and perturbation of the local optimum found by local search ([Lourenço et al. 2003; Martin et al. 1991](#)). [Ribeiro and Urrutia \(2007\)](#) presented a GRASP with ILS heuristic for the mirrored traveling tournament problem. In this case, the GRASP construction produces a solution which is passed on to the ILS procedure.

The hybridization of GRASP with data mining techniques was introduced by [Ribeiro et al. \(2006\)](#). This scheme uses a data mining algorithm to search for solution patterns that occur in high-quality elite solutions produced by the basic GRASP algorithm. These mined patterns are used as initial building blocks that guide the construction of new solutions that are submitted to local search. A survey of applications of DM-GRASP can be found in [Santos et al. \(2008\)](#).

11.6 Tricks of the Trade

1. An especially appealing characteristic of GRASP is the ease with which it can be implemented. Few parameters need to be set and tuned. Therefore, algorithm development and coding can focus on implementing efficient data structures to ensure quick GRASP iterations.
2. Most metaheuristics benefit from good initial solutions. Clever low-complexity algorithms leading to good feasible solutions can often be devised by examination of the problem structure. Good initial solutions lead to better final solutions and significantly reduce the time taken by local search.
3. Using a single, fixed value for the RCL parameter α very often hinders finding a high-quality solution, which eventually could be found if another value was used. The use of strategies such as Reactive GRASP which vary the value of α may lead to better and more diverse solutions. The reactive approach leads to improvements over the basic GRASP in terms of robustness and solution quality, due to greater diversification and less reliance on parameter tuning. In addition to the original applications reported by [Prais and Ribeiro \(2000a,b\)](#), it has also

- been applied by [Álvarez-Valdés et al. \(2008b\)](#), [Binato et al. \(2002\)](#), [Binato and Oliveira \(2002\)](#), [Boudia et al. \(2007\)](#), [Delmaire et al. \(1999\)](#) and [Scaparra and Church \(2005\)](#). Another simple strategy is to uniformly select at random a value for α at each GRASP iteration from the interval $[0, 1]$.
4. Local search procedures may be implemented using a best-improving or a first-improving strategy, as well as any combination of them. In the case of the best-improving strategy, all neighbors are investigated and the current solution is replaced by the best neighbor. In the case of a first-improving strategy, the current solution moves to the first neighbor whose cost function value is smaller than that of the current solution. Both strategies quite often lead to same-quality solutions, but in smaller computation times when the first-improving strategy is used. Premature convergence to a non-global local minimum is more likely to occur with a best-improving strategy.
 5. The definition of a neighborhood is not unique. Some implementations of metaheuristics make use of multiple neighborhood structures to improve solution quality and to speed up the search. Variable neighborhood descent (VND) allows the systematic exploration of multiple neighborhoods ([Hansen and Mladenović 2003](#)). It is based on the facts that a local minimum with respect to one neighborhood is not necessarily a local minimum with respect to another and that a global minimum is a local minimum with respect to all neighborhoods. Furthermore, VND is also based on the empirical observation that, for many problems, local minima with respect to one or more neighborhoods are relatively close to each other ([Hansen and Mladenović 2003](#)). Since a global minimum is a local minimum with respect to all neighborhoods, it should be easier to find a global minimum if more neighborhoods are explored. In the case of nested neighborhoods, the search is first confined to smaller neighborhoods. A larger neighborhood is explored only after a local minimum is found in the current, smaller neighborhood. Neighborhoods are not necessarily nested. Non-nested neighborhoods have been successfully used by, for example, [Aloise et al. \(2006\)](#).
 6. Local search can be considerably accelerated with the use of appropriate data structures and efficient algorithms. All possible attempts should be made to improve the neighborhood search procedure. Algorithms should be coded to have minimum complexity. The use of circular lists to represent and search the neighborhood is very helpful. Candidate lists storing the move values may be easy to update or may be used as quick approximations to avoid their re-evaluation at every iteration. We have seen several implementations in which the time taken by the first local search code dropped from several minutes to a few milliseconds in the final version.
 7. Path-relinking is a very effective strategy to improve solution quality and to reduce computation times, leading to more robust implementations. Any available knowledge about the problem structure should be used in the development of efficient algorithms to explore the most attractive strategy for path-relinking.
 8. Different metaheuristics make use of a number of common components, such as greedy constructions, local search, randomization, candidate lists, multiple

neighborhoods and path-relinking. Borrowing and incorporating principles from other metaheuristics leads to efficient hybridizations of GRASP, which often results in the best algorithm for some problem class.

9. There is no universal, general purpose metaheuristic that gives the best results for every problem (Wolpert and Macready 1997)—see Chap. 16. The structure of each problem should be explored to bring additional intelligence into the solution strategy. Knowledge, experience and information available in the literature for similar problems are very helpful. However, one should not be obsessed with a fixed idea or bounded by strategies that worked for other problems but might not be appropriate for the one on hand. The best algorithm is always the one that most exploits the structure of your problem and gives the best results.

11.7 Some Promising Areas for Future Application

We conclude this chapter with two promising areas for future applications of GRASP.

11.7.1 *Continuous GRASP*

Hirsch et al. (2007b) (see also Hirsch 2006) proposed an adaptation of GRASP for derivative-free continuous global optimization. Continuous GRASP (or simply C-GRASP) was shown to perform well on a set of multimodal test functions, as well as on difficult real-world applications (Hirsch et al. 2007b). It was applied to the registration of sensors in a sensor network (Hirsch et al. 2006), to compute solutions for systems of nonlinear equations (Hirsch et al. 2009), to determine which drugs are responsible for adverse reactions in patients (Hirsch et al. 2007a), and for dynamic, decentralized path planning of unmanned aerial vehicles (Hirsch and Ortiz-Pena 2009; Hirsch et al. 2007c). Improvements to the original C-GRASP (Hirsch et al. 2007b) are presented in Hirsch et al. (2010). These improvements are aimed at making implementations of the algorithm more efficient and increasing robustness, while at the same time keeping the overall algorithm simple to implement.

The local improvement procedures in the derivative-free C-GRASP sample points around the solution produced by the global greedy randomized procedure. Since they only make function evaluations and do not use gradient information, they can be used for local optimization of any type of function, including ones that are not smooth. Birgin et al. (2010) adapt C-GRASP for global optimization of functions for which gradients can be computed. This is accomplished by using GENCAN (Birgin and Martínez 2002), an active-set method for bound-constrained local minimization.

11.7.2 Probabilistic-Based Stopping Rules

The absence of effective stopping criteria is one of the main drawbacks of most metaheuristics. Implementations of such algorithms usually stop after performing a given maximum number of iterations or a given maximum number of consecutive iterations without improvement in the best known solution value, or after the stabilization of a set of elite solutions found along the search. Ribeiro et al. (2011) proposed effective probabilistic stopping rules for randomized metaheuristics such as GRASP, VNS, simulated annealing and genetic algorithms, based on the estimation of the probability of finding better solutions than the incumbent. Such probabilities may be computed and used online to estimate the trade-off between solution improvement and the time needed to achieve it. The results described in Ribeiro et al. (2011) are being extended to encompass memory-based methods such as GRASP with path-relinking and tabu search.

Sources of Additional Information

Surveys on GRASP (Feo and Resende 1995; Resende and Ribeiro 2003b, 2010), path-relinking (Resende and Ribeiro 2005a; Ribeiro and Resende 2012) and its applications (Festa and Resende 2002, 2009a,b) can be found in the literature, to which the interested reader is referred for more details.

The web page www.research.att.com/~mgcr contains an always-updated version of the annotated bibliography on GRASP which appeared in Festa and Resende (2002, 2009a,b). Source codes for GRASP heuristics for several problems are also available at <http://www.research.att.com/~mgcr/src/index.html>. The Twitter web page <http://twitter.com/graspheuristic> posts links to recently published papers on GRASP and its applications.

Time-to-target (TTT) plots display on the ordinate axis the probability that an algorithm will find a solution at least as good as a given target value within a given running time, shown on the abscissa axis. TTT plots were used by Feo et al. (1994) and have been advocated also by Hoos and Stützle (1998) as a way to characterize the running times of stochastic algorithms for combinatorial optimization. Aiex et al. (2002) advocate and largely explored the use of TTT plots to evaluate and compare different randomized algorithms running on the same problem. The use of TTT plots has been growing ever since and they have been extensively applied in computational studies of sequential and parallel implementations of randomized algorithms (see e.g. Resende and Ribeiro 2003b, 2010; Ribeiro and Rossetti 2007). The foundations of the construction of TTT plots, together with their interpretation and applications, were surveyed by Aiex et al. (2007). This reference also describes a Perl language program to create TTT plots for measured CPU times that can be downloaded from <http://www.research.att.com/~mgcr/ttpplots>.

The first application of GRASP described in the literature concerned the set-covering problem (Feo and Resende 1989). GRASP has been applied to many problems in different areas, such as routing (Argüello et al. 1997; Corberán et al. 2002; Kontoravdis and Bard 1995; Reghioui et al. 2007), logic (Deshpande and Triantaphyllou 1998; Festa et al. 2006; Pardalos et al. 1996; Resende and Feo 1996; Resende et al. 1997, 2000), covering and partitioning (Álvarez-Valdés et al. 2005; Areibi and Vannelli 1997; Feo and Resende 1989; Hammer and Rader Jr 2001), location (Abdinnour-Helm and Hadley 2000; Colomé and Serra 2001; Cravo et al. 2008; Han and Raja 2003; Holmqvist et al. 1997; Delmaire et al. 1999; Urban 1998; Klincewicz 1992), minimum Steiner tree (Canuto et al. 2001; Martins et al. 1999, 2000, 1998; Ribeiro et al. 2002), optimization in graphs (Abello et al. 1999, 2002; Ahuja et al. 2001; Arroyo et al. 2008; Feo et al. 1994; Festa et al. 2001, 2002; Holmqvist et al. 1998; Laguna et al. 1994; Laguna and Martí 2001; Martí 2001; Martins et al. 2000; Pardalos et al. 1999; Resende 1998; Resende et al. 1998; Resende and Ribeiro 1997; Ribeiro and Resende 1999; Ribeiro et al. 2002; Souza et al. 2004), assignment (Ahuja et al. 2000; Aiex et al. 2005; Feo and González-Velarde 1995; Fleurant and Glover 1999; Li et al. 1994; Mavridou et al. 1998; Murphrey et al. 1998a,b; Oliveira et al. 2004; Pardalos et al. 1995, 1997; Pitsoulis et al. 2001; Prais and Ribeiro 2000b; Resende et al. 1996; Robertson 2001), timetabling, scheduling, and manufacturing (Aiex et al. 2003; Álvarez-Valdés et al. 2008b,a; Andrade and Resende 2006; Bard and Feo 1989, 1991; Bard et al. 1996; Binato et al. 2002; Boudia et al. 2007; Commander et al. 2004; Feo and Bard 1989; Feo et al. 1995, 1996, 1991; Klincewicz and Rajan 1994; Laguna and González-Velarde 1991; Ribeiro and Urrutia 2007; Ríos-Mercado and Bard 1998, 1999; Xu and Chiu 2001; Yen et al. 2000), transportation (Argüello et al. 1997; Feo and Bard 1989; Feo and González-Velarde 1995; Scaparra and Church 2005), power systems (Binato and Oliveira 2002; Binato et al. 2001; Faria et al. 2005), telecommunications (Abello et al. 1999; Amaldi et al. 2003; Andrade and Resende 2006; Klincewicz 1992; Piñana et al. 2004; Prais and Ribeiro 2000b; Resende and Resende 1999; Resende 1998; Resende and Ribeiro 2003a; Ribeiro and Rosseti 2002; Srinivasan et al. 2000), graph and map drawing (Cravo et al. 2008; Fernández and Martí 1999; Laguna and Martí 1999; Martí 2001; Osman et al. 2003; Resende and Ribeiro 1997; Ribeiro and Resende 1999), biology (Andreatta and Ribeiro 2002; Ribeiro and Vianna 2005) and VLSI (Areibi and Vannelli 1997), among others.

GRASP is a metaheuristic very well suited for parallel implementation, due to the independence of its iterations. Parallel cooperative versions of GRASP with path-relinking may also be implemented in parallel if a centralized pool of elite solutions is kept by one of the processors. Surveys and accounts of parallel implementations of GRASP in networks of workstations, clusters and grids may be found in Cung et al. (2002), Martins et al. (2004, 2006), Resende and Ribeiro (2005b), Ribeiro et al. (2007) and Ribeiro and Rosseti (2002, 2007).

References

- Abdinnour-Helm S, Hadley SW (2000) Tabu search based heuristics for multi-floor facility layout. *Int J Prod Res* 38:365–383
- Abello J, Pardalos PM, Resende MGC (1999) On maximum clique problems in very large graphs. In: Abello J, Vitter J (eds) External memory algorithms and visualization, DIMACS 50. AMS, Providence, pp 199–130
- Abello J, Resende MGC, Sudarsky S (2002) Massive quasi-clique detection. In: Rajsbaum S (ed) LATIN 2002: theoretical informatics. LNCS 2286. Springer, Berlin, pp 598–612
- Ahuja RK, Orlin JB, Tiwari A (2000) A greedy genetic algorithm for the quadratic assignment problem. *Comput Oper Res* 27:917–934
- Ahuja RK, Orlin JB, Sharma D (2001) Multi-exchange neighborhood structures for the capacitated minimum spanning tree problem. *Math Program* 91:71–97
- Aiex RM, Resende MGC, Ribeiro CC (2002) Probability distribution of solution time in GRASP: an experimental investigation. *J Heuristics* 8:343–373
- Aiex RM, Binato S, Resende MGC (2003) Parallel GRASP with path-relinking for job shop scheduling. *Parallel Comput* 29:393–430
- Aiex RM, Pardalos PM, Resende MGC, Toraldo G (2005) GRASP with path-relinking for three-index assignment. *INFORMS J Comput* 17:224–247
- Aiex RM, Resende MGC, Ribeiro CC (2007) TTPLOTS: a Perl program to create time-to-target plots. *Optim Lett* 1:355–366
- Aloise DJ, Aloise D, Rocha CTM, Ribeiro CC, Ribeiro Filho JC, Moura LSS (2006) Scheduling workover rigs for onshore oil production. *Discret Appl Math* 154:695–702
- Álvarez-Valdés R, Parreño F, Tamarit JM (2005) A GRASP algorithm for constrained two-dimensional non-guillotine cutting problems. *J Oper Res Soc* 56:414–425
- Álvarez-Valdés R, Crespo E, Tamarit JM, Villa F (2008a) GRASP and path relinking for project scheduling under partially renewable resources. *Eur J Oper Res* 189:1153–1170
- Álvarez-Valdés R, Parreño F, Tamarit JM (2008b) Reactive GRASP for the strip-packing problem. *Comput Oper Res* 35:1065–1083
- Amaldi E, Capone A, Malucelli F (2003) Planning UMTS base station location: optimization models with power control and algorithms. *IEEE Trans Wirel Commun* 2:939–952
- Andrade DV, Resende MGC (2006) A GRASP for PBX telephone migration scheduling. In: Proc 8th INFORMS telecoms, Dallas, TX
- Andreatta AA, Ribeiro CC (2002) Heuristics for the phylogeny problem. *J Heuristics* 8:429–447
- Areibi S, Vannelli A (1997) A GRASP clustering technique for circuit partitioning. In: Gu J, Pardalos PM (eds) Satisfiability problems, DIMACS 35. AMS, Providence, pp 711–724
- Argüello MF, Bard JF, Yu G (1997) A GRASP for aircraft routing in response to groundings and delays. *J Comb Optim* 1:211–228

- Armony M, Klincewicz JC, Luss H, Rosenwein MB (2000) Design of stacked self-healing rings using a genetic algorithm. *J Heuristics* 6:85–105
- Arroyo JEC, Vieira PS, Vianna DS (2008) A GRASP algorithm for the multi-criteria minimum spanning tree problem. *Ann Oper Res* 159:125–133
- Bard JF, Feo TA (1989) Operations sequencing in discrete parts manufacturing. *Manage Sci* 35:249–255
- Bard JF, Feo TA (1991) An algorithm for the manufacturing equipment selection problem. *IIE Trans* 23:83–92
- Bard JF, Feo TA, Holland S (1996) A GRASP for scheduling printed wiring board assembly. *IIE Trans* 28:155–165
- Beltrán JD, Calderón JE, Cabrera RJ, Pérez JAM, Moreno-Vega JM (2004) GRASP/VNS hybrid for the strip packing problem. In: Proc hybrid metaheuristics, Valencia, Spain, pp 79–90
- Binato S, Oliveira GC (2002) A reactive GRASP for transmission network expansion planning. In: Ribeiro CC, Hansen P (eds) Essays and surveys in metaheuristics. Kluwer, Dordrecht, pp 81–100
- Binato S, Oliveira GC, Araújo JL (2001) A greedy randomized adaptive search procedure for transmission expansion planning. *IEEE Trans Power Syst* 16:247–253
- Binato S, Hery WJ, Loewenstern D, Resende MGC (2002) A GRASP for job shop scheduling. In: Ribeiro CC, Hansen P (eds) Essays and surveys in metaheuristics. Kluwer, Dordrecht, pp 59–79
- Birgin EG, Martínez JM (2002) Large-scale active-set box-constrained optimization method with spectral projected gradients. *Comput Optim Appl* 23:101–125
- Birgin EG, Gozzi EM, Resende MGC, Silva RMA (2010) Continuous GRASP with a local active-set method for bound-constrained global optimization. *J Glob Optim* 48:289–310
- Boudia M, Louly MAO, Prins C (2007) A reactive GRASP and path relinking for a combined production-distribution problem. *Comput Oper Res* 34:3402–3419
- Canuto SA, Resende MGC, Ribeiro CC (2001) Local search with perturbations for the prize-collecting Steiner tree problem in graphs. *Networks* 38:50–58
- Colomé R, Serra D (2001) Consumer choice in competitive location models: formulations and heuristics. *Pap Reg Sci* 80:439–464
- Commander CW, Butenko SI, Pardalos PM, Oliveira CAS (2004) Reactive GRASP with path relinking for the broadcast scheduling problem. In: Proceedings of the 40th Annual International telemetry conference, San Diego, CA, pp 792–800
- Corberán A, Martí R, Sanchís JM (2002) A GRASP heuristic for the mixed Chinese postman problem. *Eur J Oper Res* 142:70–80
- Cravo GL, Ribeiro GM, Nogueira Lorena LA (2008) A greedy randomized adaptive search procedure for the point-feature cartographic label placement. *Comput Geosci* 34:373–386
- Cung V-D, Martins SL, Ribeiro CC, Roucairol C (2002) Strategies for the parallel implementation of metaheuristics. In: Ribeiro CC, Ribeiro CC (eds) Essays and surveys in metaheuristics. Kluwer, Dordrecht, pp 263–308

- Delmaire H, Díaz JA, Fernández E, Ortega M (1999) Reactive GRASP and tabu search based heuristics for the single source capacitated plant location problem. *INFOR* 37:194–225
- Deshpande AS, Triantaphyllou E (1998) A greedy randomized adaptive search procedure (GRASP) for inferring logical clauses from examples in polynomial time and some extensions. *Math Comput Model* 27:75–99
- Duarte AR, Ribeiro CC, Urrutia S (2007a) A hybrid ILS heuristic to the referee assignment problem with an embedded MIP strategy. In: *Hybrid metaheuristics. LNCS* 4771. Springer, Berlin, pp 82–95
- Duarte AR, Ribeiro CC, Urrutia S, Haesler EH (2007b) Referee assignment in sports leagues. In: *PATAT VI. LNCS* 3867. Springer, Berlin, pp 158–173
- Faria H Jr, Binato S, Resende MGC, Falcão DJ (2005) Transmission network design by a greedy randomized adaptive path relinking approach. *IEEE Trans Power Syst* 20:43–49
- Feo TA, Bard JF (1989) Flight scheduling and maintenance base planning. *Manage Sci* 35:1415–1432
- Feo TA, González-Velarde JL (1995) The intermodal trailer assignment problem: models, algorithms, and heuristics. *Transp Sci* 29:330–341
- Feo TA, Resende MGC (1989) A probabilistic heuristic for a computationally difficult set covering problem. *Oper Res Lett* 8:67–71
- Feo TA, Resende MGC (1995) Greedy randomized adaptive search procedures. *J Glob Optim* 6:109–133
- Feo TA, Venkatraman K, Bard JF (1991) A GRASP for a difficult single machine scheduling problem. *Comput Oper Res* 18:635–643
- Feo TA, Resende MGC, Smith SH (1994) A greedy randomized adaptive search procedure for maximum independent set. *Oper Res* 42:860–878
- Feo TA, Bard JF, Holland S (1995) Facility-wide planning and scheduling of printed wiring board assembly. *Oper Res* 43:219–230
- Feo TA, Sarathy K, McGahan J (1996) A GRASP for single machine scheduling with sequence dependent setup costs and linear delay penalties. *Comput Oper Res* 23:881–895
- Fernández E, Martí R (1999) GRASP for seam drawing in mosaicking of aerial photographic maps. *J Heuristics* 5:181–197
- Festa P, Resende MGC (2002) GRASP: an annotated bibliography. In: Ribeiro CC, Hansen P (eds) *Essays and surveys in metaheuristics*. Kluwer, Dordrecht, pp 325–367
- Festa P, Resende MGC (2009a) An annotated bibliography of GRASP, part I: algorithms. *Int Trans Oper Res* 16:1–24
- Festa P, Resende MGC (2009b) An annotated bibliography of GRASP, part II: applications. *Int Trans Oper Res* 16:131–172
- Festa P, Pardalos PM, Resende MGC (2001) Algorithm 815: FORTRAN subroutines for computing approximate solution to feedback set problems using GRASP. *ACM Trans Math Softw* 27:456–464
- Festa P, Pardalos PM, Resende MGC, Ribeiro CC (2002) Randomized heuristics for the MAX-CUT problem. *Optim Methods Softw* 7:1033–1058

- Festa P, Pardalos PM, Pitsoulis LS, Resende MGC (2006) GRASP with path-relinking for the weighted MAXSAT problem. *ACM J Exp Algorithm* 11:1–16
- Fleurent C, Glover F (1999) Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory. *INFORMS J Comput* 11:198–204
- Glover F (1996) Tabu search and adaptive memory programming—advances, applications and challenges. In: Barr RS, Helgasonm RV, Kennington JL (eds) *Interfaces in computer science and operations research*. Kluwer, Dordrecht, pp 1–75
- Glover F, Laguna M (1997) Tabu search. Kluwer, Dordrecht
- Glover F, Laguna M, Martí R (2000) Fundamentals of scatter search and path re-linking. *Control Cybern* 39:653–684
- Hammer PL, Rader DJ Jr (2001) Maximally disjoint solutions of the set covering problem. *J Heuristics* 7:131–144
- Han BT, Raja VT (2003) A GRASP heuristic for solving an extended capacitated concentrator location problem. *Int J Inf Technol Decis Making* 2:597–617
- Hansen P, Mladenović N (2003) Variable neighborhood search. In: Glover F, Kochenberger G (eds) *Handbook of metaheuristics*. Kluwer, Dordrecht, pp 145–184
- Hirsch MJ (2006) GRASP-based heuristics for continuous global optimization problems. PhD thesis, University of Florida
- Hirsch MJ, Ortiz-Pena H (2009) UAV cooperative control for multiple target tracking. In: Du D-Z, Pardalos PM (eds) *DIMACS/DyDAn workshop on approximation algorithms in wireless ad hoc and sensor networks*, Piscataway, NJ
- Hirsch MJ, Pardalos PM, Resende MGC (2006) Sensor registration in a sensor network by continuous GRASP. In: *Proceeding of the MILCOM 2006*, Washington, DC
- Hirsch MJ, Meneses CN, Pardalos PM, Ragle MA, Resende MGC (2007a) A Continuous GRASP to determine the relationship between drugs and adverse reactions. In: Seref O, Kundakcioglu OE, Pardalos PM (eds) *Data Mining, systems analysis, and optimization in biomedicine*. American Institute of Physics, New York, pp 106–121
- Hirsch MJ, Meneses CN, Pardalos PM, Resende MGC (2007b) Global optimization by continuous GRASP. *Optim Lett* 1:201–212
- Hirsch MJ, Ortiz-Pena H, Sapankevych N, Neese R (2007c) Efficient flight formation for tracking of a ground target. In: *Proceeding of the National Fire Control Symposium*, San Diego, CA
- Hirsch MJ, Pardalos PM, Resende MGC (2009) Solving systems of nonlinear equations using continuous GRASP. *Nonlinear Anal Real World Appl* 10:2000–2006
- Hirsch MJ, Pardalos PM, Resende MGC (2010) Speeding up continuous GRASP. *Eur J Oper Res* 205:507–521
- Holmqvist K, Migdalas A, Pardalos PM (1997) Greedy randomized adaptive search for a location problem with economies of scale. In: Bomze IM et al (eds) *Developments in global optimization*. Kluwer, Dordrecht, pp 301–313
- Holmqvist K, Migdalas A, Pardalos PM (1998) A GRASP algorithm for the single source uncapacitated minimum concave-cost network flow problem.

- In: Pardalos PM, Du D-Z (eds) Network design: connectivity and facilities location, DIMACS 40. AMS, Providence, pp 131–142
- Hoos HH, Stützle T (1998) Evaluating Las Vegas algorithms—pitfalls and remedies. In: Proceedings of the 14th Conference on uncertainty in artificial intelligence, Madison, WI, pp 238–245
- Klincewicz JG (1992) Avoiding local optima in the p -hub location problem using tabu search and GRASP. *Ann Oper Res* 40:283–302
- Klincewicz JG, Rajan A (1994) Using GRASP to solve the component grouping problem. *Nav Res Logist* 41:893–912
- Kontoravdis G, Bard JF (1995) A GRASP for the vehicle routing problem with time windows. *ORSA J Comput* 7:10–23
- Laguna M, González-Velarde JL (1991) A search heuristic for just-in-time scheduling in parallel machines. *J Intell Manuf* 2:253–260
- Laguna M, Martí R (1999) GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS J Comput* 11:44–52
- Laguna M, Martí R (2001) A GRASP for coloring sparse graphs. *Comput Optim Appl* 19:165–178
- Laguna M, Feo TA, Elrod HC (1994) A greedy randomized adaptive search procedure for the two-partition problem. *Oper Res* 42:677–687
- Li Y, Pardalos PM, Resende MGC (1994) A greedy randomized adaptive search procedure for the quadratic assignment problem. In: Pardalos PM, Wolkowicz H (eds) Quadratic assignment and related problems, DIMACS 16. AMS, Providence, pp 237–261
- Lourenço HR, Martin OC, Stützle T (2003) Iterated local search. In: Glover F, Kochenberger G (eds) *Handbook of metaheuristics*. Kluwer, Dordrecht, pp 321–353
- Martí R (2001) Arc crossing minimization in graphs with GRASP. *IIE Trans* 33:913–919
- Martin O, Otto SW, Felten EW (1991) Large-step Markov chains for the traveling salesman problem. *Complex Syst* 5:299–326
- Martins SL, Ribeiro CC, Souza MC (1998) A parallel GRASP for the Steiner problem in graphs. In: Ferreira A, Rolim J (eds) *Proceedings of IRREGULAR'98. LNCS* 1457. Springer, Berlin, pp 285–297
- Martins SL, Pardalos PM, Resende MGC, Ribeiro CC (1999) Greedy randomized adaptive search procedures for the Steiner problem in graphs. In: Pardalos PM, Rajasekaran S, Rolim J (eds) *Randomization methods in algorithmic design*, DIMACS 43. AMS, Providence, pp 133–145
- Martins SL, Resende MGC, Ribeiro CC, Pardalos P (2000) A parallel GRASP for the Steiner tree problem in graphs using a hybrid local search strategy. *J Glob Optim* 17:267–283
- Martins SL, Ribeiro CC, Rossetti I (2004) Applications and parallel implementations of metaheuristics in network design and routing. In: *Applied computing. LNCS* 3285. Springer, Berlin, pp 205–213

- Martins SL, Ribeiro CC, Rossetti I (2006) Applications of parallel metaheuristics to optimization problems in telecommunications and bioinformatics. In: Talbi E-G (ed) Parallel combinatorial optimization. Wiley, New York, pp 301–325
- Mateus GR, Resende MGC, Silva RMA (2011) GRASP with path-relinking for the generalized quadratic assignment problem. *J Heuristics* 17:527–565
- Mavridou T, Pardalos PM, Pitsoulis LS, Resende MGC (1998) A GRASP for the biquadratic assignment problem. *Eur J Oper Res* 105:613–621
- Murphrey RA, Pardalos PM, Pitsoulis LS (1998a) A greedy randomized adaptive search procedure for the multitarget multisensor tracking problem. In: Pardalos PM and Du D-Z (eds) Network design: connectivity and facilities location, DIMACS 40. AMS, Providence, pp 277–301
- Murphrey RA, Pardalos PM, Pitsoulis LS (1998b) A parallel GRASP for the data association multidimensional assignment problem. In: Pardalos PM (ed) Parallel processing of discrete problems. The IMA volumes in mathematics and its applications 106. Springer, Berlin, pp 159–180
- Nascimento MCV, Resende MGC, Toledo FMB (2010) GRASP with path-relinking for the multi-plant capacitated plot sizing problem. *Eur J Oper Res* 200:747–754
- Oliveira CA, Pardalos PM, Resende MGC (2004) GRASP with path-relinking for the quadratic assignment problem. In: Ribeiro CC, Martins SL (eds) Proceedings of III workshop on efficient and experimental algorithms. LNCS 3059. Springer, Berlin, pp 356–368
- Osman IH, Al-Ayoubi B, Barake M (2003) A greedy random adaptive search procedure for the weighted maximal planar graph problem. *Comput Ind Eng* 45:635–651
- Pardalos PM, Pitsoulis LS, Resende MGC (1995) A parallel GRASP implementation for the quadratic assignment problem. In: Ferreira A, Rolim J (eds) Parallel algorithms for irregularly structured problems—IRREGULAR’94. Kluwer, Dordrecht, pp 115–133
- Pardalos PM, Pitsoulis LS, Resende MGC (1996) A parallel GRASP for MAX-SAT problems. LNCS 1184. Springer, Berlin/New York, pp 575–585
- Pardalos PM, Pitsoulis LS, Resende MGC (1997) Algorithm 769: Fortran subroutines for approximate solution of sparse quadratic assignment problems using GRASP. *ACM Trans Math Softw* 23:196–208
- Pardalos PM, Qian T, Resende MGC (1999) A greedy randomized adaptive search procedure for the feedback vertex set problem. *J Comb Optim* 2:399–412
- Piñana E, Plana I, Campos V, Martí R (2004) GRASP and path relinking for the matrix bandwidth minimization. *Eur J Oper Res* 153:200–210
- Pitsoulis LS, Pardalos PM, Hearn DW (2001) Approximate solutions to the turbine balancing problem. *Eur J Oper Res* 130:147–155
- Prais M, Ribeiro CC (2000a) Parameter variation in GRASP procedures. *Investigación Operativa* 9:1–20
- Prais M, Ribeiro CC (2000b) Reactive GRASP: an application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS J Comput* 12:164–176

- Reghioui M, Prins C, Labadi N (2007) GRASP with path relinking for the capacitated arc routing problem with time windows. In: Giacobini M et al (eds) Applications of evolutionary computing. LNCS 4448. Springer, Berlin, pp 722–731
- Resende MGC (1998) Computing approximate solutions of the maximum covering problem using GRASP. *J Heuristics* 4:161–171
- Resende MGC (2008) Metaheuristic hybridization with greedy randomized adaptive search procedures. In: Chen Z-L, Raghavan S (eds) TutORials in operations research. INFORMS, Hanover, pp 295–319
- Resende MGC, Feo TA (1996) A GRASP for satisfiability. In: Johnson DS, Trick MA (eds) Cliques, coloring, and satisfiability: the second DIMACS implementation challenge, DIMACS 26. AMS, Providence, pp 499–520
- Resende LIP, Resende MGC (1999) A GRASP for frame relay permanent virtual circuit routing. In: Ribeiro CC, Hansen P (eds) Extended abstracts of the III metaheuristics International Conference, Angra dos Reis, Brazil, pp 397–401
- Resende MGC, Ribeiro CC (1997) A GRASP for graph planarization. *Networks* 29:173–189
- Resende MGC, Ribeiro CC (2003a) A GRASP with path-relinking for private virtual circuit routing. *Networks* 41:104–114
- Resende MGC, Ribeiro CC (2003b) Greedy randomized adaptive search procedures. In: Glover F, Kochenberger G (eds) Handbook of metaheuristics. Kluwer, Dordrecht, pp 219–249
- Resende MGC, Ribeiro CC (2005a) GRASP with path-relinking: recent advances and applications. In: Ibaraki T, Nonobe K, Yagiura M (eds) Metaheuristics: progress as real problem solvers. Springer, Berlin, pp 29–63
- Resende MGC, Ribeiro CC (2005b) Parallel greedy randomized adaptive search procedures. In: Alba E (ed) Parallel metaheuristics: a new class of algorithms. Wiley, New York, pp 315–346
- Resende MGC, Ribeiro CC (2010) Greedy randomized adaptive search procedures: advances and applications. In: Gendreau M, Potvin J-Y (eds) Handbook of metaheuristics, 2nd edn. Springer, Berlin, pp 293–319
- Resende MGC, Werneck RF (2004) A hybrid heuristic for the p -median problem. *J Heuristics* 10:59–88
- Resende MGC, Werneck RF (2006) A hybrid multistart heuristic for the uncapacitated facility location problem. *Eur J Oper Res* 174:54–68
- Resende MGC, Pardalos PM, Li Y (1996) Algorithm 754: Fortran subroutines for approximate solution of dense quadratic assignment problems using GRASP. *ACM Trans Math Softw* 22:104–118
- Resende MGC, Pitsoulis LS, Pardalos PM (1997) Approximate solution of weighted MAX-SAT problems using GRASP. In: Gu J, Pardalos PM (eds) Satisfiability problems, DIMACS 35. AMS, Providence, pp 393–405
- Resende MGC, Feo TA, Smith SH (1998) Algorithm 787: Fortran subroutines for approximate solution of maximum independent set problems using GRASP. *ACM Trans Math Softw* 24:386–394

- Resende MGC, Pitsoulis LS, Pardalos PM (2000) Fortran subroutines for computing approximate solutions of MAX-SAT problems using GRASP. *Discret Appl Math* 100:95–113
- Resende MGC, Martí R, Gallego, M, Duarte A (2010) GRASP and path relinking for the max-min diversity problem. *Comput Oper Res* 37:498–508
- Ribeiro CC, Resende MGC (1999) Algorithm 797: Fortran subroutines for approximate solution of graph planarization problems using GRASP. *ACM Trans Math Softw* 25:342–352
- Ribeiro CC, Resende MGC (2012) Path-relinking intensification methods for stochastic local search algorithms. *J Heuristics* 18:193–214
- Ribeiro CC, Rossetti I (2002) A parallel GRASP heuristic for the 2-path network design problem. LNCS 2400. Springer, Berlin, pp 922–926
- Ribeiro CC, Rossetti I (2007) Efficient parallel cooperative implementations of GRASP heuristics. *Parallel Comput* 33:21–35
- Ribeiro CC, Souza MC (2000) Tabu search for the Steiner problem in graphs. *Networks* 36:138–146
- Ribeiro CC, Urrutia S (2007) Heuristics for the mirrored traveling tournament problem. *Eur J Oper Res* 179:775–787
- Ribeiro CC, Vianna DS (2005) A GRASP/VND heuristic for the phylogeny problem using a new neighborhood structure. *Int Trans Oper Res* 12:325–338
- Ribeiro CC, Uchoa E, Werneck RF (2002) A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS J Comput* 14:228–246
- Ribeiro MH, Plastino A, Martins SL (2006) Hybridization of GRASP metaheuristic with data mining techniques. *J Math Model Algorithm* 5:23–41
- Ribeiro CC, Martins SL, Rossetti I (2007) Metaheuristics for optimization problems in computer communications. *Comput Commun* 30:656–669
- Ribeiro CC, Rossetti I, Souza RC (2011) Effective probabilistic stopping rules for randomized metaheuristics: GRASP implementations. LNCS 6683. Springer, Berlin, pp 146–160
- Ríos-Mercado RZ, Bard JF (1998) Heuristics for the flow line problem with setup costs. *Eur J Oper Res* 110:76–98
- Ríos-Mercado RZ, Bard JF (1999) An enhanced TSP-based heuristic for makespan minimization in a flow shop with setup costs. *J Heuristics* 5:57–74
- Robertson AJ (2001) A set of greedy randomized adaptive local search procedure (GRASP) implementations for the multidimensional assignment problem. *Comput Optim Appl* 19:145–164
- Santos LF, Martins SL, Plastino A (2008) Applications of the DM-GRASP heuristic: a survey. *Int Trans Oper Res* 15:387–416
- Scaparra M, Church R (2005) A GRASP and path relinking heuristic for rural road network development. *J Heuristics* 11:89–108
- Souza MC, Duhamel C, Ribeiro CC (2004) A GRASP heuristic for the capacitated minimum spanning tree problem using a memory-based local search strategy. In: Resende MGC, de Sousa JP (eds) *Metaheuristics: computer decision-making*. Kluwer, Dordrecht, pp 627–658

- Srinivasan A, Ramakrishnan KG, Kumaram K, Aravamudam M, Naqvi S (2000) Optimal design of signaling networks for Internet telephony. In: IEEE INFO-COM 2000, Tel Aviv, Israel, vol 2, pp 707–716
- Urban TL (1998) Solution procedures for the dynamic facility layout problem. *Ann Oper Res* 76:323–342
- Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. *IEEE Trans Evol Comput* 1:67–82
- Xu JY, Chiu SY (2001) Effective heuristic procedure for a field technician scheduling problem. *J Heuristics* 7:495–509
- Yen J, Carlsson M, Chang M, Garcia JM, Nguyen H (2000) Constraint solving for inkjet print mask design. *J Imaging Sci Technol* 44:391–397

Chapter 12

Variable Neighborhood Search

Pierre Hansen and Nenad Mladenović

12.1 Introduction

Variable neighborhood search (VNS) is a metaheuristic, or framework for building heuristics, which exploits systematically the idea of neighborhood change, both in the descent to local minima and in the escape from the valleys which contain them. In this tutorial we first present the ingredients of VNS, i.e. variable neighborhood descent (VND) and Reduced VNS (RVNS) followed by the basic and then the general scheme of VNS itself which contain both of them. Extensions are presented, in particular Skewed VNS which enhances exploration of far away valleys and variable neighborhood decomposition search (VNDS), a two-level scheme for solution of large instances of various problems. In each case, we present the scheme, some illustrative examples and questions to be addressed in order to obtain an efficient implementation.

Let us consider a combinatorial or global optimization problem

$$\min_{x \in X} f(x) \quad (12.1)$$

where $f(x)$ is the objective function to be minimized and X the set of feasible solutions. A solution $x^* \in X$ is *optimal* if

$$f(x^*) \leq f(x), \forall x \in X. \quad (12.2)$$

An *exact algorithm* for problem (12.1) and (12.2), if one exists, finds an optimal solution x^* , together with the proof of its optimality, or shows that there is no feasible solution, that is $X = \emptyset$. Moreover, in practice, the time to do so should be finite

P. Hansen (✉)

GERAD and HEC Montreal, Montréal, Québec, Canada
e-mail: Pierre.Hansen@gerad.ca

N. Mladenović

School of Mathematics, Brunel University, Uxbridge, Middlesex UB8 3PH, UK

(and not too large); if one deals with a continuous function one must admit a degree of tolerance i.e. stop when a feasible solution x^* has been found such that

$$f(x^*) < f(x) + \varepsilon, \forall x \in X \quad (12.3)$$

or

$$\frac{f(x^*) - f(x)}{f(x^*)} < \varepsilon, \forall x \in X \quad (12.4)$$

for some small positive ε .

Numerous instances of problems of the form (12.1) and (12.2), arising in Operational Research and other fields, are too large for an exact solution to be found in reasonable time. It is well known from complexity theory (Garey and Johnson 1979; Papadimitriou 1994) that thousands of problems are *NP-hard*, that no algorithm with a number of steps polynomial in the size of the instances is known and that finding one for any such problem would entail obtaining one for any and all of them. Moreover, in some cases where a problem admits a polynomial algorithm, the power of this polynomial may be so large that realistic size instances cannot be solved in reasonable time in worst case, and sometimes also in the average case or most of the time.

So one is often forced to resort to heuristics, which yield quickly an approximate solution, or sometimes an optimal solution but without proof of its optimality. Some of these heuristics have a worst-case guarantee, i.e. the solution x_h obtained satisfies

$$\frac{f(x_h) - f(x)}{f(x_h)} \leq \varepsilon, \forall x \in X \quad (12.5)$$

for some ε , which is however rarely small. Moreover, this ε is usually much larger than the error observed in practice and may therefore be a bad guide in selecting a heuristic. In addition to avoiding excessive computing time, heuristics address another problem: local optima. A local optimum x_L of (1) and (2) is such that

$$f(x_L) \leq f(x), \forall x \in N(x_L) \cap X \quad (12.6)$$

where $N(x_L)$ denotes a neighborhood of x_L (ways to define such a neighborhood will be discussed below). If there are many local minima, the range of values they span may be large. Moreover, the globally optimum value $f(x^*)$ may differ substantially from the average value of a local minimum, or even from the best such value among many, obtained by some simple heuristic (a phenomenon called the Tchebycheff catastrophe by Baum 1986). There are, however, many ways to get out of local optima and, more precisely, the valleys which contain them (or set of solutions from which the descent method under consideration leads to them).

Metaheuristics are general frameworks to build heuristics for combinatorial and global optimization problems. For a discussion of the best-known of them the reader is referred to the books of surveys edited by Reeves (1993) and Glover and Kochenberger (2003) as well as to other chapters of the present volume. Some of the many successful applications of metaheuristics are also mentioned there.

VNS ([Mladenović and Hansen 1997](#); [Hansen and Mladenović 1999, 2001c, 2003](#)) is a recent metaheuristic which exploits systematically the idea of neighborhood change, both in descent to local minima and in escape from the valleys which contain them. VNS exploits systematically the following observations:

Fact 1 *A local minimum with respect to one neighborhood structure is not necessarily so for another.*

Fact 2 *A global minimum is a local minimum with respect to all possible neighborhood structures.*

Fact 3 *For many problems local minima with respect to one or several neighborhoods are relatively close to each other.*

This last observation, which is empirical, implies that a local optimum often provides some information about the global one. This may for instance be several variables with the same value in both. However, it is usually not known which ones are such. An organized study of the neighborhood of this local optimum is therefore in order, until a better one is found.

Unlike many other metaheuristics, the basic schemes of VNS and its extensions are simple and require few, and sometimes no parameters. Therefore, in addition to providing very good solutions, often in simpler ways than other methods, VNS gives insight into the reasons for such a performance, which in turn can lead to more efficient and sophisticated implementations.

The chapter is organized as follows. In the next section, we examine the preliminary problem of gathering information about the problem under study, and evaluating it. In Sect. 12.3 the first ingredient of VNS, i.e. VND, which is mostly or entirely deterministic, is studied. Section 12.4 is devoted to the second ingredient, RVNS, which is stochastic. Both ingredients are merged in the basic and the general VNS schemes, described in Sect. 12.5. Extensions are then considered. Skewed VNS, which addresses the problem of getting out of very large valleys, is discussed in Sect. 12.6. Very large instances of many problems cannot be solved globally in reasonable time; VNDS studied in Sect. 12.7 is a two-level scheme which merges VNS with successive approximation (including a two-level VNS). Various tools for analyzing in detail the performance of a VNS heuristic, and then streamlining it are presented in Sect. 12.8. They include *distance-to-target diagrams* and *valley profiles*. In each of these sections basic schemes, or tools, are illustrated by examples from papers by a variety of authors. Questions to be considered in order to get an efficient implementation of VNS are also systematically listed. Promising areas of research are outlined in Sect. 12.9. Brief conclusions complete the chapter in Sect. 12.10. Finally, sources of further information are suggested.

12.2 Preliminaries: Documentation

Once a problem of the form (12.1) and (12.2) has been selected for study and approximate solution by VNS, a preliminary step is to gather in a thorough way the papers written about it or closely related problems. Note that this may be a difficult task as papers are often numerous, dispersed among many journals and volumes of proceedings and the problem may appear (usually under different names) in several fields. Tools such as the *ISI Web of Knowledge*, *NEC Research's Citeseer* or even general web browsers such as *Google* may prove to be very useful.

There are several reasons for studying the literature on the selected problem:

- (i) *Evaluating its difficulty.* Is it NP-hard? Is it strongly NP-hard? (and hence admits no fully polynomial approximation scheme). If it is in P, what is the complexity of the best-known exact algorithm, and is it sufficiently low for realistic instances to be solvable in reasonable time?
- (ii) *Evaluating the performance of previous algorithms.* Are there some instances of (preferably real-word) data for the problem available (e.g. at http://www.informs.org/Resources/Resources/Problem_Instances/)? What are the largest instances solved exactly?
- (iii) *Evaluating the performance of previous heuristics.* Which metaheuristics have been applied to this problem? What are the performances of the resulting heuristics, in terms of size of problems solved, error and computing time (assuming comparison among computing environments, if needed, can be done in a fairly realistic way)?
- (iv) *What steps are used* in the heuristics already proposed? What are the corresponding neighborhoods of the current solution? Are codes for these heuristics available? Are codes for simple descent methods available?

Question (i)'s role is to help to assess the need for a VNS (or others) heuristic for the problem considered. Questions (ii) and (iii) aim at obtaining a benchmark to evaluate the performance of the VNS heuristic when it will be designed and implemented: a good heuristic should obtain optimal solutions for most and preferably all instances solved by an exact algorithm (which suffers from the additional burden of having to prove optimality). Moreover, the new heuristic should do as well as previous ones on most or all instances and substantially better than them on quite a few instances to be viewed as a real progress (doing slightly better on a few instances is just not sufficient).

Question (iv) aims at providing ingredients for the VNS heuristic, notably in its VND component; it also inquires indirectly about directions not yet explored. Incidentally, it raises the question of possible re-use of software, which is reasonable for standard steps, e.g. a descent with Newton's method or a variant thereof.

Initialization.

Choose f, X , neighborhood structure $N(x)$, initial solution x ;

Current step (Repeat).

- (1) Find $x' = \arg \min_{x \in N(x)} f(x)$;
- (2) If $f(x') < f(x)$ set $x' \leftarrow x''$ and iterate; otherwise, stop.

Fig. 12.1 Steepest-descent heuristic

Initialization.

Choose f, X , neighborhood structure $N(x)$, initial solution x ;

Current step (Repeat).

- (1) Find first solution $x' \in N(x)$;
- (2) If $f(x') > f(x)$, find next solution $x'' \in N(x)$; set $x' \leftarrow x''$ and iterate (2); otherwise, set $x \leftarrow x'$ and iterate (1);
- (3) If all solutions of $N(x)$ have been considered, stop.

Fig. 12.2 First-descent heuristic

12.3 Variable Neighborhood Descent

A *steepest-descent* heuristic (known also as *best improvement* local search) consists of choosing an initial solution x , finding a direction of steepest descent from x , within a neighborhood $N(x)$, and moving to the minimum of $f(x)$ within $N(x)$ along that direction; if there is no direction of descent, the heuristic stops, and otherwise it is iterated. This set of rules is summarized in Fig. 12.1.

Observe that a neighborhood structure $N(x)$ is defined for all $x \in X$; in discrete optimization problems it usually consists of all vectors obtained from x by some simple modification, e.g. complementing one or two components of a 0–1 vector. Then, at each step, the neighborhood $N(x)$ of x is explored completely. As this may be time-consuming, an alternative is to use the *first-descent* heuristic. Vectors $x' \in N(x)$ are then enumerated systematically and a move is made as soon as a descent direction is found. This is summarized in Fig. 12.2.

VND is based on Fact 1 of the Introduction, i.e. *a local optimum for a first type of move $x \leftarrow x'$ (or heuristic, or within the neighborhood $N_1(x)$) is not necessary one for another type of move $x \leftarrow \tilde{x}$ (within neighborhood $N_2(x)$)*. It may thus be advantageous to combine descent heuristics. This leads to the basic VND scheme presented in Fig. 12.3.

Caution should be exercised when applying that scheme. In particular one should consider the following questions:

- (i) What complexity do the different moves have?
- (ii) What is the best order in applying them?

Initialization. Select the set of neighborhood structures N_ℓ , for $\ell = 1, \dots, \ell_{\max}$, that will be used in the descent; find an initial solution x (or apply the rules to a given x);

Repeat the following sequence until no improvement is obtained:

- (1) Set $\ell \leftarrow 1$;
- (2) Repeat the following steps until $\ell = \ell_{\max}$:
 - (a) Exploration of neighborhood. Find the best neighbor x' of x ($x' \in N_\ell(x)$);
 - (b) Move or not. If the solution x' thus obtained is better than x , set $x \leftarrow x'$ and $\ell \leftarrow 1$; otherwise, set $\ell \leftarrow \ell + 1$;

Fig. 12.3 Steps of the basic VND

- (iii) Are the moves considered sufficient to ensure a thorough exploration of the region containing x ?
- (iv) How precise a solution is desired?

Question (i) aims at selecting and ranking moves: if they involve too many elementary changes (e.g. complementing three components or more of a 0–1 vector), the resulting heuristic may be very slow and often takes more time than an exact algorithm on small or medium size examples.

Question (ii) also bears upon computing times in relation to the quality of solutions obtained. A frequent implementation consists of ranking moves by order of complexity of their application (which is often synonymous with by size of their neighborhoods $|N_\ell(x)|$), and returning to the first one each time a direction of descent is found and a step made in that direction. Alternatively, all moves may be applied in sequence as long as descent is made for some neighborhood in the series.

Question (iii) is a crucial one: for some problems elementary moves are not sufficient to leave a narrow valley, and heuristics using them only can give very poor results. This is illustrated in Example 12.2.

Finally, the precision desired, as asked for in question (iv) will depend upon whether VND is used alone or within some larger framework, such as VNS itself. In the former case, one will strive to obtain the best solution possible within the allocated computing time; in the latter, one may prefer to get a good solution fairly quickly by the deterministic VND and to improve it later by faster stochastic search in VNS.

Example 12.1 (Simple plant location (see Cornuejols et al. 1990, for a survey)). The simple (or uncapacitated) plant location problem consists of locating a set of facilities i among a given set I of m potential locations, with fixed costs f_i , in order to minimize total costs for satisfying the demand of a given set of users J with delivery costs $c_{ij}, i \in I, j \in J$. It is expressed as follows:

$$\min_{x,y} z_P = \sum_{i=1}^m f_i y_i + \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \quad (12.7)$$

s.t.

$$\sum_{i=1}^m x_{ij} = 1, \forall j \in J \quad (12.8)$$

$$y_i - x_{ij} \geq 0, \forall i \in I, \forall j \in J \quad (12.9)$$

$$y_i \in \{0, 1\}, \forall i \in I \quad (12.10)$$

$$x_{ij} \geq 0, \forall i \in I, \forall j \in J, \quad (12.11)$$

where $y_i = 1$ if a facility is located at i , and 0 otherwise; $x_{ij} = 1$ if demand of user j is satisfied from facility i and 0 otherwise. Note that for fixed y_i , the best solution is defined by

$$x_{ij} = \begin{cases} 1 & \text{if } c_{ij} = \min_{\ell|y_\ell=1} c_{\ell j} \text{ (with minimum index } \ell \text{ in case of ties);} \\ 0 & \text{otherwise.} \end{cases}$$

Therefore neighborhoods can be defined on the y_i , e.g. by Hamming distance (or number of components with complementary values). A first heuristic, Greedy, proceeds by opening a facility ℓ with minimum total cost:

$$f_\ell + \sum_j c_{\ell j} = \min_i \left\{ f_i + \sum_j c_{ij} \right\} \quad (12.12)$$

then letting

$$c_{rj} = \min_{i|y_i=1} c_{ij}, \forall j \quad (12.13)$$

computing the gains g_i obtained by opening a facility at i

$$g_i = \sum_j \max\{c_{rj} - c_{ij}, 0\} - f_i \quad (12.14)$$

and iteratively opening the facility for which the gain is larger, as long as it is positive. Each iteration takes $O(mn)$ time.

Once the Greedy heuristic has been applied, an improved solution may be obtained by the Interchange heuristic which proceeds iteratively to the relocation of one facility at a time in the most profitable way. With an efficient implementation, the idea of which was suggested by Whitaker (1983) for the closely related p -median problem, an iteration of interchange can also be made in $O(mn)$ time.

Applying in turn Greedy and Interchange is a simple case of VND. Further moves in which one facility would be closed and two opened, or two closed and one opened, or two opened and two closed would be too costly if all possible exchanges are examined.

Example 12.2 (Minimum sum-of-squares clustering, MSSC). Given N points $a_\ell \in \mathbb{R}^p$, the MSSC problem consists of partitioning them in M classes (or clusters) C_j such as to minimize the sum of squared distances between the points and the centroids \bar{x}_i of their clusters:

$$\min \sum_{i=1}^m \sum_{\ell:a_\ell \in C_i} \|a_\ell - \bar{x}_i\|^2 \quad (12.15)$$

where

$$\bar{x}_i = \frac{1}{|C_i|} \sum_{\ell: a_\ell \in C_i} a_\ell \quad (12.16)$$

and $\|\cdot\|$ denotes the Euclidean norm.

Traditional heuristics for MSSC are: (i) H-means, which proceeds from an initial partition by moving one entity x_ℓ from its cluster to another one, in a greedy way, until no further move decreases the objective function value, and (ii) K-means, which proceeds from an initial partition by, alternatingly, finding the centroids of its clusters, and reassigning entities to the closest centroid, until stability is attained.

Computational experiments (Hansen and Mladenović 2001) show that both H-means and K-means may lead to very poor results for instances with large M and N (the relative error being sometimes greater than 100 %). This is due to bad exploration of X , or in other words, to difficulties in leaving valleys. A new *jump* move, defined as the displacement of a centroid to a point a_ℓ which does not coincide with a centroid, leads to a new VND heuristic, called J-means, which improves very substantially on both H-means and K-means.

12.4 Reduced VNS

Assume a local minimum x of f has been reached. One would then like to leave its valley, and find another deeper one. In the standard versions of VNS, no previous knowledge of the landscape is assumed, or exploited. (Note that interesting hybrids could be built, using also values of $f(x)$ at previous iteration points x). Then, the questions to be asked are

- (i) Which direction to go?
- (ii) How far?
- (iii) How should one modify moves if they are not successful?

Question (i) bears upon the possibility of reaching any feasible point $x \in X$, or every valley; the simplest answer is to choose a direction at random. For problems in 0–1 variables this will amount to complementing some variables; for continuous Euclidean problems, drawing angular coefficients at random (or, in other words, choosing at random a point on the unit ball around x) takes all points of X into account.

Question (ii) is crucial. Indeed one wants to exploit to the limit Fact 2 of the Introduction, i.e. in many combinatorial and global optimization problems, local optima tend to be close one to another and situated in one (or sometimes several) small parts of X . So once a local optimum has been reached, it contains implicit information about close better, and perhaps globally optimum, ones. It is then natural to explore first its vicinity. But, if the valley surrounding the local optimum x is large, this may not be sufficient, and what to do next is asked for in question (iii). Again a natural answer is to go further.

Initialization. Select the set of neighborhood structures \mathcal{N}_k , for $k = 1, \dots, k_{\max}$, that will be used in the search; find an initial solution x ; choose a stopping condition;

Repeat the following sequence until the stopping condition is met:

(1) Set $k \leftarrow 1$;

(2) Repeat the following steps until $k = k_{\max}$:

(a) Shaking. Generate a point x' at random from the k th neighborhood of x ($x' \in \mathcal{N}_k(x)$);

(b) Move or not. If this point is better than the incumbent, move there ($x \leftarrow x'$), and continue the search with \mathcal{N}_1 ($k \leftarrow 1$); otherwise, set $k \leftarrow k + 1$;

Fig. 12.4 Steps of the Reduced VNS

These aims are pursued in the RVNS scheme, presented in Fig. 12.4. A set of neighborhoods $N_1(x), N_2(x), \dots, N_{k_{\max}}(x)$ will be considered around the current point x (which may be or not a local optimum). Usually, these neighborhoods will be nested, i.e. each one contains the previous. Then a point is chosen at random in the first neighborhood. If its value is better than that of the incumbent (i.e. $f(x') < f(x)$), the search is re-centered there ($x \leftarrow x'$). Otherwise, one proceeds to the next neighborhood. After all neighborhoods have been considered, one begins again with the first, until a stopping condition is satisfied (usually it will be maximum computing time since the last improvement, or maximum number of iterations).

Due to the nestedness property the size of successive neighborhoods will be increasing. Therefore, one will explore more thoroughly close neighborhoods of x than farther ones, but nevertheless search within these when no further improvements are observed within the first, smaller ones.

Example 12.3 (p-median (see Labb   et al. 1995 for a survey)). This is a location problem very close to that of Simple Plant Location. The differences are that there are no fixed costs, and that the number of facilities to be opened is set at a given value p . It is expressed as follows:

$$\min \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \quad (12.17)$$

subject to

$$\sum_{i=1}^m x_{ij} = 1, \quad \forall j \quad (12.18)$$

$$y_i - x_{ij} \geq 0, \quad \forall i, j \quad (12.19)$$

$$\sum_{i=1}^m y_i = p \quad (12.20)$$

$$x_{ij}, y_i \in \{0, 1\}. \quad (12.21)$$

The Greedy and Interchange heuristics described above for Simple Plant Location are easily adapted to the p -median problem and, in fact, the latter was early proposed by Teitz and Bart (1967).

Table 12.1 5934-customer p -median problem

p	Obj. value <i>Best known</i>	CPU times			% Error		
		FI	RVNS	VNDS	FI	RVNS	VNDS
100	27,33,817.25	6,637.48	510.20	6,087.75	0.36	0.15	0.00
200	18,09,064.38	14,966.05	663.69	14,948.37	0.79	0.36	0.00
300	13,94,715.12	20,127.91	541.76	17,477.51	0.65	0.51	0.00
400	11,45,669.38	23,630.95	618.62	22,283.04	0.82	0.59	0.00
500	9,74,275.31	29,441.97	954.10	10,979.77	0.98	0.51	0.00
700	7,52,068.38	36,159.45	768.84	32,249.00	0.64	0.50	0.00
800	6,76,846.12	38,887.40	813.38	20,371.81	0.61	0.53	0.00
900	6,13,367.44	41,607.78	731.71	27,060.09	0.55	0.53	0.00
1,000	5,58,802.38	44,176.27	742.70	26,616.96	0.73	0.66	0.00
Average		28,403.90	705.00	19,786.00	0.68	0.48	0.00

Initialization. Select the set of neighborhood structures \mathcal{N}_k , for $k = 1, \dots, k_{\max}$, that will be used in the search; find an initial solution x ; choose a stopping condition;

Repeat the following sequence until the stopping condition is met:

(1) Set $k \leftarrow 1$;

(2) Repeat the following steps until $k = k_{\max}$:

(a) Shaking. Generate a point x' at random from the k th neighborhood of x ($x' \in \mathcal{N}_k(x)$);

(b) Local search. Apply some local search method with x' as initial solution; denote with x'' the so obtained local optimum;

(c) Move or not. If the local optimum x'' is better than the incumbent x , move there ($x \leftarrow x''$), and continue the search with \mathcal{N}_l ($k \leftarrow 1$); otherwise, set $k \leftarrow k + 1$;

Fig. 12.5 Steps of the basic VNS

Fast interchange, using Whitaker's (1983) data structure applies here also (Hansen and Mladenović 1997). Refinements were proposed by Resende and Werneck (2002). A comparison between that approach and RVNS is made in Hansen et al. (2001), and the results are summarized in Table 12.1. It appears that RVNS gives better results than Fast Interchange in 2.5 % of the time.

12.5 Basic and General VNS

In the previous two sections, we examined how to use variable neighborhoods in descent to a local optimum and in finding promising regions for near-optimal solutions. Merging the tools for both tasks leads to the General VNS scheme. We first discuss how to combine a local search with systematic changes of neighborhoods around the local optimum found. We then obtain the Basic VNS scheme presented in Fig. 12.5.

According to this basic scheme, a series of neighborhood structures, which define neighborhoods around any point $x \in X$ of the solution space, are first selected. Then

Initialization. Select the set of neighborhood structures \mathcal{N}_k , for $k = 1, \dots, k_{\max}$, that will be used in the shaking phase, and the set of neighborhood structures N_ℓ for $\ell = 1, \dots, \ell_{\max}$ that will be used in the local search; find an initial solution x and improve it by using RVNS; choose a stopping condition;

Repeat the following sequence until the stopping condition is met:

(1) Set $k \leftarrow 1$;

(2) *Repeat* the following steps until $k = k_{\max}$:

(a) *Shaking.* Generate a point x' at random from the k th neighborhood $\mathcal{N}_k(x)$ of x ;

(b) *Local search by VND.*

(b1) Set $\ell \leftarrow 1$;

(b2) *Repeat* the following steps until $\ell = \ell_{\max}$:

· *Exploration of neighborhood.* Find the best neighbor x'' of x' in $N_\ell(x')$;

· *Move or not.* If $f(x'') < f(x')$ set $x' \leftarrow x''$ and $\ell \leftarrow 1$; otherwise set $\ell \leftarrow \ell + 1$;

(c) *Move or not.* If this local optimum is better than the incumbent, move there ($x \leftarrow x''$), and continue the search with \mathcal{N}_1 ($k \leftarrow 1$); otherwise, set $k \leftarrow k + 1$;

Fig. 12.6 Steps of the general VNS

the local search is used and leads to a local optimum x . A point x' is selected at random within the first neighborhood $\mathcal{N}_1(x)$ of x and a descent from x' is done with the local search routine. This leads to a new local minimum x'' . At this point, three outcomes are possible: (i) $x'' = x$, i.e. one is again at the bottom of the same valley; in this case the procedure is iterated using the next neighborhood $\mathcal{N}_k(x)$, $k \geq 2$; (ii) $x'' \neq x$ but $f(x'') \geq f(x)$, i.e. another local optimum has been found, which is not better than the previous best solution (or incumbent); in this case too the procedure is iterated using the next neighborhood; (iii) $x'' \neq x$ and $f(x'') < f(x)$ i.e. another local optimum, better than the incumbent has been found; in this case the search is recentered around x'' and begins again with the first neighborhood. Should the last neighborhood be reached without a solution better than the incumbent being found, the search begins again at the first neighborhood $\mathcal{N}_1(x)$ until a stopping condition, e.g. a maximum time or maximum number of iterations or maximum number of iterations since the last improvement, is satisfied.

If instead of simple local search, one uses VND and if one improves the initial solution found by Reduced VNS, one obtains the General VNS scheme. This scheme is presented in Fig. 12.6.

Several questions about selection of neighborhood structures are in order:

- (i) What properties of the neighborhoods are mandatory for the resulting scheme to be able to find a globally optimal or near-optimal solution?
- (ii) What properties of the neighborhoods will favor finding a near-optimal solution?
- (iii) Should neighborhoods be nested? Otherwise how should they be ordered?
- (iv) What are desirable properties of the sizes of neighborhoods?

The first two questions bear upon the ability of the VNS heuristic to find the best valleys, and to do so fairly quickly. To avoid being blocked in a valley, while there

may be deeper ones, the union of the neighborhoods around any feasible solution x should contain the whole feasible set:

$$X \subseteq \mathcal{N}_1(x) \cup \mathcal{N}_2(x) \cup \dots \cup \mathcal{N}_{k_{\max}}(x), \quad \forall x \in X.$$

These sets may cover X without necessarily partitioning it, which is easier to implement, e.g. when using nested neighborhoods, i.e.

$$\mathcal{N}_1(x) \subset \mathcal{N}_2(x) \subset \dots \subset \mathcal{N}_{k_{\max}}(x), \quad X \subset \mathcal{N}_{k_{\max}}(x), \quad \forall x \in X.$$

If these properties do not hold, one might still be able to explore X completely, by traversing small neighborhoods around points on some trajectory, but it is not guaranteed. To illustrate, as mentioned before in MSSC, the neighborhoods defined by moving an entity (or even a few entities) from one cluster to another one are insufficient to get out of many local optima. Moving centers of clusters does not pose a similar problem.

Nested neighborhoods are easily obtained for many combinatorial problems by defining a first neighborhood $\mathcal{N}_1(x)$ by a type of move—e.g. 2-opt in the traveling salesman problem (TSP)—and then iterating it k times to obtain neighborhoods $\mathcal{N}_k(x)$ for $k = 2, \dots, k_{\max}$. They have the property that their sizes are increasing. Therefore if, as is often the case, one goes many times through the whole sequence of neighborhoods the first ones will be explored more thoroughly than the last ones. This is desirable in view of Fact 3 mentioned in the Introduction, i.e. that local optima tend to be close one from another.

Restricting moves to the feasible set X may be too constraining, particularly if this set is disconnected. Introducing some or all constraints in the objective function with Lagrangian multipliers, makes it possible to move to infeasible solutions. A variant of this idea is to penalize infeasibility, e.g. pairs of adjacent vertices to which the same color is assigned in graph coloring (see Zufferey et al. 2003).

Example 12.4 (Scheduling workover rigs for onshore oil production). Many oil wells in onshore fields rely on artificial lift methods. Maintenance services such as cleaning and others, which are essential to these wells, are performed by workover rigs. They are slow mobile units and, due to their high operation costs, there are relatively few workover rigs when compared with the number of wells demanding service. The problem of scheduling workover rigs consists in finding the best schedule S_i ($i = 1, \dots, m$) of the m workover rigs to attend all wells demanding maintenance services, so as to minimize the oil production loss (production before maintenance being reduced).

In Aloise et al. (2003) a basic VNS heuristic is developed for solving the Scheduling of Workover Rigs Problem. Initial schedule S_i (where S_i is an ordered set of wells serviced by workover rig i), is obtained by a *Greedy* constructive heuristic. For the shaking step $k_{\max} = 9$ neighborhoods are constructed: (1) *Swap routes* (SS): the wells and the associated routes assigned to two workover rigs are interchanged; (2) *Swap wells from the same workover rig* (SWSW): the order in which two wells are serviced by the same rig is swapped; (3) *Swap wells from different workover rig* (SWDW): two wells assigned to two different workover rigs are swapped;

Table 12.2 Average results with eight workover rigs over 20 runs of each synthetic test problem and three possible scenarios (from Aloise et al. 2003)

Problem	GA	GRASP	AS	MMAS	VNS
P-111	16,791.87	16,602.51	15,813.53	15,815.26	15,449.50
P-211	20,016.14	19,726.06	19,048.13	19,051.61	18,580.64
P-311	20,251.93	20,094.37	19,528.93	19,546.10	19,434.97

(4) *Add/Drop* (AD): a well assigned to a workover rig is reassigned to any position of the schedule of another workover rig; (5) (SWSW)²: apply twice the SWSW move; (6) (SWDW)²: apply twice the SWDW move; (7) (SWDW)³: apply three times the SWDW move; (8) (AD)²: successively apply two (AD) moves; (9) (AD)³: successively apply three (AD) moves.

For local search, the neighborhood consists of all possible exchanges of pairs of wells, i.e. the union of (SWSW) and (SWDW) from above are used.

A basic VNS is compared with a genetic algorithm, the greedy randomized adaptive procedure (GRASP) and two ant colony methods (AS and MMAS) on synthetical and real-life problems from Brazilian onshore fields. Some results on synthetical data are given in Table 12.2. On 27 possible scenarios in generating data sets (denoted by P-111, P-112, P-113, P-121, ..., P33), VNS was better than others in 85 % of the cases and MMAS in 15 %. For real-life problems, results were much better than the gains expected. For example, a daily reduction of 109 m³ (equivalent to 685.6 bbl) in the production losses over 15 days was obtained by VNS compared with Petrobras' previous solution. That leads to a total savings estimated at 6,600,000 US dollars a year.

12.6 Skewed VNS

VNS gives usually better (or as good) solutions than multistart, and much better ones when there are many local optima. This is due to Fact 3 of the Introduction: many problems have clustered local optima; often, their objective function is a globally convex one plus some noise. However, it may happen that some instances have several separated and possibly far apart valleys containing near-optimal solutions. If one considers larger and larger neighborhoods, the information related to the currently best local optimum dissolves and VNS degenerates into multistart. Moreover if the current best local optimum is not in the deepest valley this information is in part irrelevant. It is therefore of interest to modify VNS schemes in order to explore more fully valleys which are far away from the incumbent solution. This will be done by accepting to re-center the search when a solution close to the best one known, but not necessarily as good, is found, provided that it is far from this last solution. The modified VNS scheme for this variant, called Skewed VNS (SVNS) is

Initialization. Select the set of neighborhood structures \mathcal{N}_k , for $k = 1, \dots, k_{\max}$, that will be used in the search; find an initial solution x and its value $f(x)$; set $x_{\text{opt}} \leftarrow x$, $f_{\text{opt}} \leftarrow f(x)$; choose a stopping condition and a parameter value α ;

Repeat the following until the stopping condition is met:

- (1) Set $k \leftarrow 1$;
- (2) Repeat the following steps until $k = k_{\max}$:
 - (a) Shaking. Generate a point x' at random from the k th neighborhood of x ;
 - (b) Local search. Apply some local search method with x' as initial solution; denote with x'' the so-obtained local optimum;
 - (c) Improvement or not. If $f(x'') < f_{\text{opt}}$ set $f_{\text{opt}} \leftarrow f(x'')$ and $x_{\text{opt}} \leftarrow x''$;
 - (d) Move or not. If $f(x'') - \alpha \rho(x, x'') < f(x)$ set $x \leftarrow x''$ and $k \leftarrow 1$; otherwise set $k \leftarrow k + 1$.

Fig. 12.7 Steps of the Skewed VNS

presented in Fig. 12.7. The relaxed rule for re-centering uses an evaluation function linear in the distance from the incumbent, i.e. $f(x'')$ is replaced by

$$f(x'') - \alpha \rho(x, x'')$$

where $\rho(x, x'')$ is the distance from x to x'' and α a parameter. A metric for distance between solutions is usually easy to find, e.g. the Hamming distance when solutions are described by Boolean vectors or the Euclidean distance in the continuous case. Clearly, more complicated formulas could be used for re-centering; possibly, one might take into account known values at points already visited in the valley being explored.

Questions to be answered when applying SVNS are the following:

- (i) Does the problem under consideration have a roughly convex objective function, or are there several far apart deep valleys?
- (ii) How should α be chosen?

These questions can be answered, to some extent, by first using a multistart version of VNS, i.e. starting VNS from various random points and running it for a short time. Then one can look at the position of the best local optima found and see if they are clustered or dispersed. Further, one can plot values in function of distance from the corresponding local optima to the best known solution and choose α as a fraction of the average slope.

Example 12.5 (Weighted maximum satisfiability, WMAX-SAT). The satisfiability problem, in clausal form, consists in determining if a given set of m clauses (all in disjunctive or all in conjunctive form) built upon n logical variables has a solution or not. The maximum satisfiability problem consists in finding a solution satisfying the largest possible number of clauses. In the weighted maximum satisfiability problem (WMAXSAT) positive weights are assigned to the clauses and a solution maximizing the sum of weights of satisfied clauses is sought. Results of comparative experiments with VNS and tabu search (TS) heuristics on instances having 500

Table 12.3 Results for GERAD test problems for WMAXSAT ($n = 500$)

	VNS	VNS-low	SVNS-low	TS
Number of instances where best solution is found	6	4	23	5
% average error in 10 trials	0.2390	0.2702	0.0404	0.0630
% best error in 10 trials	0.0969	0.1077	0.0001	0.0457
Total number of instances	25	25	25	25

Initialization. Select the set of neighborhood structures \mathcal{N}_k , for $k = 1, \dots, k_{\max}$, that will be used in the search; find an initial solution x ; choose a stopping condition;

Repeat the following sequence until the stopping condition is met:

- (1) Set $k \leftarrow 1$;
- (2) Repeat the following steps until $k = k_{\max}$:
 - (a) Shaking. Generate a point x' at random from the k th neighborhood of x ($x' \in \mathcal{N}_k(x)$); in other words, let y be a set of k solution attributes present in x' but not in x ($y = x' \setminus x$).
 - (b) Local search. Find a local optimum in the space of y either by inspection or by some heuristic; denote the best solution found with y' and with x'' the corresponding solution in the whole space S ($x'' = (x' \setminus y) \cup y'$);
 - (c) Move or not. If the solution thus obtained is better than the incumbent, move there ($x \leftarrow x''$), and continue the search with \mathcal{N}_j ($k \leftarrow 1$); otherwise, set $k \leftarrow k + 1$;

Fig. 12.8 Steps of the basic VNDS

variables, 4,500 clauses and three variables per clause, in direct or complemented form, are given in Table 12.3 from Hansen et al. (2001). It appears that using a restricted neighborhood consisting of a few directions of steepest descent or mildest ascent in the Shaking step does not improve results, but using this idea in conjunction with SVNS improves notably upon results of basic VNS and also upon those of a TS heuristic.

12.7 Variable Neighborhood Decomposition Search

The VNDS method (Hansen et al. 2001) extends the basic VNS into a two-level VNS scheme based upon decomposition of the problem. Its steps are presented in Fig. 12.8.

Note that the only difference between the basic VNS and VNDS is in step 2b: instead of applying some local search method in the whole solution space S (starting from $x' \in \mathcal{N}_k(x)$), in VNDS we solve at each iteration a subproblem in some subspace $V_k \subseteq \mathcal{N}_k(x)$ with $x' \in V_k$. When the local search used in this step is also VNS, the two-level VNS scheme arises.

VNDS can be viewed as embedding the classical successive approximation scheme in the VNS framework.

12.8 Analyzing Performance

When a first VNS heuristic has been obtained and tested, the effort should not stop there. Indeed, it is often at this point that the most creative part of the development process takes place. It exploits systematically Fact 2 of the Introduction, that global minima are local minima for all possible neighborhoods simultaneously. The contrapositive is that if a solution $x \in X$ is a local minimum (for the current set of neighborhoods) and not a global one there are one or several neighborhoods (or moves) to be found, which will bring it to this global optimum.

The study then focuses on instances for which an optimal solution is known (or, if none or very few are available, on instances with a presumably optimal solutions, i.e. the best one found by several heuristics) and compares it with the heuristic solution obtained. Visualization is helpful and may take the form of a *distance-to-target diagram* ([Hansen and Mladenović 2003](#)). Then, the heuristic solutions, the optimal one and their symmetric difference (e.g. for the TSP) are represented on screen. Moreover, an interactive feature makes it possible to follow how the heuristic works step by step. The information thus gathered is much more detailed than one would get just from objective values and computer times if, as is often the case, the heuristic is viewed as a black box. For instance, this clearly shows that 2-opt is not sufficient to get a good solution for the TSP, that moves involving three or four edges are needed and that those edges leading to an improvement may be far apart along the tour. For another application of VNS to the TSP see [Burke et al. \(1999\)](#).

Similarly, for location problems, one can focus on those facilities which are not at their optimal location and study why, in terms of distributions of nearby users.

Another point is to study how to get out of a large valley if there exists another promising one. *Valley* (or *mountain*) *profiles* are then useful ([Hansen et al. 2001](#)). They are obtained by drawing many points x' at random within nested neighborhoods $\mathcal{N}_1(x), \mathcal{N}_2(x), \dots$ (or, which is equivalent, at increasing distance of a local minimum x) then performing one VND descent and plotting probabilities to get back to x , to get to another local minimum x'' with a value $f(x'') \geq f(x)$ or to get to an improved local minimum x' with $f(x') < f(x)$. Alternately one may also study the probabilities to go in the direction of x , i.e. $p(x, x'') \leq p(x, x')$ or towards another valley i.e. $p(x, x'') > p(x, x')$.

12.9 Promising Areas of Research

Research on Variable Neighborhood Search and its applications is currently very active. We review some of the promising areas in this section; these include a few which are barely explored yet.

A first set of areas concerns enhancements of the VNS basic scheme and ways to make various steps more efficient.

- (a) *Initialization.* Both VND and VNS, as many other heuristics, require an initial solution. Two questions then arise: *How best to choose it?* and *Does it matter?* For instance, many initialization rules have been proposed for the k -means heuristic for minimum sum-of-squares clustering, described above, 25 such rules are compared in Hansen et al. (2003a). It appears that while sensitivity of k -means to the initial solution is considerable (best results being obtained with Ward's hierarchical clustering method), VNS results depend very little on the chosen rule. The simplest one is thus best. It would be interesting to extend and generalize this result by conducting similar experiments for other problems.
- (b) *Inventory of neighborhoods.* As mentioned above, a VNS study begins by gathering material on neighborhoods used in previous heuristics for the problem under study. A systematic study of moves (or neighborhoods) used for heuristics for whole classes of problems (e.g. location, network design, routing, ...) together with the data structures most adequate for their implementation should be of basic interest for VNS as well as for other metaheuristics. Several researchers (e.g. Ahuja et al. 2000) are working in that direction.
- (c) *Distribution of neighborhoods.* When applying a General VNS scheme, neighborhoods can be used in the local search phase, in the shaking phase or in both. A systematic study of their best distribution between phases can enhance performance and provide further insight in the solution process. In particular, the trade-off between increased work in the descent, which provides better local optima, and in shaking which leads to better valleys should be focused upon.
- (d) *Ancillary tests.* VNS schemes use randomization in their attempts to find better solutions. This also avoids possible cycling. However, many moves may not lead to any improvement. This suggests adding an ancillary test (Hansen 1974, 1975) the role of which is to decide if a move should be used or not, in its general or in a restricted form. Considering again MMSC, one could try to select better the centroid to be removed from the current solution (a possible criterion being that its cluster contains a few entities only or is close to another centroid) as well as the position where it will be assigned (e.g. the location of an entity far from any other centroid and in a fairly dense region).

A second set of areas concerns changes to the Basic VNS scheme.

- (e) *Use of memory.* VNS in its present form relies only on the best solutions currently known to center the search. Knowledge of previous good solutions is forgotten, but might be useful to indicate promising regions not much explored yet. Also, characteristics common to many or most good solutions, such as variables taking the same value in all or most such solutions could be used to better focus the shaking phase. Use of memory has been much studied in tabu search and other metaheuristics. The challenge for VNS is to introduce memory while keeping simplicity.

An interesting way to use memory to enhance performance is *Reactive VNS*, explored by Braisy (2001) for the vehicle routing problem with time windows. If some constraints are hard to satisfy, their violation may be penalized more frequently than for others in the solution process.

- (f) *Parallel VNS.* Clearly, there are many natural ways to parallelize VNS schemes. A first one, within VND, is to perform local search in parallel. A second one, within VNS, is to assign the exploration of each neighborhood of the incumbent to a different processor. A third one, within VNDS, is to assign a different subproblem to each processor. [Lopez et al. \(2002\)](#) explore several options in designing a parallel VNS.
- (g) *Hybrids.* Several researchers (e.g. [Rodriguez et al. 1999](#); [Festa et al. 2001](#); [Ribeiro et al. 2001](#)) have combined VNS with other metaheuristics for various problems. Again, this is not always easy to do without losing VNS's simplicity but may lead to excellent results, particularly if the other metaheuristics are very different from VNS.

At a more general level, one might wish to explore combinations of VNS with constraint programming, instead of its development within mathematical programming as in the applications described above. This could be done in two ways: on the one hand, techniques from constraint programming could be applied to enhance VND; on the other hand, VNS could be applied to constraint programming by minimizing a sum of artificial variables measuring infeasibility and possibly weighted by some estimate of the difficulty of satisfying the corresponding constraints.

A third set of areas concerns new aims for VNS, i.e. non-standard uses:

- (h) *Solutions with bounds on the error.* VNS, as other metaheuristics, most often provides near-optimal solutions to combinatorial problems, without bounds on their error. So while such solutions may be optimal or very close to optimality, this fact cannot be recognized. One approach to obtain such bounds is to find with VNS a heuristic solution of the primal problem, deduce from it a solution to the dual (or its continuous relaxation) and then improve this dual solution by another application of VNS. Moreover, complementary slackness conditions can be used to simplify the dual. For problems with a small duality gap this may lead to a near-optimal solution, guaranteed to be very close to optimality. To illustrate, recent work of [Hansen et al. \(2003a\)](#) on the simple plant location problem (SPLP) gave solutions to instances with up to 15,000 users and 15,000 possible facilities with an error bounded by 0.05 %.
- (i) *Using VNS within exact algorithms for mixed-integer programming.* Sophisticated algorithms for mixed-integer programming often contain various phases where heuristics are applied. This is illustrated, for example, by [Desaulniers et al. \(2001\)](#) for the airline crew scheduling problem.

Extending the results described in the previous section, in the branch-and-bound framework led to solve exactly SPLP instances with up to 7,000 users ([Hansen et al. 2007](#)).

A different approach, called *local branching*, has been recently proposed by [Fischetti and Lodi \(2003\)](#) and [Fischetti et al. \(2003\)](#), both for exact and approximate resolution of large mixed-integer programs. At various branches in the branch-and-bound tree, cuts (which are not valid in general) are added; they express that among a given set of 0–1 variables, already at an integer value, only a few may change their value. They thus correspond to neighborhoods defined by the Hamming distance.

Then CPLEX is used to find the optimal solution within the neighborhood and in this way feasible solutions are more easily obtained. Improved solutions were obtained for a series of large mixed-integer programming instances from various sources, when local branching idea is combined with VNS ([Hansen et al. 2006](#); [Lazic et al. 2010](#)).

- (j) *Artificial intelligence: enhancing graph theory with VNS.* VNS, as other metaheuristics, has been extensively used to solve a variety of optimization problems in graph theory. However, it may also be used to enhance graph theory per se, following an artificial intelligence approach. This is done by the AutoGraphiX (AGX) system developed by [Caporossi and Hansen \(2000, 2003\)](#). This system considers a graph invariant (i.e. a quantity defined for all graphs of the class under study and independent of vertex and edge labelings) or a formula involving several invariants (which is itself a graph invariant). Then AGX finds extremal or near-extremal graphs for that invariant parametrizing on a few variables, often the order n (or number of vertices) and the size m (of number of edges) of the graph. Analyzing automatically or interactively these graphs and the corresponding curves of invariant values leads to the finding of new conjectures, refuting, corroborating or strengthening existing ones, and giving hints about a possible proof from the minimal list of moves needed to find the extremal graphs. To illustrate, the *energy* E of a graph is the sum of absolute values of the eigenvalues of its adjacency matrix. The following relations were obtained by [Caporossi et al. \(1999\)](#) with AGX: $E \geq 2\sqrt{m}$ and $E \geq \frac{4m}{n}$ and were easily proved. Over 70 new relations have now been obtained, in mathematics and in chemistry. Three ways to attain full automation based on the mathematics of principal component analysis, linear programming and recognition of extremal graphs together with formula manipulations are currently being studied.

12.10 Tricks of the Trade

12.10.1 Getting Started

This purpose of this section is to help students in making a first very simple version of VNS, which would not necessarily be competitive with later more sophisticated versions. Most of the steps are common to the implementation of other metaheuristics.

12.10.1.1 A Step-by-Step Procedure

1. *Become familiar with the problem.* Think about the problem at hand; in order to understand it better, make a simple numerical example and spend some time in trying to solve it by hand in your own way. Try to understand why the problem is hard and why a heuristic is needed.

2. *Read literature.* Read about the problem and solution methods in the literature.
3. *Test instances (read data).* Use your numerical example as a first instance for testing your future code, but if it is not large enough, take some from the web, or make a routine for generating random instances. In the second case, read how to generate events using uniformly distributed numbers from $(0,1)$ interval (since each programming language has statement for getting such random numbers).
4. *Data structure.* Think about how the solution of the problem will be represented in the memory. Consider two or more presentations of the same solution if they can reduce the complexity of some routines, i.e. analyze the advantages and disadvantages of each possible presentation.
5. *Initial solution.* Having a routine for reading or generating input data of the problem, the next step is to get an initial solution. For the simple version, any random feasible solution may be used, but the usual way is to develop some *greedy* constructive heuristic, which should not be very hard to do.
6. *Objective value.* Make a procedure that calculates the objective function value for a given solution. Notice that at this stage we already have all the ingredients for the Monte Carlo method: generation of random solution and calculation of objective function value. Get a solution to your problem by Monte Carlo heuristic (i.e. repeat steps 5 and 6 many times and keep the best result).
7. *Shaking.* Make a procedure for shaking. This is a key step of VNS. However, it is easy to implement and usually has only a few lines of computer code. For example, in solving the multi-source Weber problem (see Example 12.2), the easiest perturbation of the current solution is to reallocate randomly chosen entity ℓ from its cluster to another one, also chosen at random. In fact, in this case, the shaking step (or jump in the k th neighborhood) would have only three lines of the computer code:

```

For  $i = 1$  to  $k$ 
   $a(1 + n \cdot Rnd1) = 1 + m \cdot Rnd2$ 
EndFor

```

The solution is saved in the array $a(\ell) \in \{1, \dots, m\}$ that denotes membership or allocation of entity ℓ ($\ell = 1, \dots, n$); $Rnd1$ and $Rnd2$ denote random numbers uniformly distributed from the $(0,1)$ interval. Compare the results of the obtained Reduced VNS (take $k_{\max} = 2$) with the Monte Carlo method.

8. *Local search.* Choose an off-the-shelf local search heuristic (or develop a new one). In building a new local search, consider several usual moves that define the neighborhood of the solution: *drop*, *add*, *swap*, *interchange*, etc. Also, for the efficiency (speed) of the method, it is very important to pay special attention to *updating* of the incumbent solution. In other words, it is not usually necessary to use a procedure for calculating objective function values for each point in the neighborhood, i.e. it is possible to get those values by very simple calculation.
9. *Comparison.* Include a local search routine in RVNS to get the basic VNS, and compare it with other methods from the literature.

12.10.2 More Tips

Sometimes basic VNS does not provide very good results.

1. *First versus best improvement.* Compare experimentally *first* and *best improvement* strategies within local search. Previous experience suggests the following: if your initial solution is chosen at random, use first improvement, but if some constructive heuristic is used, use best improvement rule.
2. *Reduce the neighborhood.* The reason for the bad behavior of any local search may be unnecessary visits to all solutions in the neighborhood. Try to identify a *promising* subset of the neighborhood and visit only those; ideally, find a rule that automatically selects solutions from the neighborhood whose objective values are not better than the current one.
3. *Intensified shaking.* In developing more effective VNS, one must spend some time in checking how sensitive the objective function is to small change (shake) of the solution. The trade-off between intensification and diversification of the search in VNS is balanced in the shaking procedure. For some problem instances completely random jump in the k th neighborhood is too diversified. In such cases, some *intensify shaking* procedure may increase intensification of the search. For example, k -interchange neighborhood may be reduced by repeating k times *random add* followed by *best drop* moves. A special case of intensified shaking is so-called large neighborhood search, where k randomly chosen attributes of the solutions are destroyed (dropped), and then the solution is re-built in the best way (by some constructive heuristic).
4. *VND.* Analyze several possible neighborhood structures, estimate their size, make order of them, i.e. develop VND and replace the local search routine with VND to get general VNS.
5. *Experiments with parameter settings.* The single parameter of VNS is k_{\max} , which should be estimated experimentally. However, the procedure is usually not very sensitive on k_{\max} . In order to make a parameter-free VNS, one can fix its value at the value of some input parameter: e.g. for the p -median (Example 12.3), $k_{\max} = p$; for the MSSC (Example 12.2), $k_{\max} = m$.

12.11 Conclusions

The general schemes of VNS have been presented, discussed and illustrated by examples. References to many further successful applications are given in the next section. In order to evaluate a VNS research program, one needs a list of desirable properties of metaheuristics. The following eight are identified by [Hansen and Mladenović \(2003\)](#):

1. *Simplicity.* The metaheuristic should be based on a simple and clear principle, which should be largely applicable;

2. *Precision.* Steps of the metaheuristic should be formulated in precise mathematical terms, independent from the possible physical or biological analogy which was an initial source of inspiration;
3. *Coherence.* All steps of heuristics for particular problems should follow naturally from the metaheuristic's principle;
4. *Efficiency.* Heuristics for particular problems should provide optimal or near-optimal solutions for all or at least most realistic instances. Preferably, they should find optimal solutions for most problems of benchmarks for which such solutions are known, when available;
5. *Effectiveness.* Heuristics for particular problems should take moderate computing time to provide optimal or near-optimal solutions;
6. *Robustness.* The performance of heuristics should be consistent over a variety of instances, i.e. not just fine-tuned to some training set and less good elsewhere;
7. *User-friendliness.* Heuristics should be clearly expressed, easy to understand and, most important, easy to use. This implies they should have as few parameters as possible and ideally none;
8. *Innovation.* Preferably, the metaheuristic's principle and/or the efficiency and effectiveness of the heuristics derived from it should lead to new types of applications.

VNS possesses, to a large extent, all of these properties. This has led to heuristics being among the very best ones for several problems, but more importantly to insight into the solution process and some innovative applications.

Sources of Additional Information

Some Web addresses with sources of information about VNS include

- <http://scholar.google.co.uk/>—“Variable neighborhood search” gets around 36,000 files.
- <http://apps.isiknowledge.com/>—“Variable neighborhood search” in the “topics” window finds 478 VNS papers, together with 4,081 citations and VNS h-index equal to 28 (on 30 November 2010).
- http://www.gerad.ca/en/publications/cahiers_rech.php—if one chooses the option “search for papers” and in the “Abstract” box types “Variable neighborhood search”, 56 papers for downloading will appear at the screen; typing “Variable neighbourhood search” gets an additional eight papers.

Survey papers: [Hansen and Mladenović \(1999, 2001a,c, 2002a,b, 2003\)](#), [Hansen et al. \(2003a, 2008, 2010a,b\)](#) and [Kochetov et al. \(2003\)](#).

References

- Ahuja RK, Orlin JB, Sharma D (2000) Very large-scale neighborhood search. *Int Trans Oper Res* 7:301–317
- Aloise DJ, Aloise D, Rocha CTM, Ribeiro Filho JC, Moura LSS, Ribeiro CC (2006) Scheduling workover rigs for onshore oil production. *Discr Appl Math* 154:695–702
- Baum EB (1986) Toward practical ‘neural’ computation for combinatorial optimization problems. In: Denker J (ed) *Neural networks for computing*. American Institute of Physics, New York
- Braysy O (2001) Local search and variable neighborhood search algorithms for vehicle routing with time windows. *Acta Wasaensia* 87
- Burke EK, Cowling P, Keuthen R (1999) Effective local and guided variable neighborhood search methods for the asymmetric traveling salesman problem. In: Proceedings of the Evo workshops. LNCS 2037. Springer, Berlin, pp. 203–212
- Caporossi G, Hansen P (2000) Variable neighborhood search for extremal graphs 1. The AutoGraphiX system. *Discr Math* 212:29–44
- Caporossi G, Hansen P (2003) Variable neighborhood search for extremal graphs 5. Three ways to automate conjecture finding. *Discr Math* 276:81–94
- Caporossi G, Cvetković D, Gutman I, Hansen P (1999) Variable neighborhood search for extremal graphs 2. Finding graphs with extremal energy. *J Chem Inf Comput Sci* 39:984–996
- Cornuejols G, Fisher M, Nemhauser G (1990) The uncapacitated facility location problem. In: Mirchandani P, Francis R (eds) *Discrete location theory*. Wiley, New York
- Desaulniers G, Desrosiers J, Solomon MM (2001) Accelerating strategies in column generation methods for vehicle routing and crew scheduling problems. In: Essays and surveys in metaheuristics. Kluwer, Dordrecht, pp. 309–324
- Festa P, Pardalos P, Resende M, Ribeiro C (2001) GRASP and VNS for Max-cut. In: Proceedings of the MIC 2001, pp. 371–376
- Fischetti M, Lodi A (2003) Local branching. *Math Program B* 98:23–47
- Fischetti M, Polo C, Scantamburlo M (2003) A local branching heuristic for mixed-integer programs with 2-level variables. Research report, University of Padova
- Garey MR, Johnson DS (1978) Computers and intractability: a guide to the theory of NP-completeness. Freeman, New York
- Glover F, Laguna M (1997) Tabu search. Kluwer, Boston
- Glover F, Kochenberger G (eds) (2003) *Handbook of metaheuristics*. Kluwer, Dordrecht
- Hansen P (1974) Programmes mathématiques en variables 0–1. Thèse d’Agrégation de l’Enseignement Supérieur, Université Libre de Bruxelles
- Hansen P (1975) Les procédures d’optimization et d’exploration par séparation et évaluation. In: Roy B (ed) *Combinatorial programming*. Reidel, Dordrecht, pp 19–65
- Hansen P, Mladenović N (1997) Variable neighborhood search for the p -median. *Locat Sci* 5:207–226

- Hansen, P, Mladenović N (1999) An introduction to variable neighborhood search. In: Voss S et al (eds) *Metaheuristics, advances and trends in local search paradigms for optimization*. Kluwer, Dordrecht, pp 433–458
- Hansen P, Mladenović N (2001a) Variable neighborhood search: principles and applications. *Eur J Oper Res* 130:449–467
- Hansen P, Mladenović N (2001b) J-Means: A new local search heuristic for minimum sum-of-squares clustering. *Pattern Recognit* 34:405–413
- Hansen P, Mladenović N (2001c) Developments of variable neighborhood search. In: Ribeiro C, Hansen P (eds) *Essays and surveys in metaheuristics*. Kluwer, Dordrecht, pp. 415–440
- Hansen P, Mladenović N (2002a) Variable neighborhood search. In: Pardalos P, Resende M (eds) *Handbook of applied optimization*. Oxford University Press, New York, pp. 221–234
- Hansen P, Mladenović N (2002b) Recherche à voisinage variable. In: Teghem J, Pirlot M (eds) *Optimisation approchée en recherche opérationnelle*. Lavoisier Hermès, Paris, pp. 81–100
- Hansen P, Mladenović N (2003) Variable neighborhood search. In: Glover F, Kochenberger G (eds) *Handbook of metaheuristics*. Kluwer, Dordrecht, pp 145–184
- Hansen P, Jaumard B, Mladenović N, Parreira A (2000) Variable neighborhood search for weighted maximum satisfiability problem. *Les Cahiers du GERAD G-2000-62*
- Hansen P, Mladenović N, Perez-Brito D (2001) Variable neighborhood decomposition search. *J Heuristics* 7:335–350
- Hansen P, Mladenović N, Moreno-Pérez JA (2003a) Búsqueda de Entorno Variable (in Spanish). *Inteligencia Artificial* 19:77–92
- Hansen P, Ngai E, Cheung B, Mladenović N (2003b) Survey and comparison of initialization methods for k -means clustering (in preparation)
- Hansen P, Mladenovic N, Urosevic D (2006) Variable neighborhood search and local branching. *Comput Oper Res* 33:3034–3045
- Hansen P, Brimberg J, Urosevic D, Mladenovic N (2007) Primal-dual variable neighborhood for the simple plant location problem. *INFORMS J Comput* 19:552–564
- Hansen P, Mladenović N, Moreno-Pérez JA (2008) Variable neighborhood search. *Eur J Oper Res* 191:593–595
- Hansen P, Mladenovic N, Brimberg J, Moreno Pérez JA (2010a) Variable neighbourhood search. In: Gendreau M, Potvin J-Y (eds) *Handbook of metaheuristics*, 2nd edn. Kluwer, Dordrecht, pp 61–86
- Hansen P, Mladenović N, Moreno Pérez JA (2010b) Variable neighborhood search: algorithms and applications. *Ann Oper Res* 175:367–407
- Kirkpatrick S, Gellatt CD Jr, Vecchi P (1983) Optimization by simulated annealing. *Science* 220:671–680
- Kochetov Y, Mladenović N, Hansen P (2003) Lokalni poisk s chereduyshimisj okrestnostyami (in Russian). *Diskretnaja Matematika* 10:11–43

- Labbé M, Peeters D, Thisse JF (1995) Location on networks. In: Ball M et al (eds) Network routing. North-Holland, Amsterdam, pp 551–624
- Lazic J, Hanafi S, Mladenovic N, Urosevic D (2010) Variable neighborhood decomposition search for 0–1 mixed integer programs. *Comput Oper Res* 37:1055–1067
- Lopez FG, Batista BM, Moreno Pérez JA, Moreno Vega JM (2002) The parallel variable neighborhood search for the p -median problem. *J Heuristics* 8:375–388
- Mladenović N, Hansen P (1997) Variable neighborhood search. *Comput Oper Res* 24:1097–1100
- Papadimitriou C (1994) Computational complexity. Addison-Wesley, Reading
- Reeves CR (ed) (1993) Modern heuristic techniques for combinatorial problems. Blackwell, Oxford
- Resende MGC, Werneck R (2003) On the implementation of a swap-based local search procedure for the p -median problem. In: Ladner RE (ed) Proceedings of the ALENEX 2003. SIAM, Philadelphia, pp 119–127
- Ribeiro C, Uchoa E, Werneck R (2001) A hybrid GRASP with perturbations for the Steiner problem in graphs. Technical report, Computer Science Department, Catholic University of Rio de Janeiro
- Rodriguez I, Moreno-Vega M, Moreno-Perez J (1999) Heuristics for routing-median problems. SMG report, Université Libre de Bruxelles
- Teitz MB, Bart P (1968) Heuristic methods for estimating the generalized vertex median of a weighted graph. *Oper Res* 16:955–961
- Whittaker R (1983) A fast algorithm for the greedy interchange for large-scale clustering and median location problems. *INFOR* 21:95–108
- Zufferey N, Hertz A, Avanthay C (2003) Variable neighborhood search for graph colouring. *Eur J Oper Res* 151:379–388

Chapter 13

Very Large-Scale Neighborhood Search

Douglas S. Altner, Ravindra K. Ahuja, Özlem Ergun, and James B. Orlin

13.1 Introduction

One of the central issues in developing neighborhood search techniques is defining the neighborhood. As a rule of thumb, larger neighborhoods contain higher quality local optimal solutions compared to smaller neighborhoods. However, larger neighborhoods also typically require more time to search than smaller neighborhoods. A neighborhood search algorithm is not practical if the neighborhoods cannot be searched efficiently. Thus, a rapid search algorithm is needed to make efficient use of large neighborhoods.

This chapter introduces very large-scale neighborhood search (VLSN search), the technique of using a fast algorithm to implicitly search a neighborhood that is “very large” relative to the problem size. Generally speaking, VLSN search algorithms either implicitly search large neighborhoods by solving an auxiliary optimization problem or they partially explore the neighborhoods heuristically. This chapter presents an overview of the broad classes of VLSN search algorithms, which

D.S. Altner

Department of Mathematics, United States Naval Academy, Annapolis, MD, USA

e-mail: daltner@aynrand.org

R.K. Ahuja (✉)

Department of Industrial and Systems Engineering, University of Florida,
Gainesville, FL, USA

e-mail: ahuja@ufl.edu

Ö. Ergun

H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute
of Technology, Atlanta, GA, USA

e-mail: oergun@isye.gatech.edu

J.B. Orlin

Operations Research Center, Massachusetts Institute of Technology, Cambridge, MA, USA

e-mail: jorlin@mit.edu

we partition into variable-depth neighborhood search algorithms, cyclic exchange neighborhood search algorithms and other VLSN search algorithms.

Section 13.2 presents preliminary concepts and definitions. Section 13.3 describes variable-depth neighborhood search algorithms, which differ from standard neighborhood search algorithms in that the neighborhoods contain solutions at varying *depths* from the starting solution. Broadly speaking, given an initial s_0 , a solution s_i in a neighborhood of s_0 is within a depth of k if s_0 can be transformed into s_i using at most k moves. A prominent example of a variable-depth neighborhood search algorithm is the well known Lin–Kernighan search heuristic for the traveling salesman problem (TSP).

Section 13.4 presents an overview of cyclic exchange neighborhood search algorithms and lists several classes of difficult problems where these algorithms have obtained the best known solutions. Cyclic exchange neighborhood algorithms are designed for combinatorial optimization problems that require finding an optimal partition of a set of elements. Broadly speaking, a cyclic exchange neighborhood search algorithm iteratively finds a *cycle* of transfers of elements between the subsets of a partition until no improving cycle is found.

Section 13.5 focuses on VLSN search techniques that are neither variable-depth neighborhood search algorithms nor cyclic exchange neighborhood search algorithms but are still frequently used to heuristically solve difficult problems. Section 13.6 presents several implementation suggestions to improve the practical performance of VLSN search algorithms. Section 13.7 highlights several promising areas for future research. The last section offers concluding remarks.

13.2 Preliminaries

A combinatorial optimization problem (COP) is a mathematical problem of the following form: given a finite set of feasible solutions X and a function $f : X \rightarrow \Re$, find a solution x in X that maximizes $f(x)$. f is called the *objective function* and COPs can also require that the objective function be minimized instead of maximized. A COP is a *partitioning problem* if every feasible solution corresponds to a partition of a finite set of elements into two or more disjoint subsets. For example, consider the minimum multiway cut problem (MMCP), which requires finding a minimum cost partition of the vertices of a graph into k subsets such that no two vertices from a set of k terminals are assigned to the same subset. In this problem, the cost of a partition is the total weight of the edges that are between vertices in two different subsets.

A COP can also be an *ordering problem*, where every feasible solution corresponds to an ordering of a finite set of elements and the objective is to find a minimum cost ordering. A classic example of an ordering COP is the TSP, which requires choosing the order in which to visit n cities in a single trip to minimize the total travel distance.

In this chapter we study algorithms that iteratively modify a solution to construct a better solution. A *move* is a rule for modifying a solution. For example, in the

context of the TSP, a solution is an ordered list of cities such as $c_1, c_2, c_3, c_4, c_5, c_6$. One example of a move for the TSP is an *insertion move*, where a single city is removed from the ordered list of cities and is inserted elsewhere in the ordering. For example, a possible insertion move is to insert city c_3 before city c_6 , which results in the new ordering $c_1, c_2, c_4, c_5, c_3, c_6$. Another example of a move is a *swap move*, where the places of two cities in the ordering are exchanged. For example, a possible swap move is to exchange the places of cities c_2 and c_5 , which results in the ordering $c_1, c_5, c_3, c_4, c_2, c_6$. If a move improves the objective value of a solution, then the move is *improving*.

A *neighborhood* of a solution s is a function that maps s to a set of solutions where each solution in the set can be constructed from s with a single execution of a specific move. Every neighborhood is a function of a solution and a type of move. For example, given a solution s to the TSP, there is the swap move neighborhood of s , which is different from the insertion move neighborhood of s , which in turn is different from the swap move neighborhood of a solution x that is different from s . Throughout this chapter, we often say that a solution s_{i+1} is in a neighborhood of a solution s_i to informally describe that solution s_{i+1} is in the range of a neighborhood of s_i .

Any solution in a neighborhood of solution s is a *neighbor* of s . For example, the TSP solution $s_0 = (c_1, c_2, c_3, c_4, c_5, c_6)$ has the solution $s_1 = (c_1, c_5, c_2, c_3, c_4, c_6)$ as a neighbor with respect to the insertion move neighborhood of s_0 since s_1 can be obtained from s_0 by removing city c_5 from the sequence and inserting it before city c_2 . A solution is a *local optimal solution* in a neighborhood if its objective value is better than that of every solution contained in the neighborhood.

A neighborhood search algorithm, also called a local search algorithm, is an algorithm that, given a neighborhood and a starting solution, iteratively moves to a neighboring solution until a stopping criterion is satisfied. Typically, neighborhood search algorithms iteratively move to a neighboring solution with a better objective value until a local optimal solution is discovered. Algorithm 1 displays pseudocode for a generic neighborhood search algorithm.

Algorithm 1 Generic neighborhood search

Begin with initial solution s_0 .

repeat

 Find s_i , the best solution in the neighborhood of s_{i-1} .

until The objective value of s_i is no better than that of s_{i-1} .

return s_{i-1} .

There are more sophisticated neighborhood search algorithms, such as tabu search (see Chap. 9), that consider non-improving moves in an effort to explore a greater portion of the solution space. These algorithms typically terminate after they fail to discover a solution that is better than the best-known solution after a user-specified number of iterations.

A neighborhood is *explicitly searched* when every single solution in a neighborhood is evaluated during an iteration of the neighborhood search algorithm. Likewise, a neighborhood is *implicitly searched* if a locally optimal solution is obtained without explicitly evaluating every solution in the neighborhood. For example, finding an optimal solution to a binary integer program by evaluating every feasible solution is an example of an explicit search of the feasible region. In contrast, using a branch-and-bound method to find an optimal solution is an example of an implicit search, since some feasible solutions may never be explicitly evaluated if subproblems in the branch-and-bound tree are fathomed (see Chap. 2).

A very large-scale neighborhood is a neighborhood that is “very large” in size with respect to the problem input. Typically, these neighborhoods are exponentially large, but this term is often used informally to describe neighborhoods that are too large to explicitly search in practice. Similarly, a VLSN search algorithm is a neighborhood search algorithm that iteratively searches a very large-scale neighborhood. The phrase “very large-scale neighborhood search” was coined by Ahuja et al. (2002).

13.3 Variable-Depth Neighborhood Search Algorithms

A common technique for designing VLSN search algorithms is to create a neighborhood based on simultaneously executing several moves. Neighborhoods based on executing two or three moves simultaneously can often be explicitly searched in a reasonable amount of time. A neighborhood where more moves are executed simultaneously tends to have better local optimal solutions than neighborhoods based on executing a single move. However, if an unreasonable number of computations are required to find an optimal solution to the larger neighborhood then an explicit search is impractical. This motivates variable-depth neighborhood search, which is the practice of partially exploring neighborhoods based on simultaneously executing a variable number of moves.

In this section, we first present definitions for variable-depth neighborhood search algorithms. We also detail two prominent examples of variable-depth neighborhood search algorithms: Kernighan–Lin search for the maximum cut problem and Lin–Kernighan search for the TSP.

13.3.1 Definitions

A neighboring solution that can be reached by a minimum of k moves from a starting solution, s_0 , is at *depth* of k in the neighborhood of s_0 . Ordinary neighborhoods contain solutions that are all at the same depth from the starting solution. In contrast, the neighborhoods in this section, which are called *variable-depth neighborhoods*, contain solutions of varying depths. For example, a neighborhood of a TSP solution

T_0 that contains the set of all tours that can be obtained with up to five insertion moves on T_0 is a variable-depth neighborhood. In contrast, the neighborhood of T_0 that only contains tours that can be obtained after exactly two insertions moves is not a variable-depth neighborhood since this neighborhood is constructed to contain solutions at exactly a depth of two from T_0 . A *variable-depth neighborhood search algorithm* is an algorithm that heuristically searches a variable-depth neighborhood.

13.3.2 Example: Kernighan–Lin Search for the MAX CUT

We present an example of variable-depth neighborhood search by describing the well-known Kernighan–Lin search algorithm for the maximum cut problem (MAX CUT). MAX CUT is defined as follows:

Given an undirected graph $G = (V, E)$ where each edge $e \in E$ has weight w_e , partition the vertices V into two subsets X and Y to maximize the sum of the weights of the edges in the cut $(X, Y) = \{(i, j) : i \in X \text{ and } j \in Y\}$.

Every variable-depth neighborhood is with respect to a specific move. For our example, we examine the MAX CUT problem with a flip move. Given a cut (X, Y) , a *flip move* is the move that transfers one vertex from the subset X to the subset Y (or from the subset Y to the subset X).

A variable-depth neighborhood search algorithm for MAX CUT is presented in Algorithm 2. Algorithm 2 starts with an initial solution C_0 and iteratively looks for an improving solution in the neighborhood of the current solution. The i th iteration of the algorithm starts with a solution C_{i-1} and either moves to a new solution C_i if C_i has a better objective value than C_{i-1} or terminates with the solution C_{i-1} otherwise.

Within each iteration of this variable-depth search, there are several *rounds*. Before the first round of each iteration, all vertices are *unmarked* and cut C_i starts out equal to C_{i-1} , the best cut from the previous iteration. During the j th round, exactly one unmarked vertex $v_{\pi(j)}$ is flipped and then $v_{\pi(j)}$ becomes *marked* for all subsequent rounds of this iteration. In addition, flipping vertex $v_{\pi(j)}$ creates a new cut $C_{i,j}$. If $C_{i,j}$ has weight greater than that of C_i , then C_i is redefined to equal $C_{i,j}$.

An iteration continues until all vertices have been flipped except for one. The last vertex need not be flipped since doing so would invariably result in the solution used at the beginning of this iteration. In addition, at the end of each iteration, C_i is the cut created during this iteration that has the greatest weight. If C_i does not equal C_{i-1} , then C_i is used as the starting solution for the next iteration. Otherwise, the algorithm terminates with C_i .

Algorithm 2 is adapted from the variable-depth neighborhood search algorithm for the graph partitioning problem originally presented by [Kernighan and Lin \(1970\)](#).

An example of a single iteration of Algorithm 2 is executed on a five-vertex graph in Fig. 13.1. At the beginning of this iteration, the current solution is the cut (X_0, Y_0)

Algorithm 2 Variable-depth search for MAX CUT

Start with initial solution C_0 .

repeat

 All vertices are unmarked.

C_{i-1} is the current solution.

 Set C_i equal to C_{i-1} .

repeat

 Flip the unmarked vertex $v_{\pi(j)}$ that creates the cut with the largest weight. Let $C_{i,j}$ be this cut.

 Mark vertex $v_{\pi(j)}$.

if $C_{i,j}$ has greater weight than C_i **then**

 Set C_i equal to $C_{i,j}$.

end if

until All vertices but one are marked.

until C_i is not of greater weight than C_{i-1} .

return C_{i-1} , the best cut seen during this algorithm.

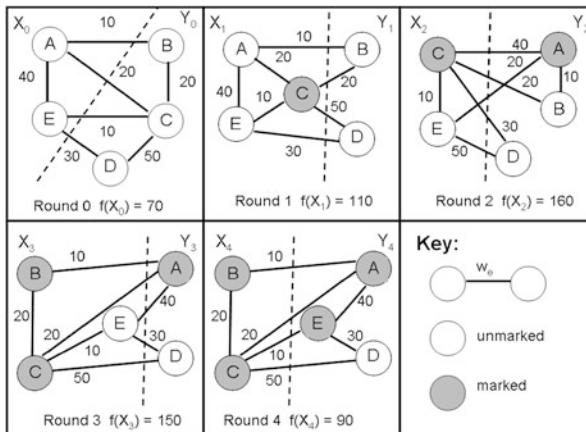


Fig. 13.1 Example of one iteration of Kernighan–Lin search for MAX CUT

where $X_0 = \{A, E\}$ and $Y_0 = \{B, C, D\}$. The weight of (X_0, Y_0) is 70. Vertex C is flipped during the first round, since of the five vertices, flipping C results in the cut with the most weight. Vertex A is flipped during the second round, since flipping A of the four unmarked vertices yields the cut with the most weight. Likewise, vertex B is flipped during the third round and vertex E during the fourth round. Vertex D is not flipped since this would recreate the starting cut. $f(X_i)$ denotes the weight of the cut (X_i, Y_i) .

At the end of this iteration, the cut with the most weight seen is (X_2, Y_2) where $X_2 = \{C, E\}$ and $Y_2 = \{A, B, D\}$. This is the cut that will be used to begin the next iteration. This cut also happens to be the optimal solution to this instance of MAX CUT. However, performing one iteration of this variable-depth neighborhood search algorithm does not guarantee an optimal solution to MAX CUT in general.

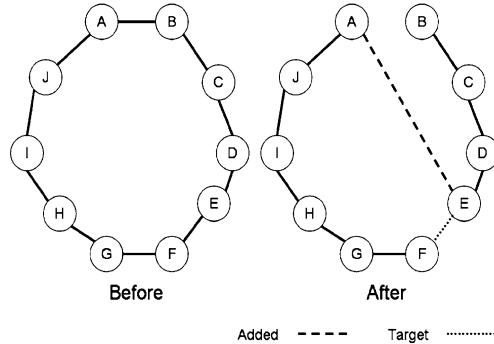


Fig. 13.2 Creating a δ -path from a TSP tour

13.3.3 Example: Lin–Kernighan Search for the TSP

A second example of a variable-depth neighborhood search algorithm is the Lin–Kernighan search heuristic for the TSP. This is a widely used heuristic in both TSP and vehicle routing solvers. Before we present an overview of this heuristic, we first introduce a few definitions.

A *δ -path* is a connected subgraph where one vertex has a degree of three, one vertex has a degree of one and all other vertices have a degree of two. Figure 13.2 illustrates a δ -path being created from a TSP tour (A, B, \dots, J) where edge (A, B) is exchanged for (A, E) . In this example, vertices $\{A, E, F, \dots, J\}$ form the *cycle* of the δ -path and vertices $\{B, C, D, E\}$ form the *stem*. The vertex with degree three in a δ -path is called the *center* of the δ -path. In Fig. 13.2, E is the center. In addition, the *target edge* is the unique edge that is (1) in the cycle, (2) incident to the center and (3) is not the edge that was just added. In Fig. 13.2, the target edge is (E, F) . This edge is called the target edge since it is invariably the next edge that is removed from the δ -path to create either a TSP tour or a better δ -path. Lastly, the *completion edge* is the unique edge that completes the tour if the target edge is deleted from a δ -path. In Fig. 13.2, the completion edge is (B, F) .

Pseudocode for the Lin–Kernighan search is presented in Algorithm 3. The algorithm has one iteration for each vertex v_i in the graph. During the iteration for v_i , the algorithm tries to construct a δ -path from the current TSP tour by deleting an edge incident to v_i and replacing it with a cheaper edge incident to v_i . If such a δ -path can be constructed, then the algorithm iteratively tries to construct a δ -path of even lower cost by replacing the target edge with another edge that creates a new δ -path.

During each iteration, the algorithm temporarily constructs a TSP tour from the current δ -path by deleting the target edge and adding the completion edge. If this tour is better than the best tour seen so far during this algorithm, then it is stored as the best tour seen during the course of the algorithm. In Algorithm 3, this is stored as T^* .

When the cost of the δ -path can no longer be improved in this fashion, the iteration for vertex v_i terminates by converting the δ -path back into a TSP tour.

This is done by deleting the target edge and replacing it with the completion edge. After executing an iteration for each vertex, the algorithm returns T^* .

Algorithm 3 Lin–Kernighan search for TSP

Start with initial TSP tour T_0 .

Let T^* equal T_0 .

for each vertex v_i **do**

 Let $T_i = \{v_1, v_2, \dots, v_n\}$ equal T^* .

if there is a δ -path P_i that is cheaper than T_i and can be constructed from T_i by replacing (v_i, v_{i+1}) with some edge (v_i, v_j) **then**

 Construct P_i .

 Let (v_k, v_{k+1}) be the target edge and let vertex $v_k = v_j$ be the center of P_i .

while A cheaper δ -path can be constructed by exchanging (v_k, v_{k+1}) for another edge (v_{k+1}, v_ℓ) **do**

 Update P_i by swapping (v_k, v_{k+1}) for (v_{k+1}, v_ℓ) .

 Let v_ℓ be the center of the updated P_i and let (v_{k+1}, v_ℓ) be the new target edge.

 Let $T(P_i)$ be the tour created by deleting the target edge and adding the completion edge.

if $T(P_i)$ costs less than T^* **then**

 Set T^* equal to $T(P_i)$.

end if

end while

end if

end for

return T^* , the best tour constructed during this algorithm.

Figures 13.3 and 13.4 illustrate a δ -path being created from another δ -path. In Fig. 13.3, the newly added edge was incident to two vertices in the cycle of the previous δ -path. In Fig. 13.4, the newly added edge was incident to one vertex in the cycle and one vertex in the stem. If this δ -path can no longer be improved by swapping out the target edge, the δ -path in the “After” image of Fig. 13.4 can be converted into a TSP tour by swapping the target edge (D, E) with edge (B, E) .

The Lin–Kernighan search for the TSP was introduced by [Lin and Kernighan \(1973\)](#). Helsgaun’s [\(2000\)](#) implementation of the Lin–Kernighan search is considered the most effective implementation of a neighborhood search algorithm for the TSP and has successfully obtained optimal solutions for all of the instances in [TSPLIB \(2012\)](#) where provably optimal solutions have been obtained. TSPLIB is a widely referenced library of benchmark instances for the TSP and related problems.

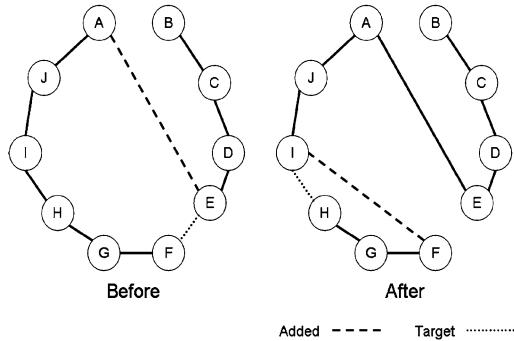


Fig. 13.3 Creating a δ -path from another δ -path

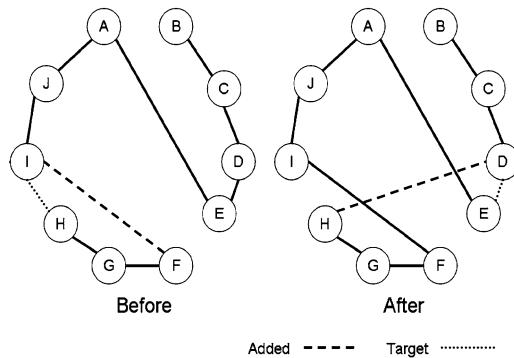


Fig. 13.4 Another example of creating a δ -path from another δ -path

13.3.4 Final Remarks on Variable-Depth Neighborhood Search Algorithms

There are many other classes of problems being solved with variable-depth neighborhood search algorithms, including vehicle routing problems ([Rego and Roucairol 1996](#); [Xu and Kelly 1996](#)), generalized assignment problems ([Yagiura et al. 2004a, 1999](#)) and clustering problems ([Dorndorf and Pesch 1994](#)). [Altner et al. \(2010\)](#) contains an extensive survey of applications of variable-depth neighborhood search algorithms.

13.4 Cyclic Exchange Neighborhood Search Algorithms

A common tactic for designing VLSN search algorithms is to try to model the problem of finding an improving set of simultaneously executed moves as a network flow problem. Cycle exchange neighborhood search algorithms are a prominent class of VLSN search algorithms that are based on finding an improving set of moves by

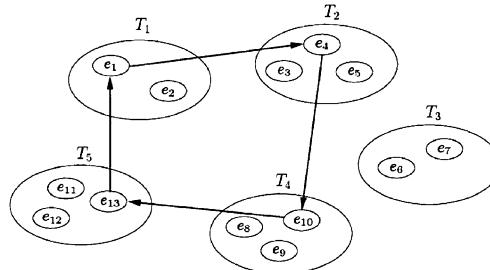


Fig. 13.5 Illustrating a generic cyclic exchange

computing a minimum cost subset-disjoint cycle in an auxiliary network. In this section, we discuss cyclic exchange neighborhoods and present an overview of their applicability to a wide range of settings. Section 13.4.1 describes many of the basic concepts involved in designing a cyclic exchange neighborhood. Section 13.4.2 discusses searching the cyclic exchange neighborhood. Sections 13.4.3 and 13.4.4 present applications of cyclic exchange neighborhood search algorithms to the Vehicle Routing Problem and the Capacitated Minimum Spanning Tree Problem respectively. Section 13.4.5 contains final remarks on cyclic exchange neighborhood search algorithms.

13.4.1 Basic Concepts

The cyclic exchange neighborhood was developed for rapidly obtaining good-quality solutions to computationally difficult minimum cost partitioning problems. Let $S = \{S_1, S_2, \dots, S_k\}$ be a partition of a set of elements E . Roughly speaking, a move in a cyclic exchange neighborhood, which is called a *cyclic exchange*, transfers one element from a subset S_i into a different subset S_j while another element is transferred from S_j into yet another subset S_k . This chain of transfers continues until an element is transferred into S_i , which completes a cycle.

Figure 13.5 illustrates a generic cyclic exchange. Here, element e_1 will be transferred from subset T_1 to subset T_2 . Element e_4 will be transferred from subset T_2 to subset T_4 . Element e_{10} will be transferred from subset T_4 to T_5 . Finally, the cyclic exchange will be completed by transferring element e_{13} from subset T_5 to subset T_1 .

A k -partition of a set of elements E is a collection of subsets $\{S_1, S_2, \dots, S_k\}$ such that no subset is empty, the sets are pairwise disjoint and the union of the sets is E . A *partitioning problem* is a combinatorial optimization problem (COP) where the objective is to find a minimum cost partition of a set of elements.

There are many well-known examples of partitioning COPs. As previously mentioned, the MMCP is one example of a partitioning problem. The vehicle routing problem (VRP) also involves a partitioning COP. The VRP is a generalization of the TSP to k salesmen (i.e. vehicles) where the objective is to visit each of n customers using at most k vehicle tours while minimizing the sum of the distances of each tour.

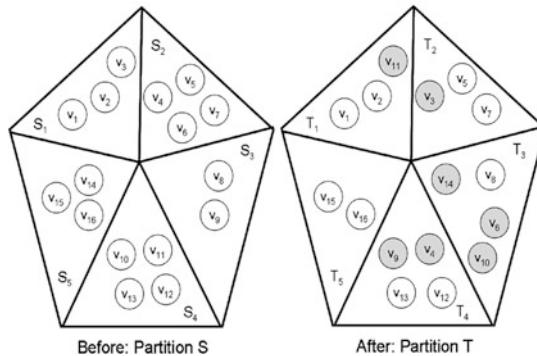


Fig. 13.6 Illustrating a partition transfer

The VRP is a partitioning problem since n customers are partitioned into k disjoint vehicle routes where the cost of each route is the minimum distance required to visit each of the customers covered by the route. Other examples of partitioning problems are provided later in this section.

A *partition transfer* is a way of shifting around the elements within a k -partition. Any cyclic exchange is also a partition transfer so Fig. 13.5 illustrates an example of a partition transfer. However, a partition transfer is a more general concept than a cyclic exchange. Figure 13.6 contains an example of a partition transfer that is not a cyclic exchange. This is because the fifth partition lost an element without receiving another element in return and because the third partition received three elements but only lost one.

A useful way to study partition transfers is to represent them with a graph. Towards this end, the (S, T) -*transfer graph* between two k -partitions S and T (of the same element set E) is defined as follows: for each element being partitioned there is a vertex in the transfer graph. Similarly, if element v_i is transferred to a subset and v_j is removed from the subset that v_i is transferred into, then there is a directed edge from the vertex corresponding to v_i to the vertex corresponding to v_j . Moreover, for each subset in T there is a vertex in the transfer graph. For each element v_i that is removed from subset S_i and transferred into subset T_j without replacing an element in T_j , there is a directed edge from the vertex corresponding to v_i to the vertex corresponding to T_j . Note that the (S, T) -transfer graph is a disjoint union of cycles and paths because each vertex of the graph has at most one incoming edge and at most one outgoing edge. Figure 13.7 illustrates the (S, T) -transfer graph of the partition transfer illustrated in Fig. 13.6.

If a (S, T) -transfer graph of a partition transfer is a single cycle then that partition transfer is a *cyclic exchange*. Likewise, if a (S, T) -transfer graph consists of a single path then T can be obtained from S via a *path exchange*. Figure 13.5 would illustrate a path exchange if the arc from e_{13} to e_1 were deleted.

The *cyclic exchange neighborhood* of a given solution S is the set of all solutions that can be obtained by applying one cyclic exchange to S . Likewise, the *path exchange neighborhood* of S is the set of all solutions that can be obtained by applying a single path exchange to S .

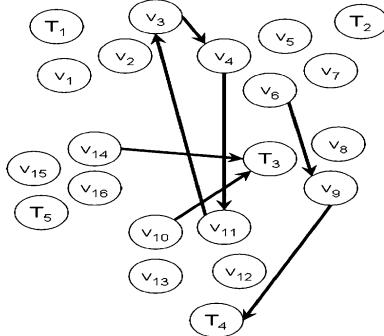


Fig. 13.7 Illustrating a (S, T) -transfer graph

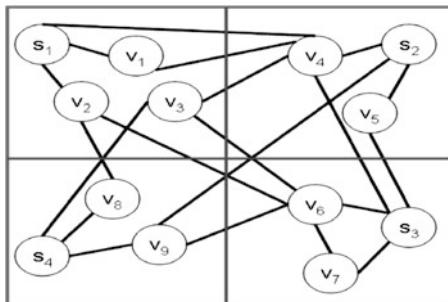


Fig. 13.8 A feasible solution to an instance of the minimum multiway cut problem

Cyclic exchange neighborhoods are rightfully classified as *very large* neighborhoods because the size of the cyclic exchange neighborhood is a function of the number of partitions and the number of elements partitioned. For example, a k -partition of n elements has a cyclic exchange neighborhood of size $O(n^k)$. When k is permitted to grow with n , the size of the neighborhood is typically exponential.

13.4.2 Finding Improving Cyclic Exchanges

Now that we have discussed what cyclic exchanges are, we focus on finding improving exchanges. Improving cyclic exchanges can be obtained by solving a network flow problem on an auxiliary graph called the *improvement graph*. Let E be the set of elements that are being partitioned. Given a k -partition $S = \{S_1, S_2, \dots, S_k\}$, the improvement graph is constructed as follows: for each element in E , add a corresponding vertex to the improvement graph. In addition, an edge (i, j) is created for each pair i, j of elements of E that are in different subsets of S .

Each edge in the improvement graph corresponds to replacing one element in a subset with another. For example, suppose S_j is the subset of S that contains

element j . Then the edge (i, j) represents transferring element i into S_j and removing element j from S_j . Given a particular cyclic exchange neighborhood, each feasible solution to a partitioning COP has a corresponding improvement graph.

For illustrative purposes, we describe the improvement graph for the feasible solution of the MMCP depicted in Fig. 13.8. Vertices s_1, s_2, s_3 and s_4 are the terminals that must be separated by the multicut. The feasible solution shown in Fig. 13.8 is a partition of the non-terminal vertices $\{v_1, v_2, \dots, v_9\}$ into four subsets:

$$\{s_1, v_1, v_2, v_3\}, \{s_2, v_4, v_5\}, \{s_3, v_6, v_7\} \text{ and } \{s_4, v_8, v_9\}.$$

Note that each subset uniquely corresponds to one of the terminals.

The corresponding improvement graph has one vertex for each non-terminal vertex. The improvement graph also has a directed edges (v_i, v_j) and (v_j, v_i) for each pair of non-terminal vertices v_i and v_j that are in different subsets in the feasible solution. Thus, for example, the improvement graph has directed edges such as $(v_1, v_4), (v_1, v_5), (v_1, v_6), (v_4, v_1)$ and (v_4, v_8) but it does not contain directed edges such as $(v_1, v_2), (v_6, v_7)$ or (v_8, v_9) .

To allow for path exchanges, four additional vertices $\{s_1, s_2, s_3, s_4\}$ are added to the improvement graph, one for each subset. In addition, there is a directed edge (v_i, s_j) for each vertex v_i and each subset corresponding to s_j such that vertex v_i is currently not in the subset corresponding to s_j . There are no directed edges originating from any of the vertices in $\{s_1, s_2, s_3, s_4\}$. Thus, for example, the improvement graph has directed edges such as $(v_3, s_2), (v_8, s_1)$ and (v_9, s_2) .

We now discuss computing the cost of a cyclic exchange. Let $c_{(v_i, v_j)}$ be the cost on edge (v_i, v_j) in the improvement graph. Specifically, this is the change in the objective value of the current k -partition if element v_i is moved from its current subset and replaces element v_j in v_j 's current subset S_j . Note that although the objective value is also impacted since v_j must be moved to a new subset, this cost is not included in $c_{(v_i, v_j)}$ because it is incorporated in the cost on the directed edges leaving v_j .

For example, the “Before” image in Fig. 13.9 depicts a cyclic exchange for the MMCP feasible solution shown in Fig. 13.8. In this cyclic exchange, $c_{(v_3, v_4)} = 3 - 4 = -1$ because the s_2 -subset loses four edges: $(v_4, v_1), (v_4, v_3), (v_4, s_1)$ and (v_4, s_3) and it only gains three edges: $(v_3, s_4), (v_3, v_4)$ and (v_3, v_6) . Similarly, $c_{(v_4, v_6)} = 4 - 3 = 1$ because the s_3 -subset gains four edges: $(v_4, v_1), (v_4, v_3), (v_4, s_1)$ and (v_4, s_2) and only loses three edges: $(v_6, v_9), (v_6, v_2)$ and (v_6, v_3) . Likewise, $c_{(v_6, v_3)} = 4 - 3 = 1$ since the s_1 -subset gains four edges: $(v_6, s_3), (v_6, v_7), (v_6, v_3)$ and (v_6, v_9) and loses three edges: $(v_3, s_4), (v_3, v_6)$ and (v_3, v_4) . Thus, the total cost of this cyclic exchange is $-1 + 1 + 1 = 1$, which means the objective value increases by one unit if the cyclic exchange that corresponds to this cycle is executed.

An edge is *between* two disjoint subsets S_i and S_j if the edge has one vertex in S_i and the other vertex in S_j . Note that some edges are now between two subsets when they were within one subset before the exchange. Specifically, these edges are $(v_4, s_2), (v_6, s_3)$ and (v_6, v_7) . Similarly, some edges remain within the same subset, such as $(v_3, v_4), (v_3, s_4)$ and (v_1, v_4) . Likewise, some edges were between two

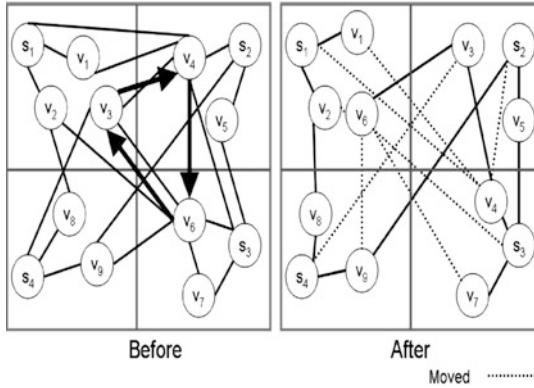


Fig. 13.9 A non-improving cyclic exchange for the MMCP

subsets before the exchange but are within the same subset after the exchange. Specifically, these edges are (v_2, v_6) and (v_4, s_3) . Lastly, the cost of the cyclic exchange equals the total number of edges that are now between two subsets minus the total number of edges that are no longer between two subsets. In this case, this is $3 - 2 = 1$ additional edge that is now between two subsets.

More generally, suppose elements v_i, v_j, v_k and v_ℓ are all currently in distinct subsets S_i, S_j, S_k and S_ℓ respectively. The cost of simultaneously executing the transfers corresponding to edges (v_i, v_j) and (v_k, v_ℓ) equals $c_{(v_i, v_j)} + c_{(v_k, v_\ell)}$ because each of these elements is in its own distinct subset. For example, consider simultaneously swapping vertices v_2 and v_8 as well as swapping vertices v_5 and v_6 in Fig. 13.8. In this example, $c_{(v_2, v_8)} = 2$ and $c_{(v_5, v_6)} = 2$ and the cost of executing both of these moves simultaneously is equal to $c_{(v_2, v_8)} + c_{(v_5, v_6)} = 4$.

Conversely, if v_j and v_ℓ are currently in the same subset $S_{j,\ell}$ then the cost of executing the transfers corresponding to edges (v_i, v_j) and (v_k, v_ℓ) is *not* necessarily equal to $c_{(v_i, v_j)} + c_{(v_k, v_\ell)}$. This is because $c_{(v_i, v_j)}$ is the cost of replacing element v_j in subset $S_{j,\ell}$ with v_i assuming all of the other elements in $S_{j,\ell}$ (including v_ℓ) remain unmoved. ($c_{(v_k, v_\ell)}$ is defined similarly.) Sticking with Fig. 13.8, suppose swapping v_2 with v_8 is performed simultaneously with swapping v_6 with v_9 . Here, $c_{(v_2, v_8)} = 2$ and $c_{(v_6, v_9)} = 3$ so $c_{(v_2, v_8)} + c_{(v_6, v_9)} = 5$. However, executing these two moves simultaneously only costs four edges since five new edges are now between subsets but the edge (v_2, v_6) is no longer between two subsets.

There are sufficient conditions for when the cost of an entire cyclic exchange $(v_1, v_2), (v_2, v_3), \dots, (v_k, v_1)$ equals the sum of the costs on the corresponding edges: $c_{(v_1, v_2)} + c_{(v_2, v_3)} + \dots + c_{(v_k, v_1)}$. Thompson and Orlin (1989) show there is such a cyclic exchange with cost c^* in the original partitioning problem if there is a corresponding *subset-disjoint* cycle in the improvement graph with cost c^* . A cycle $(v_1, v_2), (v_2, v_3), \dots, (v_k, v_1)$ is *subset-disjoint* if and only if each of the vertices v_1, v_2, \dots, v_k are in distinct subsets. The left image in Fig. 13.9 illustrates a subset-disjoint cycle since v_3, v_4 and v_6 are all in separate subsets. However, the cycle $(v_4, v_3), (v_3, v_6), (v_6, v_2), (v_2, v_4)$ (in the same partition) is not subset-disjoint since vertices v_2 and v_3 are currently in the same subset.

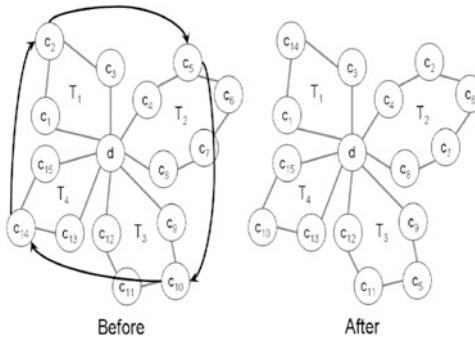


Fig. 13.10 A cyclic exchange in the VRP

A feasible solution to a minimum cost partitioning COP can be improved by executing a cyclic exchange that corresponds to a negative-cost subset-disjoint cycle in the improvement graph. Finding a minimum cost subset-disjoint cycle is NP-hard (Thompson and Orlin 1989) but it is an NP-hard problem that is relatively easy to solve in practice. A good heuristic for constructing negative-cost subset-disjoint cycles is described in Sect. 13.6. This heuristic works very well in practice even though it is not guaranteed to find a negative cost subset-disjoint cycle, even if several exist.

13.4.3 Example: Cyclic Exchange Neighborhood Search for the VRP

This section presents a cyclic exchange neighborhood search algorithm for the VRP. VRP involves a partitioning COP since the customers must be partitioned into subsets that will be serviced by a single vehicle. Thus, each subset of the partition corresponds to a unique vehicle.

Unlike a typical partitioning COP, the VRP requires ordering the elements in each subset of the partition. This is because the cost of a vehicle tour is a function of the order in which the vehicle visits the customers in its route. That being said, an edge (i, j) in the improvement graph of a VRP corresponds to removing customer i from its current vehicle tour T_i , removing customer j from its current vehicle tour T_j and inserting customer i into customer j 's former position in T_j . The improvement graph also contains one vertex that corresponds to each tour itself to allow for path exchanges.

Figure 13.10 illustrates a cyclic exchange for a VRP. The “Before” image illustrates a current solution to a VRP. Tour T_1 indicates the first vehicle visits the following customers in the following order: c_1, c_2, c_3 . Tour T_2 indicates that the second vehicle visits customers c_4, c_5, c_6, c_7 and c_8 in that order. Tour T_3 indicates that the third vehicle visits customers c_9, c_{10}, c_{11} and c_{12} in that order. Lastly, Tour T_4 indicates that the fourth vehicle visits c_{13}, c_{14} and c_{15} in that order. The vertex d

represents the depot where the vehicles begin and end their tours. The cyclic exchange on this solution replaces c_2 in T_1 with c_{14} , replaces c_5 in T_2 with c_2 , replaces c_{10} in T_3 with c_5 and replaces c_{14} in T_4 with c_{10} . The VRP solution that results from this cyclic exchange is depicted in the “After” image depicted in Fig. 13.10.

Cyclic exchange neighborhood search algorithms for variants of the VRP are detailed in Thompson and Orlin (1989), Thompson and Psaraftis (1993) and Agarwal et al. (2003).

13.4.4 Example: Cyclic Exchange Neighborhood Search for the Capacitated Minimum Spanning Tree Problem

This section presents a cyclic exchange neighborhood search algorithm for the capacitated minimum spanning tree problem (CMST). CMST is an NP-hard optimization problem where the objective is to find a minimum cost spanning tree subject to an additional constraint on the maximum number of vertices that can be in each subtree incident to the root vertex. This is a partitioning COP since the vertices of the spanning tree (except for the root, which is common to all subtrees) are the elements that must be partitioned and the subsets of the partition correspond to the vertices in each subtree. In CMST, an edge (i, j) in the improvement graph corresponds to the transfer where vertex i is removed from subtree T_i , vertex j is removed from subtree T_j and vertex i is inserted in subtree T_j . To ensure that the new spanning subtree is of minimum weight, the vertices in $V(T_j) \cup \{i\} \setminus \{j\}$ must be connected via a minimum spanning tree since merely making vertex i adjacent to all of the vertices in T_j that j was adjacent to may result in a suboptimal tree. In this context, $V(T_j)$ is the set of vertices in subtree T_j . It is typically faster to compute the minimum spanning tree on the vertices in $V(T_j) \cup \{i\} \setminus \{j\}$ by modifying the spanning subtree T_j rather than computing it from scratch.

The improvement graph also contains a vertex corresponding to each subtree to allow for path exchanges. Furthermore, the improvement graph contains an additional vertex to allow for the possibility of creating a new subtree with a path exchange. For example, the edge (i, r) in the improvement graph signifies removing vertex i from subtree T_i and creating a new subtree that only contains vertex i .

Figure 13.11 illustrates a cyclic exchange for CMST. In this example, r is the root vertex and the four subtrees T_1, T_2, T_3, T_4 before the cyclic exchange takes place are on the following respective vertex sets:

$$\{v_6, v_7, v_8, v_9\}, \{v_1, v_2, v_3, v_4, v_5\}, \{v_{13}, v_{14}, v_{15}, v_{16}, v_{17}\} \text{ and } \{v_{10}, v_{11}, v_{12}\}.$$

The cyclic exchange being illustrated replaces vertex v_9 in subtree T_1 with vertex v_3 , replaces vertex v_3 in subtree T_2 with vertex v_{17} , replaces vertex v_{17} in subtree T_3 with vertex v_{12} and replaces vertex v_{12} in subtree T_4 with vertex v_9 . The spanning tree that results from this cyclic exchange is depicted in the “After” image of Fig. 13.11.

A cyclic exchange neighborhood search algorithm for CMST is discussed by Ahuja et al. (2001).

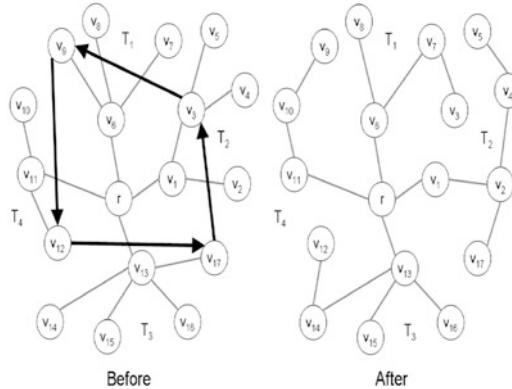


Fig. 13.11 A cyclic exchange in the capacitated minimum spanning tree problem

13.4.5 Final Remarks on Cyclic Exchange Neighborhood Search Algorithms

There are numerous other examples of cyclic exchange neighborhood search algorithms being used to rapidly compute good quality solutions to difficult problems of practical interest. These include problems in airline fleet assignment (Ahuja et al. 2007a, 2004a), exam timetabling (Abdullah et al. 2007), facility location (Ahuja et al. 2004b; Scaparra et al. 2004) inventory routing (Sindhuchao et al. 2005), machine scheduling (Frangioni et al. 2004), multi-period production planning (Ahuja et al. 2007d), trucking fleet assignment (Ambrosino et al. 2009) and weapon-target assignment (Ahuja et al. 2008). Cyclic exchange neighborhood search algorithms have also been used to obtain successful results on several classic problems in combinatorial optimization including the Quadratic Assignment Problem (Ahuja et al. 2007c) and a multiple-choice, multi-dimensional knapsack problem (Cunha and Ahuja 2005).

For an extensive survey of applications of cyclic exchange neighborhood search algorithms, see Altnér et al. (2010).

13.5 Other Very Large-Scale Neighborhood Search Algorithms

In designing VLSN search algorithms, researchers have drawn inspiration from more than 50 years of algorithmic development in mathematical programming and computer science. Here, we describe a diverse collection of approaches for developing VLSN search algorithms that do not fall under variable-depth neighborhood search algorithms or cyclic exchange neighborhood search algorithms. Section 13.5.1 presents VLSNs based on compounding independent moves,

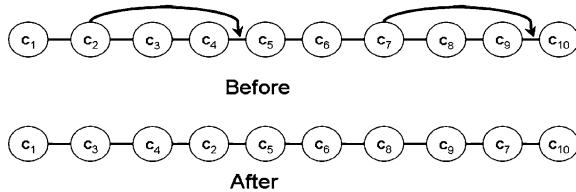


Fig. 13.12 Illustrating two compounded independent insertion moves

Sect. 13.5.2 describes VLSNs that are implicitly searched by solving a mathematical program and Sect. 13.5.3 discusses additional VLSNs that do not fit into any aforementioned category.

13.5.1 Neighborhoods Based on Compounding Independent Moves

To explain the concept of compounding independent moves, we first need a few definitions. An ordering COP is a combinatorial optimization problem that requires arranging a set of elements in an order of minimum cost. The TSP is a classic example of an ordering COP as the problem requires determining the order to visit a set of cities to minimize the total travel distance required for visiting all cities in a single trip. A solution to the TSP is an ordering of the cities. For example, the ordering $c_1, c_2, c_4, c_5, c_6, c_3, c_7, c_8, c_9, c_{10}$ corresponds to the TSP tour where city c_1 is visited first, city c_2 is visited second, ..., city c_9 is visited ninth and city c_{10} is visited last before returning to c_1 . This solution can be represented as a Hamiltonian cycle on a graph where each vertex uniquely corresponds to a city and $(c_1, c_2), (c_2, c_3), \dots, (c_9, c_{10}), (c_{10}, c_1)$ are the corresponding edges in the cycle.

There are several classes of *moves* that can be considered for an ordering COP. One such example is an insertion move. Recall that an insertion move takes one element out of its current position in a solution to an ordering COP and inserts it before another element. For example, if the solution to a TSP is to visit ten cities in the following order:

$$c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}$$

then an insertion move of inserting city c_3 before city c_7 results in the following new solution:

$$c_1, c_2, c_4, c_5, c_6, c_3, c_7, c_8, c_9, c_{10}.$$

Two insertion moves, inserting c_i before c_j and inserting c_k before c_ℓ , are *overlapping* if either $c_i \leq c_k \leq c_j$ or $c_i \leq c_\ell \leq c_j$. Two insertion moves are *independent* if they are non-overlapping. To continue our example, inserting c_2 before c_5 and inserting c_7 before c_{10} are two independent insertion moves while inserting c_2 before c_8 and inserting c_7 before c_{10} are not independent. Two independent insertion moves are *compounded* if they are executed simultaneously. Figure 13.12 illustrates

compounding the insertion moves of inserting c_2 before c_5 and inserting c_7 before c_{10} in the example we have been discussing.

Given a solution to an ordering COP, the set of all solutions that can be reached by either a single move or a single set of compounded independent moves is called the compounded independent move (CIM) neighborhood. Like the cyclic exchange neighborhood in the previous section, the CIM neighborhood can be searched by solving a network flow problem on an auxiliary graph called the improvement graph. In the context of CIM neighborhoods for ordering COPs, an *improvement graph* is a graph where each vertex corresponds to an element that must be ordered and each edge corresponds to either a move or the decision not to move either of the two elements corresponding to the vertices in the edge. Note that CIM neighborhood search algorithms are a special case of variable-depth neighborhood search algorithms and that cyclic exchange neighborhood search algorithms are a special case of CIM neighborhood search algorithms.

In the particular case of the CIM neighborhood based on compounding insertion moves for the TSP, each vertex in the improvement graph corresponds to a city. In addition, if c_1, c_2, \dots, c_n is the current solution then the edge (c_i, c_j) corresponds to inserting city c_{i+1} before city c_j . We call these edges *insertion edges* and there is one such edge for each i in $\{1, 2, \dots, n-3\}$ and for each j in $\{i+3, i+4, \dots, n\}$. In addition, the edge (c_i, c_{i+1}) corresponds to the decision to keep city c_{i+1} immediately after c_i in the solution. We call these *default edges* and there is one such edge for each i such that $1 \leq i < n$. There are no other edges in the improvement graph.

The weight on each insertion edge equals the change in objective value if the corresponding insertion move is executed. For example, the weight on the insertion edge (c_1, c_5) equals $d_{(c_1, c_3)} + d_{(c_4, c_2)} + d_{(c_2, c_5)} - d_{(c_1, c_2)} - d_{(c_2, c_3)} - d_{(c_4, c_5)}$ where $d_{(c_i, c_j)}$ is the distance from city c_i to city c_j . Each default edge has a weight of 0. Given this improvement graph, the problem of finding the best set of compounded independent insertion moves reduces to finding a least-weight path in the improvement graph. This can be done using any algorithm for finding a shortest path in an acyclic directed graph.

Finding cost-improving compounded independent insertion moves for the TSP by solving a shortest path problem on an improvement graph is just one example of a CIM neighborhood search algorithm. Other CIM neighborhood search algorithms can be designed by studying different ordering COPs, compounding different moves or modeling searching the improvement graph as a different problem in combinatorial optimization. The specific example of the CIM neighborhood for insertion moves for the TSP was originally discussed in [Ergun et al. \(2006\)](#). Ergun et al. also discuss compounding swap moves and 2-opt moves for the TSP as well as compounding these moves for the VRP. [Congram et al. \(2002\)](#) compound independent swap moves for a machine scheduling problem. CIM neighborhoods can also be searched by solving a matching problem. [Ahuja et al. \(2002\)](#) discuss how to compound moves by solving a matching problem in general and [Dror and Levy \(1986\)](#) use a matching neighborhood for improving solutions to an inventory routing problem.

<u>Original BIP:</u>	<u>LP Subproblem:</u>	<u>RINS BIP:</u>
$\text{Max } c^T x$	$\text{Max } c^T x$	$\text{Max } c^T x$
s.t. $Ax \leq b$	s.t. $Ax \leq b$	s.t. $Ax \leq b$
$x \in \{0,1\}^{10}$	$x_1 = 0, x_4 = 1$ $x \in \{0,1\}^{10}$	$x_1 = x_7 = x_8 = 0$ $x_3 = x_4 = 1$ $x \in \{0,1\}^{10}$
Incumbent Solution: $x_1 = x_3 = x_5 = x_6 = 1$ $x_2 = x_4 = x_7 = 0$ $x_8 = x_9 = x_{10} = 0$	Fractional Solution: $x_3 = x_4 = 1$ $x_1 = x_5 = x_7 = x_8 = 0$ $x_2 = 1/3, x_6 = 2/3$ $x_9 = x_{10} = 3/4$	

Fig. 13.13 Providing an example of RINS

13.5.2 Neighborhoods Based on Variable Fixing

Some VLSNs can also be searched by solving a mathematical programming subproblem created by fixing a subset of decision variables. Suppose x is a solution to a mathematical program. For a given subset I of indices, we say another feasible solution y is I -adjacent to x if $y_i = x_i$ for $i \in I$, and we say y is in the I -neighborhood of x . The problem of finding the best solution in the I -neighborhood of a solution x to a mathematical program can be formulated by adding $|I|$ constraints to the original mathematical program that fix the variables with indices in I to their current values in x .

These definitions lay the groundwork for VLSN search algorithms that search neighborhoods by solving mathematical programming subproblems. A prominent example is the relaxation-induced neighborhood search (RINS) algorithm for mixed integer programming (MIP). RINS is a local improvement heuristic used during the course of a branch-and-bound algorithm to quickly find better quality integer feasible solutions than the current best known integer feasible solution. The best known integer feasible solution at any stage during a branch-and-bound algorithm is called the *incumbent solution*.

RINS is based on the intuition that decision variables that assume the same value in the current incumbent solution and in an optimal solution to a linear programming subproblem are likely to assume these same values in the best integer feasible solution that can be found in the subtree rooted at this linear programming subproblem. We call these decision variables the *common-value variables*. The basic idea of RINS is that at some nodes of the branch-and-bound tree, a sub-MIP is solved to try to find an integer feasible solution rather than continuing to branch as normal. Specifically, the sub-MIP contains all of the constraints from the original MIP, all of the additional branching constraints from the linear programming subproblem and constraints that fix the common-value variables to the values they take in the incumbent solution. Finding the optimal solution to this sub-MIP is tantamount to implicitly searching the relaxation-induced neighborhood of the incumbent solution.

Figure 13.13 illustrates an example of how RINS constructs an integer program for searching its neighborhood. This example is on a binary integer program (BIP) with a single set of constraints $Ax \leq b$ in addition to the requirement that all variables

are binary. Suppose the incumbent solution is $x_1 = x_3 = x_5 = x_6 = 1, x_2 = x_4 = x_7 = x_8 = x_9 = x_{10} = 0$ and the branch-and-bound algorithm is currently focusing on the LP subproblem created by adding the two branching constraints: $x_1 = 0$ and $x_4 = 1$ to the original BIP. In addition, suppose the optimal solution to the LP subproblem under consideration is $x_3 = x_4 = 1, x_1 = x_5 = x_7 = x_8 = 0, x_2 = \frac{1}{3}, x_6 = \frac{2}{3}$ and $x_9 = x_{10} = \frac{3}{4}$. In this case, the relaxation-induced neighborhood can be implicitly searched by solving the BIP with the original objective function, the original set of constraints, the two branching constraints $x_1 = 0$ and $x_4 = 1$, and the common-value variable constraints $x_7 = x_8 = 0$ and $x_3 = 1$. The three mathematical programs and the two solutions described in this paragraph are listed in Fig. 13.13.

RINS was originally proposed in [Danna et al. \(2005\)](#) and has been included in the CPLEX MIP solver starting with version 9.0. Large neighborhoods that are searched by solving a constraint programming subproblem have also been used with success. Two such examples are given by [Davenport et al. \(2007\)](#) and [Pesant and Gendreau \(1999\)](#).

13.5.3 Other VLSN Search Algorithms

There are other novel ways to design VLSN search algorithms. [Ahuja et al. \(2007b\)](#) designed a VLSN search algorithm for consolidating shipments into *blocks* to minimize the cost of transporting these shipments through a complex railroad network. Ahuja et al.'s VLSN search algorithm is unique because their neighborhood is based on the physical structure of the underlying real-world problem. Specifically, for a given set of routing and blocking decisions, a neighborhood is created and searched by temporarily dismantling all shipment blocks at a specific railroad yard, temporarily re-routing the shipments on the dismantled blocks and then incrementally introducing new blocks at this yard until no more blocks can (or need to) be added.

Another possibility for developing VLSN search algorithms is by exploiting polynomially solvable subcases of NP-hard COPs. For example, [Ahuja et al. \(2002\)](#) discuss how several different polynomially solvable special cases of the TSP can be converted into VLSN search algorithms.

13.6 Tricks of the Trade

In this section, we discuss a few implementation details that enhance the practical performance of the various VLSN search algorithms discussed in this chapter.

1. *Constructing negative-cost subset-disjoint cycles.* For the construction of negative-cost subset-disjoint cycles, researchers use a modified version of the label-correcting algorithm for the shortest path problem. In particular, [Ahuja et al. \(2001\)](#) strongly recommend modifying the dequeue implementation of the label-correcting algorithm. A brief overview of this algorithm is presented here.

For each vertex v_i , a label-correcting algorithm maintains a *distance label* and a *predecessor label*. Collectively, these labels define a directed tree rooted at s , where s is the starting vertex in the shortest path problem. A vertex v_i 's distance label is an upper bound on the length of the shortest v_i-t path in the directed graph, where t is the termination vertex in the shortest path problem. A vertex v_i 's predecessor label indicates the vertex that immediately precedes v_i in the unique $s-v_i$ path defined by the distance and predecessor labels.

Broadly speaking, a label-correcting algorithm iteratively updates the distance labels until a stopping criterion is met. The algorithm terminates with a tree rooted at s and every $s-u$ path in the tree is a shortest $s-u$ path in the graph. To detect subset-disjoint cycles, this algorithm is modified to only update a vertex's distance and predecessor labels if it preserves the condition that all paths in the tree are subset-disjoint. In addition, all negative-cost subset-disjoint cycles encountered during the execution of the algorithm are stored in a list. The cycle with the negative cost of the greatest magnitude is output as the solution to the minimum cost subset-disjoint cycle problem. [Ahuja et al. \(2001\)](#) present this algorithm in great detail.

The algorithm can discover several cost-improving cyclic exchanges but it has no guarantee of discovering any even if multiple such cycles exist. Nevertheless, [Ahuja et al. \(2001\)](#) report that this algorithm works well in practice if it is run several times using different vertices for the starting vertex s . This algorithm can also be modified to allow for detecting both path exchanges and cyclic exchanges.

2. *Updating objective values.* After a local improvement, it is sometimes significantly faster to compute the objective value of the improved solution starting from that of the previous solution rather than computing the objective value of the improved solution from scratch. For example, consider updating the objective value of the TSP after performing an insertion move. Specifically, consider a TSP solution T_0 that visits the ten cities in the order $c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}$ and consider an insertion move of city c_3 before city c_7 , which results in the solution: $c_1, c_2, c_4, c_5, c_6, c_3, c_7, c_8, c_9, c_{10}$. The cost of this new solution can be computed by starting with the objective value of T_0 , subtracting the cost of traveling between the following three pairs of cities: $(c_2, c_3), (c_3, c_4)$ and (c_6, c_7) and adding the cost of traveling between the following three pairs of cities: $(c_2, c_4), (c_6, c_3)$ and (c_3, c_7) . For TSPs with a large number of cities, updating the cost of the newly obtained solution is much faster than computing the cost of the new solution from scratch. Such update heuristics can be applied to a large number of complex COPs.
3. *Updating improvement graphs.* Repeatedly building improvement graphs from scratch can be computationally expensive. It can also be unnecessarily expensive when the improvement graphs are very similar to those from the previous iteration. In practice, it is often much faster to update an improvement graph using the improvement graph from the previous iteration rather than constructing a new improvement graph from scratch.

For example, after executing a cyclic exchange that transfers four vertices in a MMCP with ten terminals, the only the edges in the improvement graph that need to be updated are those that are incident to a vertex in one of the four affected subsets. The edges in the improvement graph that are between vertices in the six other subsets do not need to be updated. In fact, it is both costly and unnecessary to re-generate them from scratch.

4. *Approximating edge weights in the improvement graph.* Recall that for cyclic exchange neighborhoods the weight of an edge in an improvement graph is the corresponding transfer's contribution to the change of the objective value if a cyclic or path exchange involving this transfer is executed. It can sometimes be computationally intensive to exactly compute these edge weights, especially for the entire improvement graph during each iteration. As a remedy, sometimes approximating the edge weights of the improvement graph significantly reduces the algorithm's running time without compromising the quality of the solutions constructed. [Thompson and Psaraftis \(1993\)](#) discuss using such an approximation for VRPs.
5. *Incorporating tabu lists.* As several other metaheuristic search algorithms, VLSN search algorithms can sometimes find better-quality solutions if moves that do not immediately improve the objective value are allowed. Tabu lists reduce the likelihood that such algorithms cycle and increase the likelihood that such algorithms explore a greater portion of the solution space. [Abdullah et al. \(2007\)](#) incorporate tabu lists with a cyclic exchange neighborhood search algorithm for an exam timetabling problem. Similarly, [Ergun et al. \(2006\)](#) use tabu lists with various CIM neighborhoods for different VRPs.
6. *Incorporating composite cyclic exchanges.* Composite cyclic exchanges are cyclic exchanges where a set (or a sequence) of elements is moved from one subset to another as part of a single subset-disjoint cycle. As with any other neighborhood, the advantage of including composite cyclic exchanges is highly dependent on how efficiently the composite cyclic exchange neighborhood can be searched. Composite cyclic exchanges can be included in a cyclic exchange neighborhood by creating vertices in the improvement graph that correspond to an entire set (or sequence) of elements. For example, [Agarwal et al. \(2003\)](#) use a composite cyclic exchange neighborhood for the VRP that allows a sequence of customers to be transferred from one vehicle tour to another. Similarly, [Ahuja et al. \(2003\)](#) use a composite cyclic exchange neighborhood for the CMST problem that allows an entire sub-subtree to be transferred from one subtree to another.

Figure 13.14 illustrates a composite cyclic exchange for a VRP. The “Before” image illustrates a current solution to a VRP. Tour T_1 indicates that the first vehicle visits customers in the following order: c_1, c_2, c_3 . Tour T_2 indicates that the second vehicle visits customers c_4, c_5, c_6, c_7 and c_8 in that order. T_3 indicates that the third vehicle visits customers c_9, c_{10}, c_{11} and c_{12} in that order. Lastly, T_4 indicates that the fourth vehicle visits c_{13}, c_{14} and c_{15} in that order. The vertex d represents the depot where the vehicles start and end their tours. The cyclic exchange on this solution replaces c_2 in T_1 with c_{14} , replaces the customer

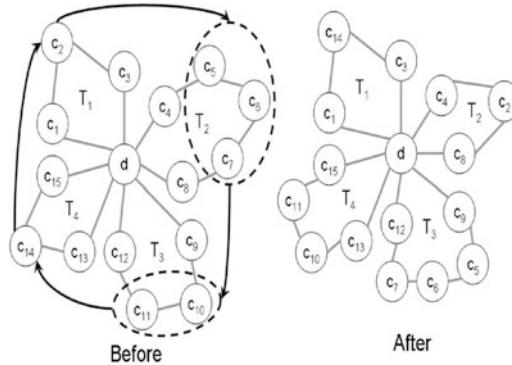


Fig. 13.14 Illustrating a composite cyclic exchange for the VRP

sequence c_5, c_6, c_7 in T_2 with c_2 , replaces the customer sequence c_{10}, c_{11} in T_3 with the sequence c_5, c_6, c_7 and replaces c_{14} in T_4 with the sequence c_{10}, c_{11} . The VRP solution that results from this cyclic exchange is the “After” image depicted in Fig. 13.14.

7. *Using VLSN search selectively.* Sometimes when a VLSN search algorithm is used as a subroutine as part of a broader algorithm it is not practical to employ it during every possible opportunity because this may substantially increase the overall running time of the broader algorithm without offering a worthwhile increase in the quality of the solution. For example, consider the RINS heuristic used during a branch-and-bound algorithm for integer programming. RINS can be used during each subproblem encountered during branch-and-bound. However, to avoid having the RINS heuristic slow down the MIP solver, the heuristic is typically used with an upper bound on the amount of time it is allowed to run. In addition, the heuristic is not always used during every subproblem. For example, the heuristic might not be effective for subproblems whose optimal solution has relatively very few common-value variables with the incumbent solution.
8. *Relaxing the subset-disjoint condition.* Many cyclic exchange neighborhood search algorithms only consider cyclic exchanges that correspond to subset-disjoint cycles in the improvement graph because the costs of the non-subset-disjoint cycles do not necessarily correspond to the change in the objective value that would transpire if the corresponding cyclic exchange is executed. However, there are some instances where the cost of any cycle in the improvement graph is still a reasonably good approximation of the change in the objective value even if the cycle is not subset-disjoint. In particular, Yagiura et al. (2004b) report advantageously relaxing the subset-disjoint condition during their cyclic exchange neighborhood search algorithm for a multi-resource generalized assignment problem.

13.7 Promising Areas for Future Research

There are a few promising directions for future research in VLSN search algorithms:

1. *Grammar-based VLSN search.* [Bompadre and Orlin \(2005\)](#) use context-free grammars to generate neighborhoods for ordering COPs. Specifically, the authors show how a polynomial number of rules of the grammar can be used to generate exponentially large neighborhoods and they develop a dynamic programming solver to search these neighborhoods. This framework unifies a variety of previous results on exponentially large neighborhoods for the TSP and generalizes them to other sequencing problems including the linear ordering problem. With regards to future research, it is worth exploring whether grammars can be used to generate efficiently searchable exponentially large neighborhoods for other COPs.
2. *Group-theoretic VLSN search.* Group theory is a branch of abstract algebra that offers insights for characterizing what mathematical structures are similar to one another. [Colletti and Barnes \(2005\)](#) show how group theory provides insight of the solution space of exchange neighborhoods for the TSP as well as a new framework for viewing neighborhood search algorithms. Colletti and Barnes attribute this group-theoretic framework as having equipped researchers to produce powerful neighborhood search algorithms for multiple complex logistics problems for the US Air Force ([Barnes et al. 2004](#); [Crino et al. 2004](#)). A possible area of future research is to extend the ideas discussed in Colletti and Barnes to apply group theory in a similar fashion to analyze the solution space of VLSNs and to design better VLSN search algorithms.
3. *Parallel VLSN search.* Parallel computing allows researchers to coordinate several processors towards completing a single computational task. The development of parallel computing has allowed researchers to run computational routines to completion using a series of processors that are prohibitively long on a single processor. Recently, operations researchers have been incorporating parallel computing techniques into many different optimization algorithms to solve large problems faster and to solve problems that were previously too large to be practical. Designing and implementing neighborhood search algorithms in a parallel computing framework is a relatively new but growing field. [Alba \(2005\)](#) contains several chapters on the design, implementation and applications of various metaheuristics, such as genetic algorithms, GRASP algorithms and simulated annealing algorithms in a parallel computing environment. A possible area of future research is to design and implement VLSN search algorithms that capitalize on available parallel computing architecture.
4. *Variable neighborhood VLSN search.* The variable neighborhood search (VNS) technique of [Mladenović and Hansen \(1997\)](#) refers to the systematics utilization of several different neighborhoods in a single neighborhood search algorithm. [Mitrović-Minić and Punnen \(2009\)](#) incorporate ideas from VNS into a VLSN search algorithm for multi-resource generalized assignment problems.

Mitrović-Minić and Punnen also discuss how their very large-scale variable neighborhood search technique can be extended to other difficult COPs, which offers direction for future research.

13.8 Conclusions

VLSN search algorithms are instrumental in rapidly computing good quality solutions for difficult COPs of practical interest. We believe there are two main components to the success of a VLSN search algorithm compared to standard neighborhood search algorithms. First, VLSNs contain significantly more solutions than a standard neighborhood, which means they are more likely to contain better-quality solutions than standard neighborhoods. Second, VLSN search algorithms only implicitly search their neighborhoods, as opposed to explicitly considering each element in the neighborhood. This substantially reduces the running time of the algorithms.

VLSN search algorithms are also receiving an increasing amount of attention from the academic community. We suspect this is because VLSN search algorithms are highly practical as they obtain promising results in a wide variety of applications. We also think the popularity of VLSN search algorithms is because they involve elegant algorithmic techniques. This makes the VLSN search literature a nice combination of algorithmic theory and practice.

For more information on the wide variety of instances in which VLSN search algorithms have been used, we recommend the survey by [Altner et al. \(2010\)](#).

References

- Abdullah S, Ahmadi S, Burke EK, Dror M, McCollum B (2007) A tabu-based large neighborhood search methodology for the capacitated examination timetabling problem. *J Oper Res Soc* 58:1494–1502
- Agarwal R, Ahuja RK, Laporte G, Shen ZJ (2003) A composite very large-scale neighborhood search algorithm for the vehicle routing problem. In: Leung, JY-T (ed) *Handbook of scheduling: algorithms, models and performance analysis*. CRC, Boca Raton, pp 49-01–49-23
- Ahuja RK, Orlin JB, Sharma D (2001) Multi-exchange neighborhood structures for the capacitated minimum spanning tree problem. *Math Program* 91:71–97
- Ahuja RK, Ergun Ö, Orlin JB, Punnen AP (2002) A survey of very large-scale neighborhood search techniques. *Discret Appl Math* 123:75–102
- Ahuja RK, Orlin JB, Sharma D (2003) A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Oper Res Lett* 31:185–194

- Ahuja RK, Liu J, Orlin JB, Goodstein J, Mukherjee A (2004a) A neighborhood search algorithm for the combined through fleet assignment model with time windows. *Networks* 44:160–171
- Ahuja RK, Orlin JB, Pallottino S, Scaparra MP, Scutellá MG (2004b) A multi-exchange heuristic for the single source capacitated facility location. *Manag Sci* 50:749–760
- Ahuja RK, Goodstein J, Mukherjee A, Orlin JB, Sharma D (2007a) A very large-scale neighborhood search algorithm for the combined through-fleet assignment model. *INFORMS J Comput* 19:416–428
- Ahuja RK, Jha KC, Liu J (2007b) Solving real-life railroad blocking problems. *Interfaces* 37:404–419
- Ahuja RK, Jha KC, Orlin JB, Sharma D (2007c) A very large-scale neighborhood search algorithm for the quadratic assignment problem. *INFORMS J Comput* 19:646–657
- Ahuja RK, Huang W, Romeijn HE, Morales DR (2007d) A heuristic approach to the multi-period single-sourcing problem with production and inventory capacities and perishability constraints. *INFORMS J Comput* 19:14–26
- Ahuja RK, Kumar A, Jha KC, Orlin JB (2008) Exact and heuristic algorithms for the weapon-target assignment problem. *Oper Res* 55:1136–1146
- Alba E (2005) Parallel metaheuristics: a new class of algorithms. Wiley, New York
- Altner DS, Ahuja RK, Ergun Ö, Orlin JB (2010) Very large-scale neighborhood search. In: Cochran JJ (ed) *Wiley Encyclopedia of operations research and management science*. Wiley, New York
- Ambrosino D, Sciomachen A, Scutellá MG (2009) A heuristic based on multi-exchange techniques for a regional fleet assignment location-routing problem. *Comput Oper Res* 36:442–460
- Barnes JW, Wiley VD, Moore JT, Ryer DM (2004) Solving the aerial fleet refueling problem using group theoretic tabu search. *Math Comput Model* 39:617–648
- Bompadre A, Orlin JB (2005) Using grammars to generate very large-scale neighborhoods for the traveling salesman problem and other sequencing problems. In: Integer programming and combinatorial optimization. LNCS 3509. Springer, Berlin, pp 437–451
- Colletti BW, Barnes JW (2005) Using group theory to construct and characterize metaheuristic search neighborhoods. In: Alidaee B, Rego C (eds) *Metaheuristic optimization via memory and evolution*. Springer, Berlin, pp 303–328
- Congram RK, Potts CN, van de Velde SL (2002) An iterated dynasearch algorithm for the single machine total weighted tardiness scheduling problem. *INFORMS J Comput* 14:52–67
- Crino JR, Moore JT, Barnes JW, Nanry WP (2004) Solving the theater distribution vehicle routing and scheduling problem using group theoretic tabu search. *Math Comput Model* 39:599–616
- Cunha CB, Ahuja RK (2005) Very large-scale neighborhood search for the K-constrained multiple knapsack problem. *J Heuristics* 11:465–481
- Danna E, Rothberg E, Le Pape C (2005) Exploring relaxation induced neighborhoods to improve MIP solutions. *Math Program* 102:71–90

- Davenport A, Kalagnanam J, Reddy C, Siegel S, Hou J (2007) An application of constraint programming to generating detailed operations schedules for steel manufacturing. In: Proceedings of the constraint programming, Providence, pp 64–75
- Dorndorf U, Pesch E (1994) Fast clustering algorithms. ORSA J Comput 6:141–153
- Dror M, Levy L (1986) A vehicle routing improvement algorithm comparison of a greedy and a matching implementation for inventory routing. Comput Oper Res 13:33–45
- Ergun Ö, Orlin JB, Steele-Feldman A (2006) Creating very large-scale neighborhoods out of smaller ones by compounding moves. J Heuristics 12:115–140
- Frangioni A, Necciari E, Scutellá MG (2004) Multi-exchange algorithms for minimum makespan machine scheduling problems. J Comb Optim 8:195–220
- Helsgaun K (2000) An effective implementation of the Lin–Kernighan traveling salesman heuristic. Eur J Oper Res 126:106–130
- Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49:291–307
- Lin S, Kernighan BW (1973) An effective heuristic algorithm for the traveling salesman problem. Oper Res 21:498–516
- Mitrović-Minić S, Punnen AP (2009) Local search intensified: very large-scale variable neighborhood search for the multi-resource generalized assignment problem. Discret Optim 6:370–377
- Mladenović N, Hansen P (1997) Variable neighborhood search. Comput Oper Res 24:1097–1100
- Pesant G, Gendreau M (1999) A constraint programming framework for local search methods. J Heuristics 5:255–279
- Rego C, Roucairol C (1996) A parallel tabu search algorithm using ejection chains for the vehicle routing problem. In: Osman IH, Kelly JP (eds) Metaheuristics: theory and applications. Kluwer, Norwell, pp 661–675
- Scaparra MP, Pallottino S, Scutellá MG (2004) Large-scale neighborhood heuristics for the capacitated vertex p -center problem. Networks 43:241–255
- Sindhuchao S, Romeijn HE, Akcali E, Boondiskulchok R (2005) An integrated inventory-routing system for multi-item joint replenishment with limited vehicle capacity. J Glob Optim 32:93–118
- Thompson PM, Orlin JB (1989) The theory of cyclic transfers. Technical report OR 200-89, MIT
- Thompson PM, Psaraftis HN (1993) Cyclic transfer algorithms for multivehicle routing and scheduling problems. Oper Res 41:935–946
- TSPLIB (2012) <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>
- Xu J, Kelly JP (1996) A network flow-based tabu search heuristic for the vehicle routing problem. Transp Sci 30:379–393
- Yagiura M, Yamaguchi T, Ibaraki T (1999) A variable-depth search algorithm for the generalized assignment problem. In: Voss S et al (eds) Metaheuristics: advances and trends in local search paradigms for optimization. Kluwer, Norwell, pp 459–471

- Yagiura M, Ibaraki T, Glover F (2004a) An ejection chain approach for the generalized assignment problem. *INFORMS J Comput* 16:133–151
- Yagiura M, Iwasaki S, Ibaraki T, Glover F (2004b) A very large-scale neighborhood search algorithm for the multi-resource generalized assignment problem. *Discret Optim* 1:87–98

Chapter 14

Constraint Programming

Eugene C. Freuder and Mark Wallace

14.1 Introduction

Constraint satisfaction problems are ubiquitous. A simple example that we will use throughout the first half of this chapter is the following scheduling problem: Choose employees A or B for each of three tasks, X, Y, Z, subject to the work rules that the same employee cannot carry out both tasks X and Y, the same employee cannot carry out both tasks Y and Z, and only employee B is allowed to carry out task Z. (Many readers will recognize this as a simple coloring problem.)

This is an example of a class of problems known as *constraint satisfaction problems* (CSPs). CSPs consist of a set of *variables* (e.g. tasks), a *domain of values* (e.g. employees) for each variable, and *constraints* (e.g. work rules) among sets of variables. The constraints specify which combinations of value assignments are allowed (e.g. employee A for task X and employee B for task Y); these allowed combinations *satisfy* the constraints. A *solution* is an assignment of values to each variable such that all the constraints are satisfied (Dechter 2003; Tsang 1993).

We stress that the basic CSP paradigm can be extended in many directions: for example, variables can be added dynamically, domains of values can be continuous, constraints can have priorities, and solutions can be *optimal*, not merely satisfactory.

Examples of constraints are:

- The meeting must start at 6:30.
- The separation between the soldermasks and nets should be at least 0.15 mm.
- This model only comes in blue and green.
- This cable will not handle that much traffic.

E.C. Freuder

University College Cork, Cork, Ireland

e-mail: e.freuder@4c.ucc.ie

M. Wallace (✉)

Monash University, Melbourne, VIC, Australia

e-mail: mark.wallace@monash.edu

- These sequences should align optimally.
- John prefers not to work on weekends.
- The demand will probably be for more than 5,000 units in August.

Examples of constraint satisfaction or optimization problems are:

- Schedule these employees to cover all the shifts.
- Optimize the productivity of this manufacturing process.
- Configure this product to meet my needs.
- Find any violations of these design criteria.
- Optimize the use of this satellite camera.
- Align these amino acid sequences.

Many application domains (e.g. design) naturally lend themselves to modeling as CSPs. Many forms of reasoning (e.g. temporal reasoning) can be viewed as constraint reasoning. Many disciplines (e.g. operations research) have been brought to bear on these problems. Many computational architectures (e.g. neural networks) have been utilized for these problems. Constraint programming can solve problems in telecommunications, internet commerce, electronics, bioinformatics, transportation, network management, supply chain management, and many other fields.

Some examples of commercial application of constraint technology are:

- Staff planning: BanqueBuxelles Lambert.
- Vehicle production optimization: Chrysler Corporation.
- Planning medical appointments: FREMAP.
- Task scheduling: Optichrome Computer Systems.
- Resource allocation: SNCF (French Railways).
- From push-to-pull manufacturing: Whirlpool.
- Utility service optimization: Long Island Lighting Company.
- Intelligent cabling of big buildings: France Telecom.
- Financial decision support system: Caisse des Dépôts.
- Load capacity constraint regulation: Eurocontrol.
- Planning of satellites missions: Alcatel Espace.
- Optimization of configuration of telecom equipment: Alcatel CIT.
- Production scheduling of herbicides: Monsanto.
- “Just in Time” transport and logistics in food industry: Sun Valley.
- Supply chain management in petroleum industry: ERG Petroli.

CSPs can be represented as *constraint networks*, where the variables correspond to nodes and the constraints to arcs. The constraint network for our sample problem appears in Fig. 14.1. Constraints involving more than two variables can be modeled with hypergraphs, but most basic CSP concepts can be introduced with binary constraints involving two variables, and that is the route we will begin with in this chapter. We will say that a value for one variable is *consistent with* a value for another if the pair of values satisfies the binary constraint between them. (This constraint could be the trivial constraint that allows all pairs of values; such constraints are not represented by arcs in the constraint network.) Note that specifying a domain of values for a variable can be viewed as providing a unary constraint on that single variable.

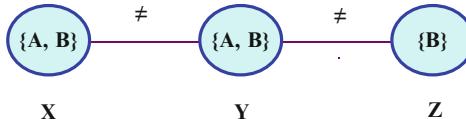


Fig. 14.1 A constraint network representation of a sample constraint satisfaction problem

This chapter focuses on the methods developed in artificial intelligence and the approaches embodied in constraint programming languages. Of course, this brief chapter can only suggest some of the developments in these fields; it is not intended as a survey, only as an introduction. Rather than beginning with formal definitions, algorithms and theorems, we will focus on introducing concepts through examples.

The constraint programming ideal is this: the programming is declarative; we simply state the problem as a CSP and powerful algorithms, provided by a constraint library or language, solve the problem. In practice, this ideal has, of course, been only partially realized, and expert constraint programmers are needed to refine modeling and solving methods for difficult problems.

14.2 Inference

Inference methods make implicit constraint information explicit. Inference can reduce the effort involved in searching for solutions or even synthesize solutions without search. The most common form of inference is known as *arc consistency*. In our sample problem, we can infer that B is not a possible value for Y because there is *no* value for Z that, together with B, satisfies the constraint between Y and Z. This can be viewed as making explicit the fact that the unary constraint on the variable Y does not allow B.

This inference process can *propagate*: after deleting B from the domain of Y, there is no value remaining for Y that together with A for X will satisfy the constraint between X and Y, therefore we can delete A from the domain of X (see Fig. 14.2). If we repeatedly eliminate inconsistent values in this fashion until any value for any variable is consistent with some value for all other variables, we have achieved arc consistency. Many algorithms have been developed to achieve arc consistency efficiently (Bessière and Régin 2001; Mackworth 1977).

Eliminating inconsistent values by achieving arc consistency can greatly reduce the space we must search through for a solution. Arc consistency methods can also be interleaved with search to dynamically reduce the search space, as we shall see in the next section.

Beyond arc consistency lies a broad taxonomy of consistency methods. Many of these can be viewed as some form of (i, j) -consistency (Freuder 1985). A CSP is (i, j) -consistent if, given any consistent set of i values for i variables, we can find j values for any other j variables, such that the $i + j$ values together satisfy

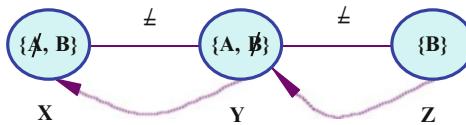


Fig. 14.2 Arc consistency propagation

all the constraints on the $i + j$ variables. Arc consistency is $(1, 1)$ -consistency. $(k - 1, 1)$ -consistency, or k -consistency, for successive values of k constitute an important constraint hierarchy (Freuder 1978).

More advanced forms of consistency processing often prove impractical either because of the processing time involved or because of the space requirements. For example, 3-consistency, otherwise known as *path consistency*, is elegant because it can be shown to ensure that given values for any two variables one can find values that satisfy all the constraints forming any given path between these variables in the constraint network. However, achieving path consistency means making implicit binary constraint information explicit, and storing this information can become too costly for large problems.

For this reason variations on *inverse consistency*, or $(1, j - 1)$ -consistency, which can be achieved simply by domain reductions, have attracted some interest (Debruyne and Bessière 2001). Various forms of *learning* achieve *partial k*-consistency during search (Dechter 1990; Katsirelos and Bacchus 2005). For example, if we modified our sample problem to allow only A for Z, and we tried assigning B to X and A to Y during a search for a solution to this problem, we would run into a *dead end*: no value would be possible for Z. From that we could learn that the constraint between X and Y should be extended to rule out the pair (B, A), achieving partial path consistency.

Interchangeability (Freuder 1991) provides another form of inference, which can also eliminate values from consideration. Suppose that we modify our sample problem to add employees C and D who can carry out task X. Values C and D would be interchangeable for variable X because in any solution using one we can substitute the other. Thus we can eliminate one in our search for solutions—and if we want to, just substitute it back into any solutions we find. Just as with consistency processing there is a local form of interchangeability that can be efficiently computed. In a sense, inconsistency is an extreme form of interchangeability; all inconsistent values are interchangeable in the null set of solutions that utilize them.

14.3 Modeling

Modeling is a critical aspect of constraint satisfaction. Given a user's understanding of a problem, we must determine how to model the problem as a constraint satisfaction problem. Some models may be better suited for efficient solution than others (Régin 2001).

Experienced constraint programmers may add constraints that are *redundant* in the sense that they do not change the set of solutions to the problem, in the hope that adding these constraints may still be cost effective in terms of reducing problem solving effort. Added constraints that do eliminate some, but not all, of the solutions, may also be useful, e.g. to break symmetries in the problem (Walsh 2012).

Specialized constraints can facilitate the process of modeling problems as CSPs, and associated specialized inference methods can again be cost-effective. For example, imagine that we have a problem with four tasks, two employees who can handle each, but three of these tasks must be undertaken simultaneously. This temporal constraint can be modeled by three separate binary inequality constraints between each pair of these tasks; arc consistency processing of these constraints will not eliminate any values from their domains. On the other hand an *alldifferent* constraint, that can apply to more than two variables at a time, not only simplifies the modeling of the problem, but an associated inference method can eliminate all the values from a variable domain, proving the problem unsolvable. Specialized constraints may be identified for specific problem domains, e.g. scheduling problems.

It has even proven useful to maintain multiple complete models for a problem *channeling* the results of constraint processing between the two (Cheng et al. 1999). As has been noted, a variety of approaches have been brought to bear on constraint satisfaction, and it may prove useful to model part of a problem as, for example, an integer programming problem. Insight is emerging into basic modeling issues, e.g. binary versus non-binary models (Bacchus et al. 2002).

In practice, modeling can be an iterative process. Users may discover that their original specification of the problem was incomplete or incorrect or simply impossible. The problems themselves may change over time.

14.4 Search

In order to find solutions we generally need to conduct some form of search. One family of search algorithms attempts to build a solution by *extending* a set of consistent values for a subset of the problem variables, repeatedly adding a consistent value for one more variable, until a complete solution is reached. Another family of algorithms attempts to find a solution by *repairing* an inconsistent set of values for all the variables, repeatedly changing an inconsistent value for one variable, until a complete solution is reached. (Extension and repair techniques can also be combined.)

Often extension methods are systematic and *complete*, they will eventually try all possibilities, and thus find a solution or determine unsolvability, while often repair methods are stochastic and incomplete. The hope is that completeness can be traded off for efficiency.

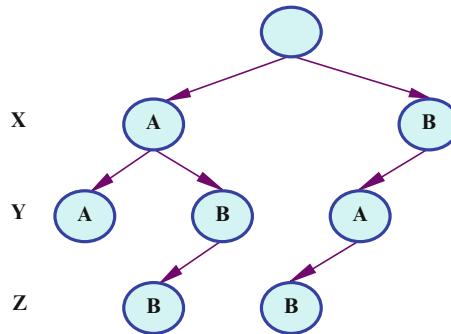


Fig. 14.3 Backtrack search tree for example problem

14.4.1 Extension

The classic extension algorithm is *backtrack search*. Figure 14.3 shows a backtrack search tree representing a trace of a backtrack algorithm solving our sample problem.

A depth-first traversal of this tree corresponds to the order in which the algorithm tried to fit values into a solution. First the algorithm chose to try A for X, then A for Y. At this point it recognized that the choice of A for Y was inconsistent with the choice of A for X: it failed to satisfy the constraint between X and Y. Thus there was no need to try a choice for Z; instead the choice for Y was changed to B. But then B for Z was found to be inconsistent, and no other choice was available, so the algorithm “backed up” to look for another choice for Y. None was available so it backed up to try B for X. This could be extended to A for Y and finally to B for Z, completing the search.

Backtrack search can prune away many potential combinations of values simply by recognizing when an assignment of values to a subset of the variables is already inconsistent and cannot be extended. However, backtrack search is still prone to *thrashing behavior*. A wrong decision early on can require an enormous amount of backing and filling before it is corrected. Imagine, for example, that there were 100 other variables in our example problem, and, after initially choosing A for X and B for Y, the search algorithm tried assigning consistent values to each of those 100 variables before looking at Z. When it proved impossible to find a consistent value for Z (assuming the search was able to get that far successfully) the algorithm would begin trying different combinations of values for all those 100 variables, all in vain.

A variety of modifications to backtrack search address this problem ([Kondrak and van Beek 1997](#)). They all come with their own overhead, but the search effort savings can make the overhead worthwhile.

Heuristics can guide the search order. For example, the *minimal domain size* heuristic suggests that as we attempt to extend a partial solution we consider the variables in order of increasing domain size; the motivation there is that we are more likely to fail with fewer values to choose from, and it is better to encounter

failure higher in the search tree than lower down when it can induce more thrashing behavior. Using this heuristic in our example we would have first chosen B for Z, then proceeded to a solution without having to back up to a prior level in the search tree. While “fail first” makes sense for the order in which to consider the variables, “succeed first” makes sense for the order in which to try the values for the variables.

Various forms of inference can be used prospectively to prune the search space. For example, search choices can be interleaved with arc consistency maintenance. In our example, if we tried to restore arc consistency after choosing A for X, we would eliminate B from the domain of Z, leaving it empty. At this point we would know that A for X was doomed to failure and could immediately move on to B. Even when failure is not immediate, “look ahead” methods that infer implications of search choices can prune the remaining search space. Furthermore, *dynamic* search order heuristics can be informed by this pruning, e.g. the minimal domain size heuristic can be based on the size of the domains after look-ahead pruning. Maintaining arc consistency is an extremely effective and widely used technique ([Sabin and Freuder 1997](#)).

Memory can direct various forms of *intelligent backtracking* ([Dechter and Frost 2002](#)). For example, suppose that in our example for some reason our search heuristics directed us to start the search by choosing B for Y followed by A for X. Of course, B the only choice for Z would then fail. Basic backtrack search would back up *chronologically* to then try B for X. However, if the algorithm *remembers* that failure to find a value for Z was based solely on conflict with the choice for Y, it can *jump back* to try the alternative value A at the Y level in the search tree without unnecessarily trying B for X. The benefits of maintaining arc consistency overlap with those of intelligent backtracking, and the former may make the latter unnecessary.

Search can also be reorganized to try alternatives in a top-down as opposed to bottom-up manner. This responds to the observation that heuristic choices made early in the extension process, when the remaining search space is unconstrained by the implications of many previous choices, may be most prone to failure. For example, *limited discrepancy search* iteratively restarts the search process increasing the number of *discrepancies*, or deviations from heuristic advice, that are allowed, until a solution is found ([Harvey and Ginsberg 1995](#)). (The search effort at the final discrepancy level dominates the upper-bound complexity computation, so the redundant search effort is not as significant as it might seem.)

Extensional methods can be used in an incomplete manner. As a simple example, *random restart*, starting the search over as soon as a dead end is reached, with a stochastic element to the search order, can be surprisingly successful ([Gomes et al. 1997](#)).

14.4.2 Repair

Repair methods start with a complete assignment of values to variables, and work by changing the value assigned to a variable in order to improve the solution. Each such change is called a *move*, and the new assignment is termed a *neighbor* of

the previous assignment. Genetic algorithms, which create a new assignment by combining two previous assignments, rather than by moving to a neighbor of a single assignment, can be viewed as a form of repair.

Repair methods utilize a variety of metaphors, physical (hill climbing, simulated annealing) and biological (neural networks, genetic algorithms). For example, we might start a search on our example problem by choosing value A for each variable. Then, seeking to *hill climb* in the search space to an assignment with fewer inconsistencies, we might choose to change the value of Y to B; and we would be done. Hill climbing is a repair-based algorithm in which each move is required to yield a neighbor with a better cost than before. It cannot, in general, guarantee to produce an optimal solution at the point where the algorithm stops because no neighbor has a better cost than the current assignment.

Repair methods can also use heuristics to guide the search. For example, the *min-conflicts* heuristic suggests finding an inconsistent value and then changing it to the alternative value that minimizes the amount of inconsistency remaining ([Minton et al. 1992](#)).

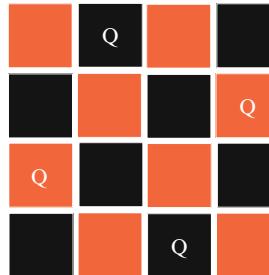
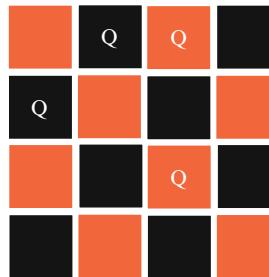
The classic repair process risks getting stuck at a local maximum, where complete consistency has not been achieved, but any single change will only increase inconsistency, or *cycling* through the same set of inconsistent assignments. There are many schemes to cope. A stochastic element can be helpful. When an algorithm has to choose between equally desirable alternatives it may do so randomly. When no good alternative exists it may start over, or jump to a new starting point. Simulated annealing allows moves to neighbors with a worse cost with a given probability. Memory can also be utilized to guide the search and avoid cycling (tabu search).

14.5 Example

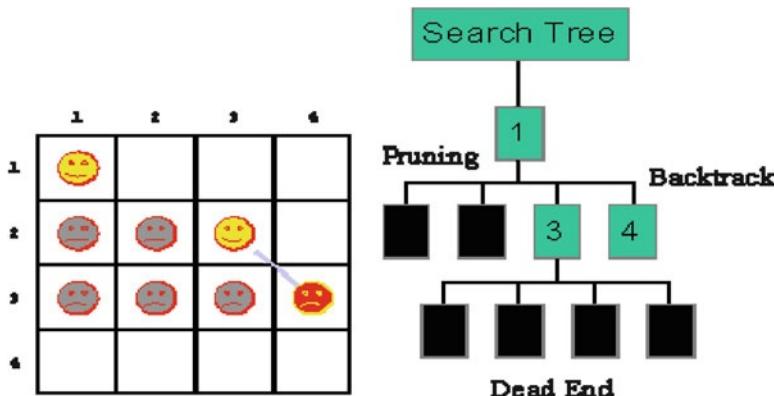
We illustrate simple modeling, search and inference now with another example. The Queens Problem involves placing queens on a chessboard such that they do not attack one another. A simple version only uses a four-by-four corner of the chessboard to place four queens.

Queens in chess attack horizontally, vertically and diagonally. So, for example, the two queens on the dark squares above attack each other diagonally, the two queens on the light squares attack vertically. One solution is:

If we model this problem as a CSP where the variables are the 4 queens and the values for each queen are the 16 squares, we have 65,536 possible combinations to explore, looking for one where the constraints (the queens do not attack each other) are satisfied. If we observe that we can only have 1 queen per row, and model the problem with a variable corresponding to the queen in each row, each variable having 4 possible values corresponding to the squares in the row, we have only 256 possibilities to search through.

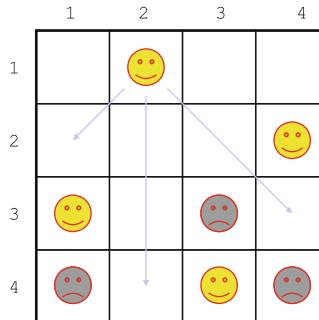


The beginning of the backtrack search tree for this example is shown below. After placing the first row queen in the first column, the first successful spot for the second row queen is in the third column. However, that leaves no successful placement for the third row queen, and we need to backtrack:



In fact, there will be quite a lot of backtracking to do here before we find a solution. However, arc consistency inference can reduce the amount of search we do considerably. Consider what happens if we seek arc consistency after placing a queen in the second column of row 1. This placement directly rules out any square that can be attacked by this queen, of course, and, in fact, arc consistency propagation proceeds

to rule out additional possibilities until we are left with a solution. The queens in column 3 of row 3 and column 4 of row 4 are ruled out because they are attacked by the only possibility left for row 2. After the queen in column 3 of row 3 is eliminated, the queen in column 1 of row 4 is attacked by the only remaining possibility for row 3, so it too can be eliminated:



14.6 Tractability

CSPs are in general NP-hard. Analytical and experimental progress has been made in characterizing tractable and intractable problems. The results have been used to inform algorithmic and heuristic methods.

14.6.1 Theory

Tractable classes of CSPs have been identified based on the structure of the constraint network, e.g. tree structure, and on the types of constraints allowed, e.g. max-closed ([Jeavons and Cooper 1995](#)). Tractability has been associated with sets of constraints defining a specific class of problems, e.g. temporal reasoning problems defined by *simple temporal networks* ([Dechter et al. 1991](#)).

If a constraint network is tree-structured, there will be a *width-one* ordering for the variables in which each variable is directly constrained by at most one variable earlier in the ordering. In our sample problem, which has a trivial tree structure, the ordering X, Y, Z is width-one: Y is constrained by X and Z by Y; the ordering X, Z, Y is not width-one: Y is constrained by both X and Z. If we achieve arc consistency and use a width-one ordering as the order in which we consider variables when trying to extend a partial solution, backtrack search will in fact be *backtrack-free*: for each variable we will be able to find a consistent value without backing up to reconsider a previously instantiated variable ([Freuder 1982](#)).

Max-closure requires that if (ab) and (cd) both satisfy the constraint, then also $(\max(ac), \max(bd))$ will satisfy the constraint. If all the constraints in a problem are max-closed, the problem will be tractable. The *less than* constraint is max-closed, e.g. $4 < 6, 2 < 9$ and $4 < 9$. In fact, simple temporal networks are max-closed.

14.6.2 Experiment

Intuitively it seems natural that many random CSPs would be relatively easy: loosely constrained problems would be easy to solve, highly constrained problems would be easy to prove unsolvable. What is more surprising is the experimental evidence that as we vary the constrainedness of the problems there is a sharp *phase transition* between solvable and unsolvable regions, which corresponds to a sharp spike of *really hard* problems (Cheeseman et al. 1991). (“Phase transition” is a metaphor alluding to physical transitions, such as the one between water and ice.)

14.7 Optimization

Optimization arises in a variety of contexts. If all the constraints in a problem cannot be satisfied, we can seek the *best* partial solution (Freuder and Wallace 1992). If there are many solutions to a problem, we may have some criteria for distinguishing the best one. We can distinguish *hard constraints*, which have to be satisfied, from *soft constraints*, which do not necessarily have to be. Soft constraints can allow us to express probabilities or preferences that make one solution *better* than another.

Again we face issues of modeling, inference and search. What does it mean to be the *best* solution; how do we find the *best* solution? We can assign scores to local elements of our model and combine those to obtain a global score for a proposed solution, then compare these to obtain an optimal solution for the problem.

A simple case is the Max-CSP problem, where we seek a solution that satisfies as many constraints as possible. Here each satisfied constraint scores 1, the score for a proposed solution is the sum of the satisfied constraints, and an optimal solution is one with a maximal score. Many alternatives, such as fuzzy, probabilistic and possibilistic CSPs, have been captured under the general framework of semiring-based or valued CSPs (Bistarelli et al. 1996).

Backtrack search methods can be generalized to branch and bound methods for seeking optimal solutions, where a partial solution is abandoned when it is clear that it cannot be extended to a better solution than one already found. Inference methods can be generalized. Repair methods can often find close to optimal solutions quickly.

14.8 Algorithms

14.8.1 Handling Constraints

Constraint technology is an approach that solves problems by reasoning on the constraints, and when no further inferences can be drawn, making a search step. Thus inference and search are interleaved.

For problems involving hard constraints—ones that must be satisfied in any solution to the problem—reasoning on the constraints is a very powerful technique.

The secret of much of the success of constraint technology comes from its facility to capitalize on the structure of the problem constraints. This enables *global* reasoning to be supported, which can guide a more *intelligent* search procedure than would otherwise be possible.

In this section we introduce some different forms of reasoning and its use in solving problems efficiently.

14.8.2 Domains, and Constraint Propagation

In general, domain constraint propagation algorithms take a set of variables and the original domains as input, and either report inconsistency, or output smaller domains for the variables. Since propagation algorithms can extract more information, each time the input domains become smaller, and since the propagation behavior also makes the domains of the variables smaller, the propagation algorithms can cooperate through these domains. The output from one algorithm is input to another, whose output can in turn be input to the first algorithm. Thus many different propagation algorithms can cooperate, together yielding domain reductions which are much stronger than simply pooling the information from the separate algorithms.

In the following code we constrain two variables, X and Y, to take values in the range 1–10. We constrain X to be greater than Y and (inconsistently!) we also constrain Y to be greater than X:

```
?- X::1..10, Y::1..10
fd:(X>Y), fd:(Y>X)
```

If the propagation algorithm for each constraint makes the bounds consistent, then the first constraint will yield new domains

X::2..10, Y::1..9

and the second constraint will yield new domains

$X::1..9, Y::2..10.$

Pooling the deduced information, we get the intersection of the new domains:

$X::2..9, Y::2..9.$

By contrast, if the propagation algorithms communicate through the variable domains, then they will yield new domains, which are then input to the other algorithm until, at the fifth step, the inconsistency between the two constraints is detected.

The interaction of the different algorithms is predictable, even though the algorithms are completely independent, so long as they have certain natural properties. Specifically,

- The output domains must be a subset of the input domains;
- If with input domains ID the algorithm produces output domains OD, then with any input domains which are a subset of ID, the output domains must be a subset of OD.

These properties guarantee that the information produced by the propagation algorithms, assuming they are executed until no new information can be deduced, is guaranteed to be the same, independent of the order in which the different algorithms (or *propagation steps*) are executed.

14.8.3 Constraints and Search

14.8.3.1 Separating Constraint Handling from Search

The constraint programming paradigm supports clarity, correctness and maintainability by separating the statement of the problem as far as possible from the details of the algorithm used to solve it. The problem is stated in terms of its decision variables, the constraints on those variables, and an expression to be optimized.

As a toy example the employee task problem introduced at the beginning of this chapter can be expressed as follows:

```
?- X::[a,b], Y::[a,b], Z::[b], % Set up variables
X= \=Y, Y= \=Z. % set up constraints.
```

This states that the variable X can take either of the two values, a or b; similarly Y can take a or b; but Z can only take the value b. Additionally, the value taken by X must be different from that taken by Y; and the values taken by Y and Z must also be different.

To map this problem statement into an algorithm, the developer must

- Choose how to handle each constraint
- Specify the search procedure.

For example, assuming `fd` is a solver which the developer chooses to handle the constraints, and labeling is a search routine, the whole program is written as follows:

```
?- X::[a,b], Y::[a,b], Z::[b],
   fd:(X = \= Y), fd:(Y = \= Z),
   labeling([X,Y,Z]).
```

This code sends the constraint $X = \neq Y$ to the finite domain solver called `fd`; and the constraint $Y = \neq Z$ is also sent to `fd`. The list of variables X , Y and Z is then passed to the labeling routine.

Domain propagation algorithms take as input the domains of the variables, and yield smaller domains. For example, given that the domain of Z is $[b]$, the `fd` solver for the constraint $Y = \neq Z$ immediately removes b from the domain of Y , yielding a new domain $[a]$. Every time the domain of one of those variables is reduced by another propagation algorithm, the algorithm *wakes* and reduces the domains further (possibly waking the other algorithm). In the above example, when the domain of Y is reduced to $[a]$, the constraint $X = \neq Y$ wakes, and removes b from the domain of X , reducing the domain of X to $[b]$.

The domain of a variable may be reduced either by propagation, or instead by a search decision. In this case propagation starts as before, and continues until no further information can be derived. Thus search and constraint reasoning are automatically interleaved.

14.8.3.2 Search Heuristics Exploiting Constraint Propagation

Most real applications cannot be solved to optimality because they are simply too large and complex. In such cases it is crucial that the algorithm is directed towards areas of the search space where low-cost feasible solutions are most likely to be found.

Constraint propagation can yield very useful information, which can be used to guide the search for a solution. Not only can propagation exclude impossible choices *a priori*, but it can also yield information about which choices would be optimal in the absence of certain (awkward) constraints.

Because constraint propagation occurs after each search step, the resulting information is used to support *dynamic* heuristics, where the next choice is contingent upon all the information about the problem gathered during the current search.

In short, incomplete extension search techniques can produce high-quality solutions to large complex industrial applications in areas such as transportation and logistics. The advantage is that the solutions respect all the hard constraints and are therefore applicable in practice.

14.8.4 Global Constraints

In this section we introduce a variety of application-specific “global” constraints. These constraints achieve more, and more efficient, propagation behavior than would be possible using combinations of the standard equality and disequality constraints introduced above. We first outline two global constraints, one called “alldifferent” for handling sets of disequality constraints, and one called “schedule” for handling resource usage by a set of tasks. A complete catalog of global constraints is available at www.emn.fr/z-info/sdemasse/gccat.

14.8.4.1 Alldifferent

Consider the disequality constraint “ $= \neq$ ” used in the employee task example at the beginning of this chapter. Perhaps surprisingly, global reasoning on a set of such constraints brings more than local reasoning. Suppose we have a list and want to make local changes until all the elements of the list are distinct. If each element has the same domain (i.e. the same set of possible values), then it suffices to choose any element in conflict (i.e. an element whose value occurs elsewhere in the list), and change it to a new value which doesn’t occur elsewhere. If, however, the domains of different elements of the list are different, then there is no guarantee that local improvement will converge to a list whose elements are all different. However, there is a polynomial time graph-based algorithm (Régin 1994) which guarantees to detect if there is no solution to this problem, and otherwise it reduces the domains of all the elements until they contain only values that participate in a solution. The constraint that all the elements of a list must be distinct is usually called alldifferent in constraint programming.

For example, consider the case

```
?- X::[a,b], Y::[a,b], Z::[a,b,c],
alldifferent([X,Y,Z]).
```

In this case Régin’s algorithm (Régin 1994) for the alldifferent constraint will reduce the domain of Z to just [c]. As we saw above, the behavior of reducing the variables domains in this manner is called *constraint propagation*.

Global constraint propagation algorithms work co-operatively within a constraint programming framework. In the above example, the propagation algorithm for the alldifferent constraint takes as input the list of variables, and their original domains. As a result it outputs new, smaller, domains for those variables.

14.8.4.2 Schedule

Consider a task scheduling problem comprising a set of tasks with release times (i.e. earliest start times) and due dates (i.e. latest end times), each of which requires

a certain amount of resource, running on a set of machines that provide a fixed amount of resource.

The schedule constraint works, in principle, by examining each time period within the time horizon.

- First the algorithm calculates how much resource r_i each task i must necessarily take up within this period.
- If the sum $\sum_i r_i$ exceeds the available resource, then the constraint reports an inconsistency.
- If the sum $\sum_{i \neq j} r_i$ takes up so much resource that task j cannot be scheduled within the period, and the remaining resource within the period is r_T , then task j is constrained only to use an amount r_T of resource within this period.

For non-preemptive scheduling, this constraint may force such a task j to start a certain amount of time before the period begins, or end after it. The information propagated narrows the bounds on the start times of certain tasks.

Whilst this kind of reasoning is expensive to perform, there are many quicker, but theoretically less complete forms of reasoning, such as “edge-finding”, which can be implemented in a time quadratic in the number of tasks.

14.8.4.3 Further Global Constraints

The different global constraints outlined above have proven themselves in practice. Using global constraints such as schedule, constraint programming solves benchmark problems in times competitive with the best complete techniques. There are two main advantages of global constraints in constraint programming, in addition to their efficiency in solving standard benchmark problems:

1. They can be augmented by any number of application-specific “side” constraints. The constraint programming framework allows all kinds of constraints to be thrown in, without requiring any change to the algorithms handling the different constraints.
2. They return high-quality information about the problem which can be used to focus the search. Not only do they work with complete search algorithms, but also they guide incomplete algorithms to return good solutions quickly.

Constraint programming systems can include a range of propagation algorithms supporting global reasoning for constraints appearing in different kinds of applications such as rostering, transportation, network optimization, and even bioinformatics.

14.8.4.4 Analysis

One of the most important requirements of a programming system is support for reusability. Many complex models developed by operations researchers have made very little practical impact, because they are so hard to reuse. The concept of a

global constraint is inherently quite simple. It is a constraint that captures a class of subproblems, with any number of variables. Global constraints have built-in algorithms, which are specialized for treating the problem class. Any new algorithm can be easily captured as a global constraint and reused. Global constraints have had a major impact, and are used widely and often as a tool in solving complex real-world problems. They are, arguably, the most important contribution that constraint programming has brought to Operations Research (OR).

14.8.5 Different Constraint Behaviors

Constraint reasoning may derive other information than domain reductions. Indeed any valid inference step can be made by a propagation algorithm. For example, from the constraints

$$X > Y, Y > Z$$

propagation can derive that

$$X > Z.$$

To achieve co-operation between propagation algorithms, and termination of propagation sequences, constraint programming systems typically require propagation algorithms to behave in certain standard ways. Normally they are required to produce information of a certain kind, for example domain reductions.

An alternative to propagating domain reductions is to propagate new linear constraints. Just as domain propagation ideally yields the smallest domains which include all values that could satisfy the constraint, so linear propagation ideally yields the convex hull of the set of solutions. In this ideal case, linear propagation is stronger than domain propagation, because the convex hull of the set of solutions is contained in the smallest set of variable domains (termed the *box*) that contains them.

14.8.6 Extension and Repair Search

Extension search is conservative in that, at every node of the search tree, all the problem constraints are satisfied. Repair search is optimistic, in the sense that a variable assignment at a search node may be, albeit promising, actually inconsistent with one or more problem constraints.

14.8.6.1 Constraint Reasoning and Extension Search

Constraint reasoning, in the context of extension search, corresponds to logical deduction. The domain reductions, or new linear constraints yielded by propagation, are indeed a consequence of the constraint in the input state.

14.8.6.2 Constraint Reasoning and Repair Search

Constraint reasoning can also be applied in the context of repair search. The standard method for handling constraints in repair search is to check them in each search state. If a constraint is violated then the “cost” of the state is increased by an amount computed by the constraint checker (either simply counting the number of violated constraints, or assessing the degree of violation of each constraint and penalizing it accordingly). More radically, any state which includes a violated constraint can be rejected, and the assignment leading to this state immediately undone. These methods of handling violated constraints are, of course, fully covered in other chapters of this book.

There are, however, other methods of moving forward from a search state in which constraints are violated. These methods were pioneered in the mathematical programming community, for handling integer-linear problems. They are applicable in case the violated constraint is—for whatever reason—too difficult to directly enforce during search. When such a violation occurs, these methods return to a previous search state (in which no constraints were violated) and add new easier constraints which preclude the violation. Like propagation, the method of dealing with a (violated difficult) constraint yields new information, in the form of easy constraints that can be dealt with actively by the search procedure. We distinguish two ways of choosing the easy constraints: constraint generation and separation. These are best illustrated by example.

1. *Constraint generation.* For an example of constraint generation, consider a traveling salesman problem (TSP) which is being solved by integer/linear programming. At each search node an integer/linear problem is solved, which only approximates the actual TSP constraint. It generates “routes” in which every location has a predecessor and a successor, but unfortunately the route is sometimes composed of two or more unconnected cycles. Consider a search node which represents a route with a detached cycle. This violates the TSP constraint in a way that can be fixed by adding a linear constraint enforcing a unit flow out from the set of *cities* in the detached cycle. This is the *generated* constraint, at the given search node. The search is complete at the first node where the TSP constraint is no longer violated. Constraint generation can be used in cases where the awkward constraints can be expressed by a conjunction of easy constraints, although the number of such easy constraints in the conjunction may be too large for them all to be imposed.
2. *Separation.* Separation behavior is required to fix any violated constraint which cannot be expressed as a conjunction of easy constraints (however large). If the awkward constraint can be approximated, arbitrarily closely, by a (conjunction of) disjunction(s) of easy constraints, then separation can be used. Constraint reasoning yields one of the easy constraints—one that is violated by the current search node—and imposes it so that the algorithm which produces the next search node is guaranteed to satisfy this easy constraint. Completeness is maintained by imposing the other easy constraints in the disjunction on other branches of the search tree.

14.8.6.3 Languages and Systems

One drawback of the logical basis of Constraint Programming is that search methods are not naturally expressed in logic, and repair-based search includes many states which violate constraints and are therefore logically inconsistent. Depth-first search with backtracking is inbuilt in constraint logic programming (CLP, see next section), and therefore concise and natural to express and control (e.g. [Refalo 2004](#)). Other forms of tree search, such as best-first, do not fit so well, and repair-based search is not even expressible in most implementations of CLP. For controlling tree search a number of search control languages have been developed, including SALSA ([Laburthe and Caseau 1998](#)), ToOLS ([de Givry and Jeannin 2006](#)) and search combinators ([Schrijvers et al. 2011](#)). For repair search in a constraint programming framework the Comet language ([Van Hentenryck and Michel 2009](#)) has been highly influential. More recently Comet has been extended to support both repair-based search and extension search.

A key feature of Comet is the concept of an “invariant”, which is a constraint that retains information used during search. When implementing GSAT ([Selman et al. 1992](#)), by way of example, an invariant is used to record, for each problem variable in each search state, the change in the number of satisfied propositions that would occur if the variable’s value were to be changed. The invariant is specified as a constraint, but maintained by an efficient incremental algorithm.

14.9 Constraint Languages

14.9.1 Constraint Logic Programming

The earliest constraint programming languages, such as *Ref-Arf* and *Alice*, were specialized to a particular class of algorithms. The first general-purpose constraint programming languages were constraint handling systems embedded in logic programming ([Jaffar and Lassez 1987; Van Hentenryck et al. 1992](#)), called *constraint logic programming* (CLP). Examples are CLP(fd), HAL, SICStus and ECLiPSe. Certainly logic programming is an ideal host programming paradigm for constraints, and CLP systems are widely used in industry and academia.

Logic programming is based on relations. In fact, every procedure in a logic program can be read as a relation. However, the definition of a constraint is exactly the same thing—a *relation*. Consequently, the extension of logic programming to CLP is entirely natural. Logic programming also has backtrack search built-in, and this is easily modified to accommodate constraint propagation. CLP has been enhanced with some high-level control primitives, allowing active constraint behaviors to be expressed with simplicity and flexibility. The direct representation of the application in terms of constraints, together with the high-level control, results in short simple programs. Since it is easy to change the model and, separately, the behavior of a

program, the paradigm supports experimentation with problem solving methods. In the context of a rapid application methodology, it even supports experimentation with the problem (model) itself.

14.9.2 Modeling Languages

On the other hand, operations researchers have introduced a wide range of highly sophisticated specialized algorithms for different classes of problems. These algorithms are only applicable to special classes of problem models expressible in modeling languages such as AMPL ([Fourer et al. 2002](#)) and AIMMS.

For many OR researchers CLP and Localizer are too powerful—they seek a modeling language rather than a computer programming language in which to encode their problems. Traditional mathematical modeling languages used by OR researchers have offered little control over the search and the constraint propagation. OPL ([Van Hentenryck 1999](#)) and its more expressive successors Essence ([Frisch et al. 2007](#)) and Zinc ([Marriott et al. 2008](#)) give more control to the algorithm developer. They represent a step towards a full constraint programming language.

By contrast, a number of application development environments (e.g. Visual CHIP) have appeared recently that allow the developer to define and apply constraints graphically, rather than by writing a program. This represents a step in the other direction!

14.9.3 Constraint Satisfaction and Optimization Systems

As constraint technology has matured the community has recognized that it is not a standalone technology, but a weapon in an armory of mathematical tools for tackling complex problems. Indeed, an emerging role for constraint programming is as a framework for combining techniques such as constraint propagation, integer/linear and quadratic programming, interval reasoning, global optimization and metaheuristics.

Several systems are now available that support such a combination of tools and techniques, including G12 ([Stuckey et al. 2005](#)), Comet ([Van Hentenryck and Michel 2009](#)), IBM ILOG Optimization Studio, Microsoft Solver Foundation and NumberJack, from the Cork Constraint Computation Center. These systems support alternative interfaces to enable their users to exploit the underlying tools in the combination most suited to their particular applications.

A hybridization that has recently proven very successful is the combination of CP with propositional satisfiability solvers ([Nieuwenhuis et al. 2005; Stuckey 2010](#)).

14.10 Applications

14.10.1 Current Areas of Application

Constraint programming is based on logic. Consequently any formal specification of an industrial problem can be directly expressed in a constraint program. The drawbacks of earlier declarative programming paradigms have been

- That the programmer had to encode the problem in a way that was efficient to execute on a computer;
- That the end user of the application could not understand the formal specification.

The first breakthrough of constraint programming has been to separate the logical representation of the problem from the efficient encoding in the underlying constraint solvers. This separation of logic from implementation has opened up a range of applications in the area of control, verification and validation.

The second breakthrough of constraint programming has been in the area of software engineering. The constraint paradigm has proven to accommodate a wide variety of problem-solving techniques, and has enabled them to be combined into hybrid techniques and algorithms, suited to whatever problem is being tackled.

As important as the algorithms to the success of constraint technology, has been the facility to link models and solutions to a graphical user interface that makes sense to the end user. Having developers display the solutions in a form intelligible to the end users, forces the developers to put themselves into the shoes of the users.

Moreover, not only are the final solutions displayed to the user: it is also possible to display intermediate solutions found during search, or even partial solutions. The ability to animate the search in a way that is intelligible to the end user means the users can put themselves into the shoes of the developers. In this way the crucial relationship and understanding between developers and end users is supported and users feel themselves involved in the development of the software that will support them in the future.

As a consequence, constraint technology has been applied very successfully in a range of combinatorial problem-solving applications, extending those traditionally tackled using operations research.

The two main application areas of constraint programming are, therefore,

1. Control, verification, and validation;
2. Combinatorial problem solving.

14.10.2 Applications in Control, Verification and Validation

Engineering relies increasingly on software, not only at the design stage, but also during operation. Consider the humble photocopier. Photocopiers are not as humble as they used to be—each system comprises a multitude of components, such

as feeders, sorters, staplers and so on. The next generation of photocopiers will have orders of magnitude more components than now. The challenge of maintaining compatibility between the different components, and different versions of the components, has become unmanageable.

Xerox has turned to constraint technology to specify the behavior of the different components in terms of constraints. If a set of components are to be combined in a system, constraint technology is applied to determine whether the components will function correctly and coherently. The facility to specify behavior in terms of constraints has enabled engineers at Xerox not only to simulate complex systems in software but also to revise their specifications before constructing anything and achieve compatibility first time.

Control software has traditionally been expressed in terms of finite-state machines. Proofs of safety and reachability are necessary to ensure that the system only moves between safe states (e.g. the lift never moves while the door is open) and that required states are reached (the lift eventually answers every request). Siemens has applied constraint technology to validate control software, using techniques such as Boolean unification to detect any errors. Similar techniques are also used by Siemens to verify integrated circuits.

Constraint technology is also used to prove properties of software. For example, abstract interpretation benefits from constraint technology in achieving the performance necessary to extract precise information about concrete program behavior.

Finally, constraints are being used not only to verify software but to monitor and restrict its behavior at runtime. *Guardian Agents* ensure that complex software, in medical applications for example, never behaves in a way that contravenes the certain safety and correctness requirements.

For applications in control, validation and verification, the role of constraints is to model properties of complex systems in terms of logic, and then to prove theorems about the systems. The main constraint reasoning used in this area is propositional theorem proving. For many applications, even temporal properties are represented in a form such that they can be proved using propositional satisfiability.

Nevertheless, the direct application of abstract interpretation to concurrent constraint programs offers another way to prove properties of complex dynamic systems.

14.10.3 Combinatorial Problem Solving

Commercially, constraint technology has made a huge impact in problem-solving areas such as transportation, logistics, network optimization, scheduling and timetabling, production control, configuration and design, and it is also showing tremendous potential in new application areas such as bioinformatics and virtual-reality systems.

Starting with applications to transportation, constraint technology is now used by airline, bus and railway companies all over the world. Applications include timetabling, fleet scheduling, crew scheduling and rostering, stand, slot and platform allocation.

Constraints have been applied in the logistics area for parcel delivery, food, chilled goods, and even nuclear waste. As in other application areas, the major IT system suppliers (such as SAP and I2) are increasingly adopting constraint technology.

Constraints have been applied for Internet service planning and scheduling, for minimizing traffic in banking networks, and for optimization and control of distribution and maintenance in water and gas pipe networks. Constraints are used for network planning (bandwidth, routing, peering points), optimizing network flow and pumping energy (for gas and water), and assessing user requirements.

Constraint technology appears to have established itself as the technology of choice in the areas of short-term scheduling, timetabling and rostering. The flexibility and scalability of constraints was proven in the European market (for example at Dassault and Monsanto), but is now used worldwide.

It has been used for timetabling activities in schools and universities, for rostering staff at hospitals, call centers, banks and even radio stations. An interesting and successful application is the scheduling of satellite operations.

The chemical industry has an enormous range of complex production processes whose scheduling and control is a major challenge, currently being tackled with constraints. Oil refineries and steel plants also use constraints in controlling their production processes. Indeed, many applications of constraints to production scheduling also include production monitoring and control.

The majority of commercial applications of constraint technology have, to date, used finite-domain propagation. Finite domains are a very natural way to represent the set of machines that can carry out a task, the set of vehicles that can perform a delivery, or the set of rooms/stands/platforms where an activity can be carried out. Making a choice for one task, precludes the use of the same resource for any other task which overlaps it, and propagation captures this easily and efficiently.

Naturally, most applications involve many groups of tasks and resources with possibly complex constraints on their availability (for example, personnel regulations may require that staff have 2 weekends off in 3, that they must have a day off after each sequence of night-shifts, and that they must not work more than 40 h a week). For complex constraints like this a number of special constraints have been introduced which not only enable these constraints to be expressed quite naturally, but also associate highly efficient specialized forms of finite-domain propagation with each constraint.

14.10.4 Other Applications

14.10.4.1 Constraints and Graphics

An early use of constraints was for building graphical user interfaces. Now these interfaces are highly efficient and scalable, allowing a diagram to be specified in terms of constraints so that it still carries the same impact and meaning whatever the size or shape of the display hardware. The importance of this functionality in the context

of the Internet and mobile computing is very clear, and constraint-based graphics are likely to have a major impact in the near future. Constraints are also used in design, involving both spatial constraints and, in the case of real-time systems design, temporal constraints.

14.10.4.2 Constraint Databases

Constraint databases have not yet made a commercial impact, but it is a good bet that future information systems will store constraints as well as data values. The first envisaged application of constraint databases is to geographical information systems. Environmental monitoring will follow, and subsequently design databases supporting both the design and maintenance of complex artifacts such as airplanes.

14.11 Potpourri

There are many more topics that could merit an additional section of their own. Here we briefly sample a few of these. [O'Sullivan \(2012\)](#) discusses some opportunities and challenges for constraint programming.

14.11.1 Dynamic Constraint Problems and Soft Constraints

We may need to handle problems that change over time (for example due to machine breakdown, newly placed priority orders, or late running). *Dynamic CSPs* add or delete constraints to produce a sequence of problems ([Dechter and Dechter 1988](#)). After the problem changes, we might want to find a solution close to the solution to the previous problem, or we may simply be interested in finding a new solution as quickly as possible. We may seek robust solutions that are more likely to remain solutions even if the problem does change.

14.11.2 Explanation

Users may feel more comfortable when an explanation can accompany a solution. Explanation is particularly important when a problem is unsolvable. The user wants to know why, and can use advice on modifying the problem to permit a solution, or preferences can be codified a priori to guide subsequent search ([Amilhastre et al. 2002](#)). A related set of problems confronts the need constraint programmers have to better understand the solution process. Explanation and visualization of this process can assist in debugging constraint programs, computing solutions more quickly, and finding solutions closer to optimal ([Deransart et al. 2000; Junker 2004](#)).

14.11.3 Synthesizing Models and Algorithms

Ideally people with constraints to satisfy or optimize would simply state their problems, in a form congenial to the problem domain, and from this statement a representation suited to efficient processing and an appropriate algorithm to do the processing would be synthesized automatically. In practice, considerable human expertise is often needed to perform this synthesis. The challenge is to automate the modelling and solving process ([O’Sullivan 2010](#)).

14.11.4 Distributed Processing

Distributed constraint processing arises in many contexts. There are parallel algorithms for constraint satisfaction and concurrent constraint programming languages. There are applications where the problem itself is distributed in some manner. There are computing architectures that are naturally distributed, e.g. neural networks. There is synergy between constraint processing and software agents. Agents have issues that are naturally viewed in constraint-based terms, e.g. negotiation. Agents can be used to solve constraint satisfaction problems ([Yokoo et al. 1998](#)).

14.11.5 Uncertainty

Real-world problems may contain elements of uncertainty. Data may be problematic. The future may not be known. For example, decisions about fuel purchases may need to be made based on uncertain demand dependent on future weather patterns. We want to model and compute with constraints in the presence of such uncertainty ([Walsh 2002](#)). Many problems exhibit a form of symmetry ([Cohen et al. 2006](#)). For example, in the Queens problem discussed earlier, the two solutions are symmetric in the sense that one is the “mirror image” of the other. Reducing or “breaking” symmetries can save search. Avoiding symmetries is one consideration in choosing how to model a problem. Given a model with symmetry, one can seek to add constraints that break symmetry ([Walsh 2012](#)). For example, one could add a constraint to the Queens problem requiring the queen in the first row to be placed in one of the first two columns. There are also methods that avoid symmetry dynamically during search by recognizing “symmetric” portions of the search space.

14.12 Tricks of the Trade

The constraints community uses a variety of different tools to solve complex problems. There are a number of constraint programming systems available, which support constraint propagation, search and a variety of other techniques. For pedagogical

purposes we will simply show the solution of a simple problem, solved using one constraint programming system, MiniZinc. This system is free for research use, and can be downloaded from www.g12.csse.unimelb.edu.au/minizinc.

We consider a one-machine scheduling problem. The requirement is to schedule a set of tasks on a machine. Each task has a fixed duration, and each has an earliest start time (the *release date*) and a latest end time (the *due date*). How should we schedule the tasks so as to finish soonest?

In constraint programming a problem is handled in three stages:

1. Initialize the problem variables;
2. Constrain the variables;
3. Search for values for the variables that satisfy the constraints.

14.12.1 Initializing Variables

For a scheduling problem, we introduce a time horizon: all tasks will start and end within this horizon. A variable is declared to represent the start time of each task and this variable can only take a value between 0 (the start of our schedule) and the time horizon. A completion time variable is also declared. This is the time when the last task finishes, and also takes a value between 0 and the time horizon. In this model we have fixed the number of tasks as 5 and the time horizon as 50. These variables are encoded as follows:

```
int: Horizon = 50 ;
int: NTasks = 5 ;

array[1..NTasks] of var 0..Horizon: start ;
var 0..Horizon: all_completed ;
```

In this model we have chosen to represent time as a sequence of time points each represented by an integer between 0 and 50.

14.12.2 Constrain the Variables

The start time of each task is constrained to be after the release date of the task, and the end time before the due date. We write “forall (t in 1..NTasks)” in order to apply the constraints to each of the five tasks.

```
array[1..NTasks] of int: Release_date = [0,10,11,20,25] ;
array[1..NTasks] of int: Duration = [12,10,8,10,5] ;
array[1..NTasks] of int: Due_date = [17,40,30,35,45] ;
```

```

constraint  forall (t in 1..NTasks)
            ( start[t] >= Release_date[t] ) ;

constraint  forall (t in 1..NTasks)
            ( start[t]+Duration[t] <= Due_date[t] ) ;

```

The completion time is constrained to be greater than or equal to the end of each task:

```

constraint  forall (t in 1..NTasks)
            ( start[t] + Duration[t] <= all_completed ) ;

```

(Later, during search, MiniZinc will find the minimum possible completion time, which will ensure that it is actually equal to the end of one of the tasks.)

The interesting constraints are the ones that prevent two tasks running on the machine at the same time. To express this condition, we define a “predicate”, which states that either the second task starts after the first plus its duration or else (written “ $\vee/$ ”) the first task starts after the second plus its duration.

```

predicate not_at_same_time(1..NTasks: t1, 1..NTasks: t2) =
    start[t1] + Duration[t1] <= start[t2]
    \/
    start[t2] + Duration[t2] <= start[t1] ;

```

The constraint enforces that this predicate holds for every pair of tasks:

```

constraint  forall (t1,t2 in 1..NTasks where t1 < t2)
            ( not_at_same_time(t1,t2) ) ;

```

Such a constraint, whose definition is an arbitrary predicate, cannot be expressed in a mathematical modelling language.

14.12.3 Search and Propagation

MiniZinc has a default search where the user need only specify what expression should be minimized, namely:

```

solve
    minimize all_completed ;

```

However, it is normal in constraint programming for the user to specify the order in which variables are assigned a value during search, and an order in which values are tried. A static variable order can be specified for example as follows:

```
solve :: int_search(start, input_order, indomain_min, complete)
minimize all_completed ;
```

The part of the line after the “::” is termed an “annotation”. Annotations do not change the logic of a model, but only add control. This annotation specifies that the start variables should be assigned a value in the order that they appear in the array. The first value to try is the earliest possible time point, and then later and later time points when exploring other alternatives during search. The search should be “complete” meaning every possible alternative should be explored until it is proven that the best solution has been found.

Naturally MiniZinc does not have to try every time point for every task start time because some values are ruled out by constraints (for example if the first task has been assigned a start time of 0, then other tasks will only have possible start times starting from the end of the first task—that is, time point 15).

Moreover, once a solution has been found more time points will be removed from the set of alternatives for each task. For example if a solution has been found with a completion time of 45 then, for the start time of the last task, all time points after 40 are ruled out.

The search control annotations offer many other possibilities. Instead of a static order, the next variable to be assigned might be chosen on the basis of the current search state. A common dynamic variable choice heuristic is to select next the variable with the fewest alternative values remaining, which is called “first_fail”. Also instead of selecting a value the next variable may simply have its set of alternatives reduced by simply ruling out the higher valued ones as a group (this is termed “domain splitting”). For scheduling tasks, domain splitting is an effective search method. Thus we can control the search as follows:

```
solve :: int_search(start, first_fail, indomain_split, complete)
minimize all_completed ;
```

The whole MiniZinc program for solving the one-machine scheduling problem is as follows. To run it load MiniZinc and run it on a file called “your_name.mzn” containing this model.

```
int: Horizon = 50 ;
int: NTasks = 5 ;

array[1..NTasks] of var 0..Horizon: start ;
var 0..Horizon: all_completed ;

array[1..NTasks] of int: Release_date = [0,10,11,20,25] ;
array[1..NTasks] of int: Duration = [12,10,8,10,5] ;
array[1..NTasks] of int: Due_date = [17,40,30,35,45] ;

constraint forall (t in 1..NTasks)
```

```

( start[t] >= Release_date[t] ) ;

constraint forall (t in 1..NTasks)
    ( start[t]+Duration[t] <= Due_date[t] ) ;

constraint forall (t in 1..NTasks)
    ( start[t] + Duration[t] <= all_completed ) ;

predicate not_at_same_time(1..NTasks: t1, 1..NTasks: t2) =
    start[t1] + Duration[t1] <= start[t2]
    \vee start[t2] + Duration[t2] <= start[t1] ;

constraint forall (t1,t2 in 1..NTasks where t1 < t2)
    ( not_at_same_time(t1,t2) ) ;

solve :: int_search(start, first_fail, indomain_split, complete)
minimize all_completed ;

```

14.12.4 Introducing Redundant Constraints

The first way to enhance this algorithm is by adding a global constraint, specialized for scheduling problems (see *Global Constraints* above). The new constraint does not remove any solutions: it is logically redundant. However, its powerful propagation behavior enables parts of the search space, where no solutions lie, to be pruned. Consequently, the number of search steps is reduced—dramatically for larger problems! The algorithm was devised by operations researchers, but it has been encapsulated by constraint programmers as a single constraint.

14.12.5 Adding Search Heuristics

The next enhancement is to choose at each search step, first the task with the earliest due date. Whilst this does tend to yield feasible solutions, it does not necessarily produce good solutions, until the end time constraints become tight.

14.12.6 Using an Incomplete Search Technique

For very large problems, complete search may not be possible. In this case the algorithm may be controlled so as to limit the effort wasted in exploring unpromising

parts of the search space. This can be done simply by limiting the number of times a non-preferred ordering of tasks is imposed during search and backtracking.

The above techniques combine very easily, and the combination is very powerful indeed. As a result, constraint programming is currently the technology of choice for operational scheduling problems where task orderings are significant.

Sources of Additional Information

Sources of information about constraint programming include:

- The International Conferences on Principles and Practice of Constraint Programming, whose proceedings are available in the Springer LNCS series;
- The International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR), whose proceedings are also published in Springer LNCS;
- The *Constraints* journal published by Springer;
- *Handbook of Constraint Programming*, ed. Rossi et al., Elsevier, 2006;
- *Constraint Processing*, Rina Dechter. Morgan Kaufmann, 2003;
- *Programming with Constraints: an Introduction*, Kim Marriott and Peter Stuckey. MIT Press, 1998;
- Online Guide to Constraint Programming, maintained by Roman Barták: <http://kti.ms.mff.cuni.cz/~bartak/constraints/>
- Constraints groups:
 - <http://tech.groups.yahoo.com/group/constraints/>
 - <http://groups.google.ie/group/comp.constraints>
- The Association for Constraint Programming: <http://4c.ucc.ie/a4cp/>
- Constraint Programming Online: <http://4c.ucc.ie/cponline/>

Acknowledgements Some of this material is based upon works supported by the Science Foundation Ireland under Eugene Freuder's Grant No. 00/PI.1/C075; some of his contribution to this chapter was prepared while he was at the University of New Hampshire. Richard Wallace and Dan Sabin provided some assistance. The contents of this chapter overlap with a chapter by the same authors on Constraint Satisfaction in the Handbook of Metaheuristics, edited by Fred W. Glover and Gary A. Kochenberger, and published by Kluwer Academic Press.

References

- Amilhastre J, Fargier H, Marquis P (2002) Consistency restoration and explanations in dynamic CSPs—application to configuration. *Artif Intell* 135:199–234
 Bacchus F, Chen X, van Beek P, Walsh T (2002) Binary vs non-binary constraints. *Artif Intell* 140:1–37

- Bessière C, Régin J (2001) Refining the basic constraint propagation algorithm. In: Proc. 17th IJCAI, Seattle, pp 309–315
- Bistarelli S, Fargier H, Montanari U, Rossi F, Schiex T, Verfaillie G (1996) Semiring-based CSPs and valued CSPs: basic properties. In: Jampel M et al (eds) Over-constrained systems. LNCS 1106. Springer, Berlin, pp 111–150
- Cheeseman P, Kanefsky B, Taylor W (1991) Where the really hard problems are. In: Proc. 12th IJCAI, Sydney. Morgan Kaufmann, San Mateo, pp 331–337
- Cheng B, Choi K, Lee J, Wu J (1999) Increasing constraint propagation by redundant modeling: an experience report. *Constraints* 4:167–192
- Cohen C, Jeavons P, Jefferson C, Petrie K, Smith B (2006) Symmetry definitions for constraint satisfaction problems. *Constraints* 11:115–137
- Debruyne R, Bessière C (2001) Domain filtering consistencies. *J Artif Intell Res* 14:205–230
- Dechter R (1990) Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif Intell* 41:273–312
- Dechter R (2003) Constraint processing. Morgan Kaufmann, San Mateo
- Dechter R, Dechter A (1988) Belief maintenance in dynamic constraint networks. In: Proc. 7th AAAI, Saint Paul, pp 37–42
- Dechter R, Frost D (2002) Backjump-based backtracking for constraint satisfaction problems. *Artif Intell* 136:147–188
- Dechter R, Meiri I, Pearl J (1991) Temporal constraint networks. *Artif Intell* 49:61–95
- de Givry S, Jeannin L (2006) A unified framework for partial and hybrid search methods in constraint programming. *Comput OR* 33:2805–2833
- Deransart P, Hermenegildo M, Maluszynski J (eds) (2000) Analysis and visualization tools for constraint programming. LNCS 1870. Springer, Berlin
- Fourer R, Gay DM, Kernighan BW (2002) AMPL: a modeling language for mathematical programming. Duxbury, Pacific Grove
- Freuder E (1978) Synthesizing constraint expressions. *Commun ACM* 11:958–966
- Freuder E (1982) A sufficient condition for backtrack-free search. *J Assoc Comput Mach* 29:24–32
- Freuder E (1985) A sufficient condition for backtrack-bounded search. *J Assoc Comput Mach* 32:755–761
- Freuder E (1991) Eliminating interchangeable values in constraint satisfaction problems. In: Proc. 9th AAAI, Anaheim, pp 227–233
- Freuder E, Wallace R (1992) Partial constraint satisfaction. *Artif Intell* 58:21–70
- Frisch A, Grum M, Jefferson C, Martinez M, Miguel I (2007) The design of ESSENCE: a constraint language for specifying combinatorial problems. In: Proc. 20th IJCAI, Hyderabad, pp 80–87
- Gomes C, Selman B, Crato N (1997) Heavy-tailed distributions in combinatorial search. In: Principles and practice of constraint programming-CP97. LNCS 1330. Springer, Berlin
- Harvey W, Ginsberg M (1995) Limited discrepancy search. In: Proc. 14th IJCAI 1995, Montreal, pp 607–615

- Jaffar J, Lassez J-L (1987) Constraint logic programming. In: Proceedings of the annual ACM symposium on principles of programming languages, Munich. ACM, New York, pp 111–119
- Jeavons P, Cooper M (1995) Tractable constraints on ordered domains. *Artif Intell* 79:327–339
- Junker U (2004) QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In: Proc. 16th AAAI 2004, San Jose, pp 167–172
- Katsirelos G, Bacchus F (2005) Generalized NoGoods in CSPs. In: Proceedings of the AAAI, Pittsburgh, pp 390–396
- Kondrak G, van Beek P (1997) A theoretical evaluation of selected backtracking algorithms. *Artif Intell* 89:365–387
- Laburthe F, Caseau Y (1998) SALSA: a language for search algorithms. In: Proceedings of the 4th international conference on the principles and practice of constraint programming, Pisa, pp 310–324
- Mackworth A (1977) Consistency in networks of relations. *Artif Intell* 8:99–118
- Marriott K, Nethercote N, Rafeh R, Stuckey PJ, Garcia de la Banda M, Wallace M (2008) The design of the Zinc modelling language. *Constraints* 13:229–267
- Minton S, Johnston MD, Philips AB, Laird P (1992) Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling. *Artif Intell* 58:161–205
- Nieuwenhuis R, Oliveras A, Tinelli C (2005) Abstract DPLL and abstract DPLL modulo theories. In: Logic for programming, artificial intelligence, and reasoning. LNCS 3452. Springer, Berlin, pp 36–50
- O’Sullivan B (2010) Automated modelling and solving in constraint programming. In: Proceedings of the 24th National Conference on Artificial Intelligence, Atlanta, AAAI Palo Alto, pp 1493–1497
- O’Sullivan B (2012) Opportunities and challenges for constraint programming. In: Proceedings of the 26th National Conference on Artificial Intelligence, Atlanta, Toronto, AAAI Palo Alto, pp 2148–2152
- Refalo P (2004) Impact-based search strategies for constraint programming. In: Proceedings of the international conference on constraint programming (CP 2004), Toronto. LNCS 3258. Springer, Berlin, pp 557–571
- Régis J-C (1994) A filtering algorithm for constraints of difference in CSPs. In: Proc. 12th AAAI, Seattle, pp 362–367
- Régis J-C (2001) Minimization of the number of breaks in sports scheduling problems using constraint programming. In: Freuder E, Wallace R (eds) Constraint programming and large scale discrete optimization. DIMACS 57. AMS, Providence, pp 115–130
- Sabin D, Freuder E (1997) Understanding and improving the MAC algorithm. In: Principles and practice of constraint programming—Proc CP 1997, Linz. LNCS 1330. Springer, Berlin, pp 167–181
- Schrijvers T, Tack G, Wuille P, Samulowitz H, Stuckey PJ (2011) Search combinatorics. In: Proceedings of the international conference on constraint programming (CP 2011), Perugia. LNCS 6876. Springer, Berlin, pp 774–788

- Selman B, Levesque H, Mitchell D (1992) A new method for solving hard satisfiability problems. In: Proc. 10th AAAI, San Jose, pp 440–446
- Stuckey P (2010) Lazy clause generation: combining the power of SAT and CP (and MIP?) solving. In: Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. LNCS 6140. Springer, Berlin, pp 5–9
- Stuckey PJ, Garcia de la Banda M, Maher M, Marriott K, Slaney J, Somogyi Z, Wallace M, Walsh T (2005) The G12 project: mapping solver independent models to efficient solutions. Logic programming, 21st international conference. LNCS 3668. Springer, Berlin, pp 9–13
- Tsang E (1993) Foundations of constraint satisfaction. Academic, London
- Van Hentenryck P (1999) The OPL optimization programming language. MIT, Cambridge
- Van Hentenryck P, Michel L (2009) Constraint-based local search. MIT, Cambridge
- Van Hentenryck P, Simonis H, Dincbas M (1992) Constraint satisfaction using constraint logic programming. *Artif Intell* 58:113–159
- Walsh T (2002) Stochastic constraint programming. In: Proceedings of the ECAI-2002, Lyon. IOS, Amsterdam, pp 111–115
- Walsh T (2012) Symmetry breaking constraints: recent results. In: Proc. 26th AAAI, Toronto, pp 2192–2198
- Yokoo M, Durfee E, Ishida T, Kuwabara K (1998) The distributed CSP: formalization and algorithms. *IEEE Trans Knowl Data Eng* 10:673–685

Chapter 15

Multi-objective Optimization

Kalyanmoy Deb

15.1 Introduction

Multi-objective optimization is an integral part of optimization activities and has a tremendous practical importance, since almost all real-world optimization problems are ideally suited to be modeled using multiple conflicting objectives. The classical means of solving such problems were primarily focused on scalarizing multiple objectives into a single objective, whereas the evolutionary means have been to solve a multi-objective optimization problem as it is. In this chapter, we discuss the fundamental principles of multi-objective optimization, the differences between multi-objective optimization and single-objective optimization, and describe a few well-known classical and evolutionary algorithms for multi-objective optimization. Two application case studies reveal the importance of multi-objective optimization in practice. A number of research challenges are then highlighted. The chapter concludes by suggesting a few tricks of the trade and mentioning some key resources to the field of multi-objective optimization.

Many real-world search and optimization problems are naturally posed as non-linear programming problems having multiple conflicting objectives. Due to lack of suitable solution techniques, such problems were artificially converted into a single-objective problem and solved. The difficulty arose because such problems give rise to a set of trade-off optimal solutions (known as *Pareto-optimal* solutions), instead of a single optimum solution. It then becomes important to find not just one

K. Deb (✉)

Koenig Endowed Chair Professor, Department of Electrical and Computer Engineering, Michigan State University, East Lansing, 428 S. Shaw Lane, 2120 EB, MI 48824, USA

Professor, Department of Computer Science, Michigan State University, East Lansing, MI, USA

Department of Mechanical Engineering, Michigan State University, East Lansing, MI, USA
e-mail: kdeb@egr.msu.edu

Pareto-optimal solution, but as many of them as possible. This is because any two such solutions constitute a trade-off between the objectives, and users will be in a better position to make a choice when such trade-off solutions are unveiled.

Classical methods use a different philosophy in solving these problems, mainly because of a lack of a suitable optimization methodology to find multiple optimal solutions efficiently. They usually require repetitive applications of an algorithm to find multiple Pareto-optimal solutions and on some occasions such applications do not even guarantee the finding of any Pareto-optimal solutions. In contrast, the population approach of evolutionary algorithms (EAs) is an efficient way to find multiple Pareto-optimal solutions simultaneously in a single simulation run. This aspect has made research and applications in evolutionary multi-objective optimization (EMO) popular over the past one-and-a-half decades. The interested reader may explore current research issues and other important studies in various texts ([Deb 2001](#); [Coello et al. 2002](#); [Goh and Tan 2009](#); [Bagchi 1999](#)), conference proceedings ([Zitzler et al. 2001a](#); [Fonseca et al. 2003](#); [Coello et al. 2005](#); [Obayashi et al. 2007](#); [Ehrgott et al. 2009](#); [Takahashi et al. 2011](#)) and numerous research papers (archived and maintained in [Coello 2003](#)).

In this tutorial, we discuss the fundamental differences between single- and multi-objective optimization tasks. The conditions for optimality in a multi-objective optimization problem are described and a number of state-of-the-art multi-objective optimization techniques, including one evolutionary method are presented. To demonstrate that the evolutionary multi-objective methods are capable and ready for solving real-world problems, we present a couple of interesting case studies. Finally, a number of important research topics in the area of EMO are discussed.

A multi-objective optimization problem (MOOP) deals with more than one objective function. In most practical decision-making problems, multiple objectives or multiple criteria are evident. Because of a lack of suitable solution methodologies, a MOOP has been mostly cast and solved as a single-objective optimization problem in the past. However, there exist a number of fundamental differences between the working principles of single- and multi-objective optimization algorithms because of which the solution of a MOOP must be attempted using a multi-objective optimization technique. In a single-objective optimization problem, the task is to find one solution (except in some specific multi-modal optimization problems, where multiple optimal solutions are sought) which optimizes the sole objective function. Extending the idea to multi-objective optimization, it may be wrongly assumed that the task in a multi-objective optimization is to find an optimal solution corresponding to each objective function. Certainly, multi-objective optimization is much more than this simple idea. We describe the concept of multi-objective optimization by using an example problem.

Let us consider the decision-making involved in buying an automobile car. Cars are available at prices ranging from a few thousand to few hundred thousand dollars. Let us take two extreme hypothetical cars, i.e. one costing about 10,000 dollars (solution 1) and another costing about a 100,000 dollars (solution 2), as shown in Fig. 15.1. If the cost is the only objective of this decision-making process, the optimal choice is solution 1. If this were the only objective to all buyers, we would have

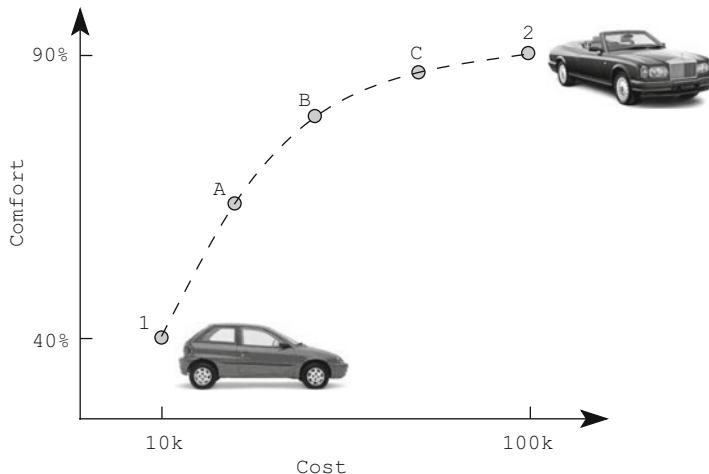


Fig. 15.1 Hypothetical trade-off solutions are illustrated for a car-buying decision-making problem

seen only one type of car (solution 1) on the road and no car manufacturer would have produced any expensive cars. Fortunately, this decision-making process is not a single-objective one. Barring some exceptions, it is expected that an inexpensive car is likely to be less comfortable. The figure indicates that the cheapest car has a hypothetical comfort level of 40 %. To rich buyers for whom comfort is the only objective of this decision-making, the choice is solution 2 (with a hypothetical maximum comfort level of 90 %, as shown in the figure). This so-called two-objective optimization problem need not be considered as the two independent optimization problems, the results of which are the two extreme solutions discussed above. Between these two extreme solutions, there exist many other solutions, where a trade-off between cost and comfort exists. A number of such solutions (solutions A, B and C) with differing costs and comfort levels are also shown in the figure. Thus, between any two such solutions, one is better in terms of one objective, but this betterment comes only from a sacrifice on the other objective. In this sense, all such trade-off solutions are optimal solutions to a MOOP. Often, such trade-off solutions provide a clear *front* on an objective space plotted with the objective values. This front is called the *Pareto-optimal* front and all such trade-off solutions are called Pareto-optimal solutions.

15.1.1 How Is It Different from Single-Objective Optimization?

It is clear from the above description that there exist a number of differences between single- and multi-objective optimization tasks. The latter have the following properties:

- Cardinality of the optimal set is usually more than one,
- There are two distinct goals of optimization, instead of one, and
- They possess two different search spaces.

We discuss each of the above properties in the following paragraphs.

First of all, we have seen from the above car-buying example that a multi-objective optimization with conflicting objectives results in a number of Pareto-optimal solutions, unlike the usual notion of only one optimal solution associated with a single-objective optimization task. However, there exist some single-objective optimization problems which also contain multiple optimal solutions (of equal or unequal importance). In some sense, multi-objective optimization is similar to that in such *multi-modal optimization* tasks. However, in principle, there is a difference, which we would like to highlight here. In most MOOPs, the Pareto-optimal solutions have certain similarities in their decision variables ([Deb 2003](#)). On the other hand, between one local or global optimal solution and another in a multi-modal optimization problem, there may not exist any such similarity. For a number of engineering case studies ([Deb 2003](#)), an analysis of the obtained trade-off solutions revealed the following properties:

- Among all Pareto-optimal solutions, some decision variables take identical values. Such a property of the decision variables means that the solution is an optimum solution.
- Other decision variables take different values causing the solutions to have a trade-off in their objective values.

Secondly, unlike the sole goal of finding the optimum in a single-objective optimization, here there are two distinct goals:

- Convergence to the Pareto-optimal solutions and
- Maintenance of a set of maximally spread Pareto-optimal solutions.

In a sense, these goals are independent of each other. An optimization algorithm must have specific properties for achieving each of the goals.

One other difference between single-objective and multi-objective optimization is that in multi-objective optimization the objective functions constitute a multi-dimensional space, in addition to the usual decision variable space common to all optimization problems. This additional space is called the *objective space*, \mathcal{Z} . For each solution \mathbf{x} in the, there exists a point in the objective space, denoted by $\mathbf{f}(\mathbf{x}) = \mathbf{z} = (z_1, z_2, \dots, z_M)^T$. The mapping takes place between an n -dimensional solution vector and an M -dimensional objective vector. Figure 15.2 illustrates these two spaces and a mapping between them. Although the search process of an algorithm takes place on the decision variables space, many interesting algorithms (particularly MOEAs) use the objective space information in their search operators. However, the presence of two different spaces introduces a number of interesting flexibilities in designing a search algorithm for multi-objective optimization.

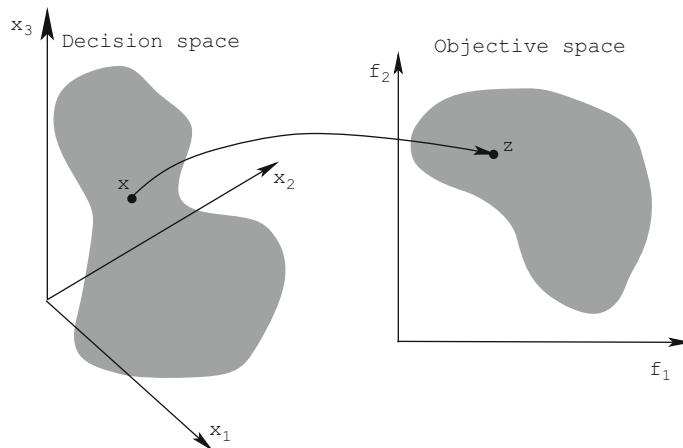


Fig. 15.2 Representation of the decision variable space and the corresponding objective space

15.2 Two Approaches to Multi-objective Optimization

Although the fundamental difference between single- and multiple-objective optimization lies in the cardinality in the optimal set, from a practical standpoint a user needs only one solution, no matter whether the associated optimization problem is single-objective or multi-objective. In the case of multi-objective optimization, the user is now in a dilemma. Which of these optimal solutions must one choose? Let us try to answer this question for the case of the car-buying problem. Knowing the number of solutions that exist in the market with different trade-offs between cost and comfort, which car does one buy? This is not an easy question to answer. It involves many other considerations, such as the total finance available to buy the car, distance to be driven each day, number of passengers riding in the car, fuel consumption and cost, depreciation value, road conditions where the car is to be mostly driven, physical health of the passengers, social status and many other factors. Often, such higher-level information is non-technical, qualitative and experience-driven. However, if a set of trade-off solutions are already worked out or available, one can evaluate the pros and cons of each of these solutions based on all such non-technical and qualitative, yet still important, considerations and compare them to make a choice. Thus, in a multi-objective optimization, ideally the effort must be in finding the set of trade-off optimal solutions by considering all objectives to be important. After a set of such trade-off solutions are found, a user can then use higher-level qualitative considerations to make a choice. Therefore, we suggest the following principle for an *ideal multi-objective optimization procedure*:

- Step 1 Find multiple trade-off optimal solutions with a wide range of values for objectives.
- Step 2 Choose one of the obtained solutions using higher-level information.

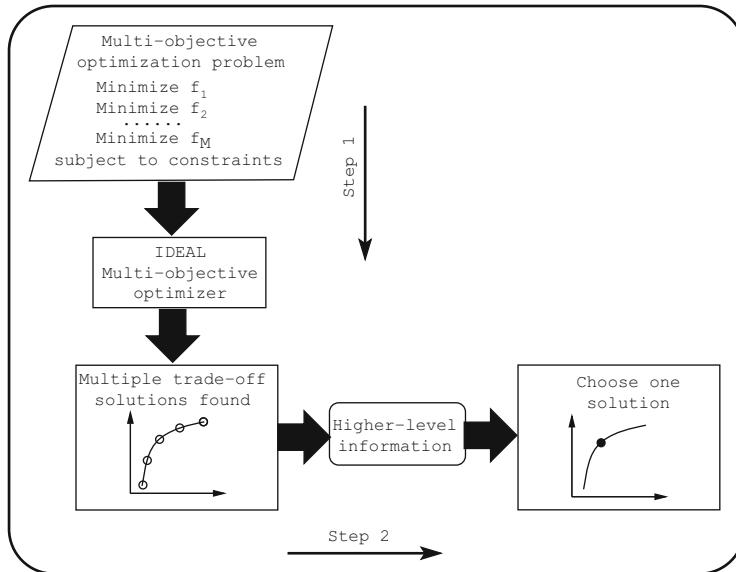


Fig. 15.3 Schematic of an ideal multi-objective optimization procedure

Figure 15.3 shows schematically the principles in an ideal multi-objective optimization procedure. In Step 1 (vertically downwards), multiple trade-off solutions are found. Thereafter, in Step 2 (horizontally, towards the right), higher-level information is used to choose one of the trade-off solutions. With this procedure in mind, it is easy to realize that single-objective optimization is a degenerate case of multi-objective optimization. In the case of single-objective optimization with only one global optimal solution, Step 1 will find only one solution, thereby not requiring us to proceed to Step 2. In the case of single-objective optimization with multiple global optima, both steps are necessary to first find all or many of the global optima and then to choose one from them by using the higher-level information about the problem.

If thought of carefully, each trade-off solution corresponds to a specific order of importance of the objectives. It is clear from Fig. 15.1 that solution A assigns more importance to cost than to comfort. On the other hand, solution C assigns more importance to comfort than to cost. Thus, if such a relative preference factor among the objectives is known for a specific problem, there is no need to follow the above principle for solving a MOOP. A simple method would be to form a composite objective function as the weighted sum of the objectives, where a weight for an objective is proportional to the preference factor assigned to that particular objective. This method of scalarizing an objective vector into a single composite objective function converts the MOOP into a single-objective optimization problem. When such a composite objective function is optimized, in most cases it is possible to obtain one particular trade-off solution. This procedure of handling MOOPs is much simpler, though still being more subjective than the above ideal procedure. We call

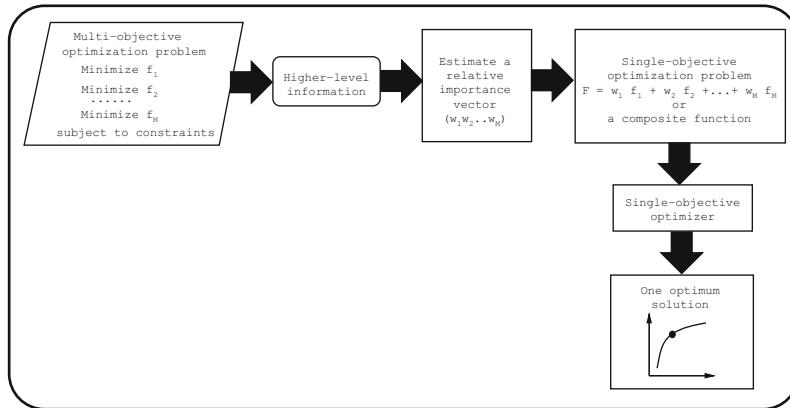


Fig. 15.4 Schematic of a preference-based multi-objective optimization procedure

this procedure a *preference-based* multi-objective optimization. A schematic of this procedure is shown in Fig. 15.4. Based on the higher-level information, a preference vector \mathbf{w} is first chosen. Thereafter, the preference vector is used to construct the composite function, which is then optimized to find a single trade-off optimal solution by a single-objective optimization algorithm. Although not often practiced, the procedure can be used to find multiple trade-off solutions by using a different preference vector and repeating the above procedure.

It is important to appreciate that the trade-off solution obtained by using the preference-based strategy is largely sensitive to the relative preference vector used in forming the composite function. A change in this preference vector will result in a (hopefully) different trade-off solution. Besides this difficulty, it is intuitive to realize that finding a relative preference vector itself is highly subjective and not straightforward. This requires an analysis of the non-technical, qualitative and experience-driven information to find a quantitative relative preference vector. Without any knowledge of the likely trade-off solutions, this is an even more difficult task. Classical multi-objective optimization methods which convert multiple objectives into a single objective by using a relative preference vector of objectives work according to this preference-based strategy. Unless a reliable and accurate preference vector is available, the optimal solution obtained by such methods is highly subjective to the particular user.

The ideal multi-objective optimization procedure suggested earlier is less subjective. In Step 1, a user does not need any relative preference vector information. The task there is to find as many different trade-off solutions as possible. Once a well-distributed set of trade-off solutions is found, Step 2 then requires certain problem information in order to choose one solution. It is important to mention that in Step 2, the problem information is used to evaluate and compare each of the obtained trade-off solutions. In the ideal approach, the problem information is not used to search for a *new* solution; instead, it is used to choose one solution from a set of already obtained trade-off solutions. Thus, there is a fundamental difference in

using the problem information in both approaches. In the preference-based approach, a relative preference vector needs to be supplied without any knowledge of the possible consequences. However, in the proposed ideal approach, the problem information is used to choose one solution from the obtained set of trade-off solutions. We argue that the ideal approach in this matter is more methodical, more practical, and less subjective. At the same time, we highlight the fact that if a reliable relative preference vector is available to a problem, there is no reason to find other trade-off solutions. In such a case, a preference-based approach would be adequate.

In the next section, we make the above qualitative idea of multi-objective optimization more quantitative.

15.3 Non-dominated Solutions and Pareto-Optimal Solutions

Most multi-objective optimization algorithms use the concept of dominance in their search. Here, we define the concept of dominance and related terms and present a number of techniques for identifying dominated solutions in a finite population of solutions.

15.3.1 Special Solutions

We first define some special solutions which are often used in multi-objective optimization algorithms.

15.3.1.1 Ideal Objective Vector

For each of the M conflicting objectives, there exists one different optimal solution. An objective vector constructed with these individual optimal objective values constitutes the ideal objective vector.

Definition 15.1. The m th component of the ideal objective vector \mathbf{z}^* is the constrained minimum solution of the following problem:

$$\left. \begin{array}{l} \text{Minimize } f_m(\mathbf{x}) \\ \text{subject to } \mathbf{x} \in \mathcal{S} \end{array} \right\}. \quad (15.1)$$

Thus, if the minimum solution for the m th objective function is the decision vector $\mathbf{x}^{*(m)}$ with function value f_m^* , the ideal vector is as follows:

$$\mathbf{z}^* = \mathbf{f}^* = (f_1^*, f_2^*, \dots, f_M^*)^T.$$

In general, the ideal objective vector (\mathbf{z}^*) corresponds to a non-existent solution (Fig. 15.5). This is because the minimum solution of Eq. (15.1) for each objective function need not be the same solution. The only way an ideal objective vector

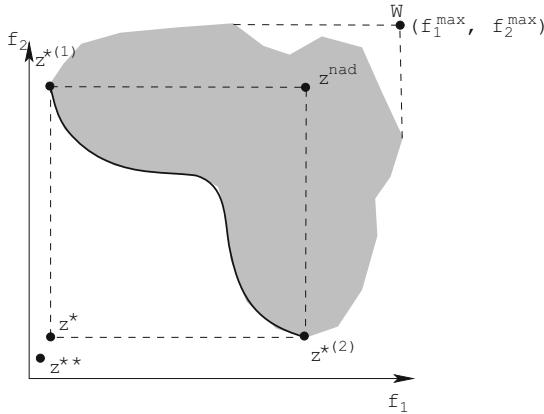


Fig. 15.5 The ideal, utopian and nadir objective vectors

corresponds to a feasible solution is when the minimal solutions to all objective functions are identical. In this case, the objectives are not conflicting to each other and the minimum solution to any objective function would be the only optimal solution to the MOOP. Although the ideal objective vector is usually non-existent, it is also clear from Fig. 15.5 that solutions closer to the ideal objective vector are better. Moreover, many algorithms require the knowledge of the lower bound on each objective function to normalize objective values in a common range.

15.3.1.2 Utopian Objective Vector

The ideal objective vector denotes an array of the lower bound of all objective functions. This means that for every objective function there exists at least one solution in the feasible search space sharing an identical value with the corresponding element in the ideal solution. Some algorithms may require a solution which has an objective value strictly better than (and not equal to) that of any solution in the search space. For this purpose, the utopian objective vector is defined as follows.

Definition 15.2. A utopian objective vector \mathbf{z}^{**} has each of its components marginally smaller than that of the ideal objective vector, or $\mathbf{z}_i^{**} = \mathbf{z}_i^* - \varepsilon_i$ with $\varepsilon_i > 0$ for all $i = 1, 2, \dots, M$.

Figure 15.5 shows a utopian objective vector. Like the ideal objective vector, the utopian objective vector also represents a non-existent solution.

15.3.1.3 Nadir Objective Vector

Unlike the ideal objective vector which represents the lower bound of each objective in the entire feasible search space, the nadir objective vector \mathbf{z}^{nad} represents

the upper bound of each objective in the entire Pareto-optimal set, and not in the entire search space. A nadir objective vector must not be confused with a vector of objectives (marked as “W” in Fig. 15.5) found by using the worst feasible function values f_i^{\max} in the entire search space. The nadir objective vector may represent an existent or a non-existent solution, depending on the convexity and continuity of the Pareto-optimal set. In order to normalize each objective in the entire range of the Pareto-optimal region, the knowledge of nadir and ideal objective vectors can be used as follows:

$$f_i^{\text{norm}} = \frac{f_i - z_i^*}{z_i^{\text{nad}} - z_i^*}. \quad (15.2)$$

15.3.2 Concept of Domination

Most multi-objective optimization algorithms use the concept of domination. In these algorithms, two solutions are compared on the basis of whether one dominates the other or not. We will describe the concept of domination in the following paragraph.

We assume that there are M objective functions. In order to cover both minimization and maximization of objective functions, we use the operator \triangleleft between two solutions i and j as $i \triangleleft j$ to denote that solution i is better than solution j on a particular objective. Similarly, $i \triangleright j$ for a particular objective implies that solution i is worse than solution j on this objective. For example, if an objective function is to be minimized, the operator \triangleleft would mean the “ $<$ ” operator, whereas if the objective function is to be maximized, the operator \triangleleft would mean the “ $>$ ” operator. The following definition covers mixed problems with minimization of some objective functions and maximization of the rest of them.

Definition 15.3. A solution $\mathbf{x}^{(1)}$ is said to dominate the other solution $\mathbf{x}^{(2)}$ if both conditions 1 and 2 are true:

1. The solution $\mathbf{x}^{(1)}$ is no worse than $\mathbf{x}^{(2)}$ in all objectives, or $f_j(\mathbf{x}^{(1)}) \leq f_j(\mathbf{x}^{(2)})$ for all $j = 1, 2, \dots, M$.
2. The solution $\mathbf{x}^{(1)}$ is strictly better than $\mathbf{x}^{(2)}$ in at least one objective, or $f_{\bar{j}}(\mathbf{x}^{(1)}) < f_{\bar{j}}(\mathbf{x}^{(2)})$ for at least one $\bar{j} \in \{1, 2, \dots, M\}$.

If either of these conditions is violated, the solution $\mathbf{x}^{(1)}$ does not dominate the solution $\mathbf{x}^{(2)}$. If $\mathbf{x}^{(1)}$ dominates the solution $\mathbf{x}^{(2)}$ (or mathematically $\mathbf{x}^{(1)} \preceq \mathbf{x}^{(2)}$), it is also customary to write any of the following:

- $\mathbf{x}^{(2)}$ is dominated by $\mathbf{x}^{(1)}$
- $\mathbf{x}^{(1)}$ is non-dominated by $\mathbf{x}^{(2)}$ or
- $\mathbf{x}^{(1)}$ is non-inferior to $\mathbf{x}^{(2)}$.

Let us consider a two-objective optimization problem with five different solutions shown in the objective space, as illustrated in Fig. 15.6a. Let us also assume

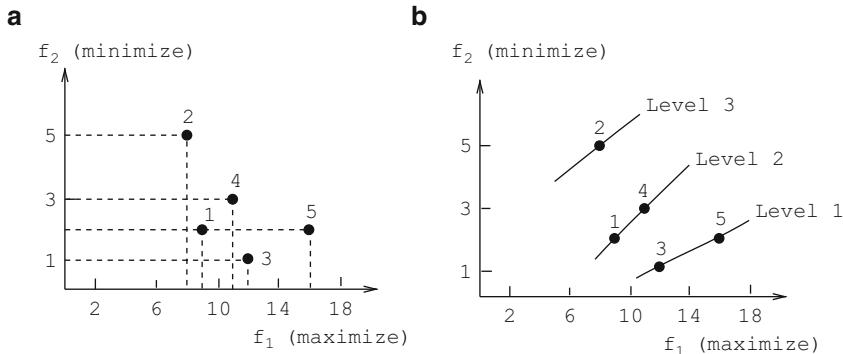


Fig. 15.6 A set of five solutions and the corresponding non-dominated fronts

that the objective function 1 needs to be maximized while the objective function 2 needs to be minimized. Five solutions with different objective function values are shown in this figure. Since both objective functions are of importance to us, it is usually difficult to find one solution which is best with respect to both objectives. However, we can use the above definition of domination to decide which solution is better among any two given solutions in terms of both objectives. For example, if solutions 1 and 2 are to be compared, we observe that solution 1 is better than solution 2 in objective function 1 and solution 1 is also better than solution 2 in objective function 2. Thus, both of the above conditions for domination are satisfied and we may write that solution 1 dominates solution 2. We take another instance of comparing solutions 1 and 5. Here, solution 5 is better than solution 1 in the first objective and solution 5 is no worse (in fact, they are equal) than solution 1 in the second objective. Thus, both the above conditions for domination are also satisfied and we may write that solution 5 dominates solution 1.

It is intuitive that if a solution $\mathbf{x}^{(1)}$ dominates another solution $\mathbf{x}^{(2)}$, the solution $\mathbf{x}^{(1)}$ is better than $\mathbf{x}^{(2)}$ in the parlance of multi-objective optimization. Since the concept of domination allows a way to compare solutions with multiple objectives, most multi-objective optimization methods use this domination concept to search for non-dominated solutions.

15.3.3 Properties of Dominance Relation

Definition 15.3 defines the dominance relation between any two solutions. There are three possibilities that can be the outcome of the dominance check between two solutions 1 and 2. That is (i) solution 1 dominates solution 2, (ii) solution 1 gets dominated by solution 2, or (iii) solutions 1 and 2 do not dominate each other. Let us now discuss the different binary relation properties (Cormen et al. 1990) of the dominance operator.

- *Reflexive.* The dominance relation is *not reflexive*, since any solution p does not dominate itself according to Definition 15.3. The second condition of dominance relation in Definition 15.3 does not allow this property to be satisfied.
- *Symmetric.* The dominance relation is also *not symmetric*, because $p \preceq q$ does not imply $q \preceq p$. In fact, the opposite is true. That is, if p dominates q , then q does not dominate p . Thus, the dominance relation is *asymmetric*.
- *Antisymmetric.* Since the dominance relation is not symmetric, it cannot be anti-symmetric as well.
- *Transitive.* The dominance relation is *transitive*. This is because if $p \preceq q$ and $q \preceq r$, then $p \preceq r$.

There is another interesting property that the dominance relation possesses. If solution p does not dominate solution q , this does not imply that q dominates p .

In order for a binary relation to qualify as an ordering relation, it must be at least transitive (Chankong and Haimes 1983). Thus, the dominance relation qualifies as an ordering relation. Since the dominance relation is not reflexive, it is a *strict partial order*. In general, if a relation is reflexive, antisymmetric and transitive, it is loosely called a *partial order* and a set on which a partial order is defined is called a *partially ordered set*. However, it is important to note that the dominance relation is not reflexive and is not antisymmetric. Thus, the dominance relation is not a partial-order relation in its general sense. The dominance relation is only a strict partial-order relation.

15.3.4 Pareto Optimality

Continuing with the comparisons in the previous section, let us compare solutions 3 and 5 in Fig. 15.6, because this comparison reveals an interesting aspect. We observe that solution 5 is better than solution 3 in the first objective, while solution 5 is worse than solution 3 in the second objective. Thus, the first condition is not satisfied for both of these solutions. This simply suggests that we cannot conclude that solution 5 dominates solution 3, nor can we say that solution 3 dominates solution 5. When this happens, it is customary to say that solutions 3 and 5 are non-dominated with respect to each other. When both objectives are important, it cannot be said which of the two solutions 3 and 5 is better.

For a given finite set of solutions, we can perform all possible pair-wise comparisons and find which solution dominates which and which solutions are non-dominated with respect to each other. At the end, we expect to have a set of solutions, any two of which do not dominate each other. This set also has another property. For any solution outside of this set, we can always find a solution in this set which will dominate the former. Thus, this particular set has a property of dominating all other solutions which do not belong to this set. In simple terms, this means that the solutions of this set are better compared to the rest of the solutions. This set is given a special name. It is called the *non-dominated set* for the given set of

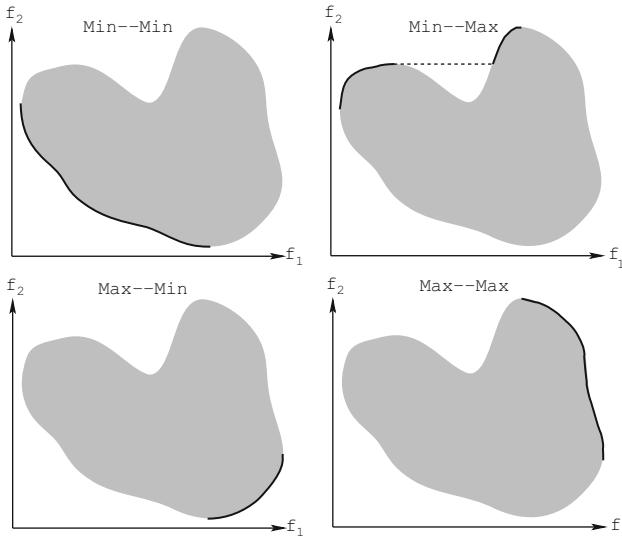


Fig. 15.7 Pareto-optimal solutions are marked with continuous curves for four combinations of two types of objectives

solutions. In the example problem, solutions 3 and 5 constitute the non-dominated set of the given set of five solutions. Thus, we define a set of non-dominated solutions as follows.

Definition 15.4 (Non-dominated set). Among a set of solutions P , the non-dominated set of solutions P' are those that are not dominated by any member of the set P .

When the set P is the entire search space, or $P = \mathcal{S}$, the resulting non-dominated set P' is called the *Pareto-optimal set*. Figure 15.7 marks the Pareto-optimal set with continuous curves for four different scenarios with two objectives. Each objective can be minimized or maximized. In the top-left panel, the task is to minimize both objectives f_1 and f_2 . The solid curve marks the Pareto-optimal solution set. If f_1 is to be minimized and f_2 is to be maximized for a problem having the same search space, the resulting Pareto-optimal set is different and is shown in the top-right panel. Here, the Pareto-optimal set is a union of two disconnected Pareto-optimal regions. Similarly, the Pareto-optimal sets for two other cases—(maximizing f_1 , minimizing f_2) and (maximizing f_1 , maximizing f_2)—are shown in the bottom-left and bottom-right panels, respectively. In any case, the Pareto-optimal set always consists of solutions from a particular edge of the feasible search region.

It is important to note that an MOEA can be easily used to handle all of the above cases by simply using the domination definition. However, to avoid any confusion, most applications use the duality principle (Deb 1995) to convert a maximization problem into a minimization problem and treat every problem as a combination of

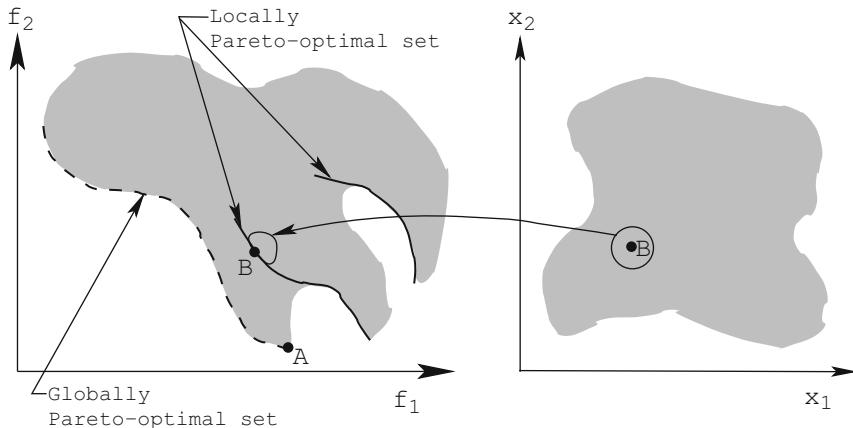


Fig. 15.8 Locally and globally Pareto-optimal solutions

minimizing all objectives. Like global and local optimal solutions in the case of single-objective optimization, there could be global and local Pareto-optimal sets in multi-objective optimization.

Definition 15.5 (Globally Pareto-optimal set). The non-dominated set of the entire feasible search space \mathcal{S} is the globally Pareto-optimal set.

Definition 15.6. If for every member \mathbf{x} in a set \underline{P} there exists no solution \mathbf{y} (in the neighborhood of \mathbf{x} such that $\|\mathbf{y} - \mathbf{x}\|_\infty \leq \varepsilon$, where ε is a small positive number) dominating any member of the set \underline{P} , then solutions belonging to the set \underline{P} constitute a locally Pareto-optimal set.

Figure 15.8 shows two locally Pareto-optimal sets (marked by continuous curves).

When any solution (say “B”) in this set is perturbed locally in the decision variable space, no solution can be found dominating any member of the set. It is interesting to note that for continuous search space problems, the locally Pareto-optimal solutions need not be continuous in the decision variable space and the above definition will still hold good. Zitzler (1999) added a neighborhood constraint on the objective space in the above definition to make it more generic. By the above definition, it is also true that a globally Pareto-optimal set is also a locally Pareto-optimal set.

15.3.5 Procedure for Finding Non-dominated Solutions

Finding the non-dominated set of solutions from a given set of solutions is similar in principle to finding the minimum of a set of real numbers. In the latter case, when two numbers are compared to identify the smaller number, a ‘<’ relation operation is

used. In the case of finding the non-dominated set, the dominance relation \preceq can be used to identify the better of two given solutions. Here, we discuss one simple procedure for finding the non-dominated set (we call here the best non-dominated front). Many MOEAs require to find the best non-dominated solutions of a population and some MOEAs require to sort a population according to different non-domination levels. We present one algorithm for each of the tasks.

15.3.5.1 Finding the Best Non-dominated Front

In this approach, every solution from the population is checked with a partially filled population for domination. To start with, the first solution from the population is kept in an empty set P' . Thereafter, each solution i (the second solution onwards) is compared with all members of the set P' , one by one. If the solution i dominates any member of P' , then that solution is removed from P' . In this way non-members of the non-dominated solutions get deleted from P' . Otherwise, if solution i is dominated by any member of P' , the solution i is ignored. If solution i is not dominated by any member of P' , it is entered in P' . This is how the set P' grows with non-dominated solutions. When all solutions of the population are checked, the remaining members of P' constitute the non-dominated set.

Identifying the non-dominated set

- Step 1 Initialize $P' = \{1\}$. Set solution counter $i = 2$.
- Step 2 Set $j = 1$.
- Step 3 Compare solution i with j from P' for domination.
- Step 4 If i dominates j , then delete the j th member from P' or else update $P' = P' \setminus \{P'^{(j)}\}$. If $j < |P'|$, increment j by one and then go to Step 3. Otherwise, go to Step 5. Alternatively, if the j th member of P' dominates i , increment i by one and then go to Step 2.
- Step 5 Insert i in P' or update $P' = P' \cup \{i\}$. If $i < N$, increment i by one and go to Step 2. Otherwise, stop and declare P' as the non-dominated set.

Here, we observe that the second element of the population is compared with only one solution P' , the third solution with at most two solutions of P' , and so on. This requires a maximum of $1 + 2 + \dots + (N - 1)$ or $N(N - 1)/2$ domination checks. This computation is also $O(MN^2)$. It is interesting to note that the size of P' may not always increase (dominated solutions will get deleted from P') and not every solution in the population may be required to be checked with all solutions in the current P' set (the solution may get dominated by a solution of P'). Thus, the actual computational complexity may be smaller than the above estimate.

Another study (Kung et al. 1975) suggested a binary-search-like algorithm for finding the best non-dominated front with a complexity $O(N(\log N)^{M-2})$ for $M \geq 4$ and $O(N \log N)$ for $M = 2$ and 3.

15.3.5.2 A Non-dominated Sorting Procedure

Using the above procedure, each front can be identified with at most $O(MN^2)$ computations. In certain scenarios, this procedure may demand more than $O(MN^2)$ computational effort for the overall non-dominated sorting of a population. Here, we suggest a completely different procedure which uses a better bookkeeping strategy requiring $O(MN^2)$ overall computational complexity.

First, for each solution we calculate two entities: (i) *domination count* n_i , the number of solutions which dominate the solution i , and (ii) S_i , a set of solutions which the solution i dominates. This requires $O(MN^2)$ comparisons. At the end of this procedure, all solutions in the first non-dominated front will have their domination count as zero. Now, for each of these solutions (each solution i with $n_i = 0$), we visit each member (j) of its set S_i and reduce its domination count by one. In doing so, if for any member j the domination count becomes zero, we put it in a separate list P' . After such modifications on S_i are performed for each i with $n_i = 0$, all solutions of P' would belong to the second non-dominated front. The above procedure can be continued with each member of P' and the third non-dominated front can be identified. This process continues until all solutions are classified.

An $O(MN^2)$ non-dominated sorting algorithm

- Step 1 For each $i \in P$, $n_i = 0$ and initialize $S_i = \emptyset$. For all $j \neq i$ and $j \in P$, perform Step 2 and then proceed to Step 3.
- Step 2 If $i \preceq j$, update $S_p = S_p \cup \{j\}$. Otherwise, if $j \preceq i$, set $n_i = n_i + 1$.
- Step 3 If $n_i = 0$, keep i in the first non-dominated front P_1 (we called this set P' in the above paragraph). Set a front counter $k = 1$.
- Step 4 While $P_k \neq \emptyset$, perform the following steps.
- Step 5 Initialize $Q = \emptyset$ for storing next non-dominated solutions. For each $i \in P_k$ and for each $j \in S_i$,

 - Step 5a Update $n_j = n_j - 1$.
 - Step 5b If $n_j = 0$, keep j in Q , or perform $Q = Q \cup \{j\}$.

- Step 6 Set $k = k + 1$ and $P_k = Q$. Go to Step 4.

Steps 1–3 find the solutions in the first non-dominated front and require $O(MN^2)$ computational complexity. Steps 4–6 repeatedly find higher fronts and require at most $O(N^2)$ comparisons, as argued below. For each solution i in the second- or higher level of non-domination, the domination count n_i can be at most $N - 1$. Thus, each solution i will be visited at most $N - 1$ times before its domination count becomes zero. At this point, the solution is assigned a particular non-domination level and will never be visited again. Since there are at most $N - 1$ such solutions, the complexity of identifying second and more fronts is $O(N^2)$. Thus, the overall complexity of the procedure is $O(MN^2)$. It is important to note that although the time complexity has reduced to $O(MN^2)$, the storage requirement has increased to $O(N^2)$.

When the above procedure is applied to the five solutions of Fig. 15.6a, we obtain three non-dominated fronts as shown in Fig. 15.6b. From the dominance relations, the solutions 3 and 5 are the best, followed by solutions 1 and 4. Finally, solution 2

belongs to the worst non-dominated front. Thus, the ordering of solutions in terms of their non-domination level is as follows: ((3,5), (1,4), (2)). A study ([Jensen 2003b](#)) suggested a divided-and-conquer method to reduce the complexity of sorting to $O(N \log^{M-1} N)$.

15.4 Some Approaches to Multi-objective Optimization

In this section, we briefly mention two commonly used classical multi-objective optimization methods and thereafter present a commonly used EMO method.

15.4.1 Classical Method: Weighted-Sum Approach

The weighted-sum method, as the name suggests, scalarizes a set of objectives into a single objective by pre-multiplying each objective with a user-supplied weight. This method is the simplest approach and is probably the most widely used classical approach. If we are faced with the two objectives of minimizing the cost of a product and minimizing the amount of wasted material in the process of fabricating the product, one naturally thinks of minimizing a weighted sum of these two objectives. Although the idea is simple, it introduces a not-so-simple question. What values of the weights must one use? Of course, there is no unique answer to this question. The answer depends on the importance of each objective in the context of the problem and a scaling factor. The scaling effect can be avoided somewhat by normalizing the objective functions. After the objectives are normalized, a composite objective function $F(\mathbf{x})$ can be formed by summing the weighted normalized objectives and the problem is then converted to a single-objective optimization problem as follows:

$$\begin{aligned} & \text{Minimize } F(\mathbf{x}) = \sum_{m=1}^M w_m f_m(\mathbf{x}) \\ & \text{subject to } g_j(\mathbf{x}) \geq 0, \quad j = 1, 2, \dots, J \\ & \quad h_k(\mathbf{x}) = 0, \quad k = 1, 2, \dots, K \\ & \quad x_i^{(L)} \leq x_i \leq x_i^{(U)}, \quad i = 1, 2, \dots, n. \end{aligned} \quad (15.3)$$

Here, w_m ($\in [0, 1]$) is the weight of the m th objective function. Since the minimum of the above problem does not change if all weights are multiplied by a constant, it is the usual practice to choose weights such that their sum is one, or $\sum_{m=1}^M w_m = 1$.

Mathematically oriented readers may find a number of interesting theorems regarding the relationship between the optimal solution of the above problem to the true Pareto-optimal solutions in classical texts ([Chankong and Haimes 1983; Miettinen 1999; Ehrgott 2000](#)).

Let us now illustrate how the weighted-sum approach can find Pareto-optimal solutions of the original problem. For simplicity, we consider the two-objective problem shown in Fig. 15.9. The feasible objective space and the corresponding

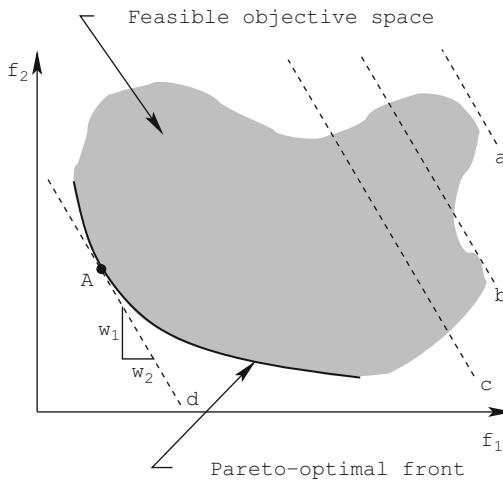


Fig. 15.9 Illustration of the weighted-sum approach on a convex Pareto-optimal front

Pareto-optimal solution set are shown. With two objectives, there are two weights w_1 and w_2 , but only one is independent. Knowing any one, the other can be calculated by simple subtraction. It is clear from the figure that a choice of a weight vector corresponds to a pre-destined optimal solution on the Pareto-optimal front, as marked by the point A. By changing the weight vector, a different Pareto-optimal point can be obtained. However, there are a couple of difficulties with this approach:

1. A uniform choice of weight vectors does not necessarily find a uniform set of Pareto-optimal solutions on the Pareto-optimal front (Deb 2001).
2. The procedure cannot be used to find Pareto-optimal solutions which lie on the non-convex portion of the Pareto-optimal front.

The former issue makes it difficult for the weighted-sum approach to be applied reliably to any problem in order to find a good representative set of Pareto-optimal solutions. The latter issue arises due to the fact that a solution lying on the non-convex Pareto-optimal front can never be the optimal solution of the problem given in Eq. (15.3).

15.4.2 Classical Method: ϵ -Constraint Method

In order to alleviate the difficulties faced by the weighted-sum approach in solving problems having non-convex objective spaces, the ϵ -constraint method is used. Haimes et al. (1971) suggested reformulating the MOOP by just keeping one of the objectives and restricting the rest of the objectives within user-specified values. The modified problem is as follows:

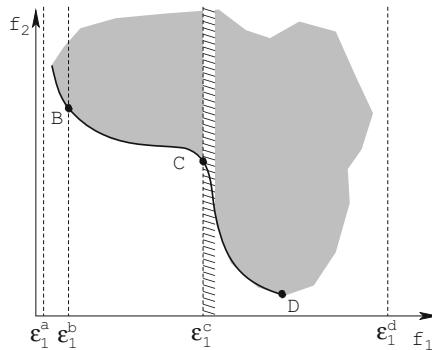


Fig. 15.10 The ϵ -constraint method

$$\left. \begin{array}{l} \text{Minimize } f_\mu(\mathbf{x}) \\ \text{subject to } f_m(\mathbf{x}) \leq \epsilon_m, \quad m = 1, 2, \dots, M \text{ and } m \neq \mu \\ g_j(\mathbf{x}) \geq 0, \quad j = 1, 2, \dots, \\ h_k(\mathbf{x}) = 0, \quad k = 1, 2, \dots, K \\ x_i^{(L)} \leq x_i \leq x_i^{(U)}, i = 1, 2, \dots, n. \end{array} \right\} \quad (15.4)$$

In the above formulation, the parameter ϵ_m represents an upper bound of the value of f_m and need not necessarily mean a small value close to zero.

Let us say that we retain f_2 as an objective and treat f_1 as a constraint: $f_1(\mathbf{x}) \leq \epsilon_1$. Figure 15.10 shows four scenarios with different ϵ_1 values. Let us consider the third scenario with $\epsilon_1 = \epsilon_1^c$ first. The resulting problem with this constraint divides the original feasible objective space into two portions, $f_1 \leq \epsilon_1^c$ and $f_1 > \epsilon_1^c$. The left portion becomes the feasible solution of the resulting problem stated in Eq. (15.4). Now, the task of the resulting problem is to find the solution which minimizes this feasible region. From Fig. 15.10, it is clear that the minimum solution is C. In this way, intermediate Pareto-optimal solutions can be obtained in the case of non-convex objective space problems by using the ϵ -constraint method.

One of the difficulties of this method is that the solution to the problem stated in Eq. (15.4) largely depends on the chosen ϵ vector. Let us refer to Fig. 15.10 again. Instead of choosing ϵ_1^c , if ϵ_1^a is chosen, there exists no feasible solution to the stated problem. Thus, no solution would be found. On the other hand, if ϵ_1^d is used, the entire search space is feasible. The resulting problem has the minimum at D. Moreover, as the number of objectives increases, there exist more elements in the ϵ vector, thereby requiring more information from the user.

15.4.3 Evolutionary Multi-objective Optimization (EMO) Method

Over the years, a number of multi-objective EAs () emphasizing non-dominated solutions in a EA population have been suggested. In this section, we shall describe one state-of-the-art algorithm popularly used in EMO studies.

15.4.3.1 Elitist Non-dominated Sorting GA (NSGA-II)

The non-dominated sorting GA or NSGA-II procedure (Deb et al. 2002) for finding multiple Pareto-optimal solutions in a MOOP has the following three features:

1. It uses an elitist principle,
2. It uses an explicit diversity preserving mechanism, and
3. It emphasizes the non-dominated solutions.

In NSGA-II, the offspring population Q_t is first created by using the parent population P_t and the usual genetic operators (Goldberg 1989). Thereafter, the two populations are combined to form R_t of size $2N$. Then, a non-dominated sorting is used to classify the entire population R_t . Once the non-dominated sorting is over, the new population is filled by solutions of different non-dominated fronts, one at a time. The filling starts with the best non-dominated front and continues with solutions of the second non-dominated front, followed by the third non-dominated front, and so on. Since the overall population size of R_t is $2N$, not all fronts may be accommodated in N slots available in the new population. All fronts which could not be accommodated are simply deleted. When the last allowed front is being considered, there may exist more solutions in the last front than the remaining slots in the new population. This scenario is illustrated in Fig. 15.11. Instead of arbitrarily discarding some members from the last acceptable front, the solutions which will make the diversity of the selected solutions the highest are chosen. The NSGA-II procedure is outlined in the following.

NSGA-II

Step 1 Combine parent and offspring populations and create $R_t = P_t \cup Q_t$. Perform a non-dominated sorting to R_t and identify different fronts: \mathcal{F}_i , $i = 1, 2, \dots$, etc.

Step 2 Set new population $P_{t+1} = \emptyset$. Set a counter $i = 1$.

Until $|P_{t+1}| + |\mathcal{F}_i| < N$, perform $P_{t+1} = P_{t+1} \cup \mathcal{F}_i$ and $i = i + 1$.

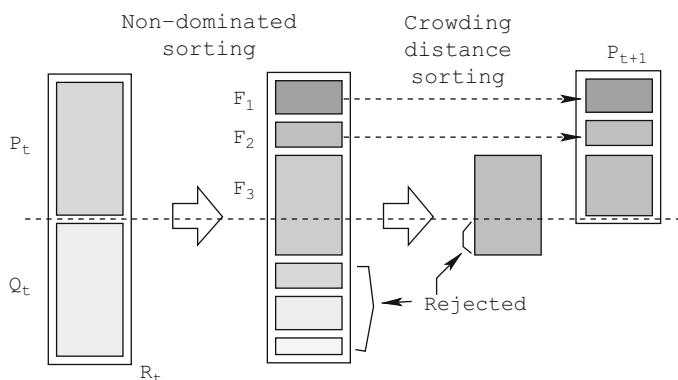


Fig. 15.11 Schematic of the NSGA-II procedure

Step 3 Perform the *Crowding-sort* ($\mathcal{F}_i, <_c$) procedure and include the most widely spread ($N - |P_{t+1}|$) solutions by using the crowding distance values in the sorted \mathcal{F}_i to P_{t+1} .

Step 4 Create offspring population Q_{t+1} from P_{t+1} by using the crowded tournament selection, crossover and mutation operators.

In Step 3, the crowding-sorting of the solutions of front i (the last front which could not be accommodated fully) is performed by using a *crowding distance metric*, which we describe later. The population is arranged in descending order of magnitude of the crowding distance values. In Step 4, a crowding tournament selection operator, which also uses the crowding distance, is used.

The crowded comparison operator ($<_c$) compares two solutions and returns the winner of the tournament. It assumes that every solution i has two attributes:

1. A non-domination rank r_i in the population,
2. A local (d_i) in the population.

The crowding distance d_i of a solution i is a measure of the normalized search space around i which is not occupied by any other solution in the population. Based on these two attributes, we can define the crowded tournament selection operator as follows.

Definition 15.7. *Crowded tournament selection operator:* A solution i wins a tournament with another solution j if any of the following conditions are true:

1. If solution i has a better rank, that is, $r_i < r_j$.
2. If they have the same rank but solution i has a better crowding distance than solution j , that is, $r_i = r_j$ and $d_i > d_j$.

The first condition makes sure that the chosen solution lies on a better non-dominated front. The second condition resolves the tie of both solutions being on the same non-dominated front by deciding on their crowded distance. The one residing in a less crowded area (with a larger crowding distance d_i) wins. The crowding distance d_i can be computed in various ways. However, in NSGA-II, we use a crowding distance metric, which requires $O(MN \log N)$ computations.

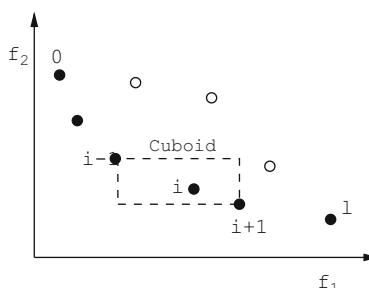


Fig. 15.12 The crowding distance calculation

To get an estimate of the density of solutions surrounding a particular solution i in the population, we take the average distance of two solutions on either side of solution i along each of the objectives. This quantity d_i serves as an estimate of the perimeter of the cuboid formed by using the nearest neighbors as the vertices (we call this the *crowding distance*). In Fig. 15.12, the crowding distance of the i th solution in its front (marked with filled circles) is the average side-length of the cuboid (shown by a dashed box). The following algorithm is used to calculate the crowding distance of each point in the set \mathcal{F} .

Crowding distance assignment procedure: `Crowding-sort($\mathcal{F}, <_c$)`

- Step C1 Call the number of solutions in \mathcal{F} as $l = |\mathcal{F}|$. For each i in the set, first assign $d_i = 0$.
- Step C2 For each objective function $m = 1, 2, \dots, M$, sort the set in worse order of f_m or, find the sorted indices vector: $I^m = \text{sort}(f_m, >)$.
- Step C3 For $m = 1, 2, \dots, M$, assign a large distance to the boundary solutions, or $d_{I_1^m} = d_{I_l^m} = \infty$, and for all other solutions $j = 2$ to $(l - 1)$, assign

$$d_{I_j^m} = d_{I_j^m} + \frac{f_m^{(I_{j+1}^m)} - f_m^{(I_{j-1}^m)}}{f_m^{\max} - f_m^{\min}}.$$

Index I_j denotes the solution index of the j th member in the sorted list. Thus, for any objective, I_1 and I_l denote the lowest and highest objective function values, respectively. The second term on the right-hand side of the last equation is the difference in objective function values between two neighboring solutions on either side of solution I_j . Thus, this metric denotes half of the perimeter of the enclosing cuboid with the nearest-neighboring solutions placed on the vertices of the cuboid (Fig. 15.12). It is interesting to note that for any solution i the same two solutions $(i + 1)$ and $(i - 1)$ need not be neighbors in all objectives, particularly for $M \geq 3$. The parameters f_m^{\max} and f_m^{\min} can be set as the population-maximum and population-minimum values of the m th objective function. The above metric requires M sorting calculations in Step C2, each requiring $O(N \log N)$ computations. Step C3 requires N computations. Thus, the complexity of the above distance metric computation is $O(MN \log N)$ and the overall complexity of one generation of NSGA-II is $O(MN^2)$, governed by the non-dominated sorting procedure.

15.4.4 Sample Simulation Results

In this section, we show the simulation results of NSGA-II on two test problems. The first problem (SCH1) is simple two-objective problem with a convex Pareto-optimal front:

$$\text{SCH1 : } \begin{cases} \text{Minimize } f_1(x) = x^2 \\ \text{Minimize } f_2(x) = (x - 2)^2 \\ \quad -10^3 \leq x \leq 10^3. \end{cases} \quad (15.5)$$

The second problem (KUR) has a disjointed set of Pareto-optimal fronts:

$$\text{KUR : } \begin{cases} \text{Minimize } f_1(\mathbf{x}) = \sum_{i=1}^2 \left[-10 \exp(-0.2 \sqrt{x_i^2 + x_{i+1}^2}) \right] \\ \text{Minimize } f_2(\mathbf{x}) = \sum_{i=1}^3 [|x_i|^{0.8} + 5 \sin(x_i^3)] \\ -5 \leq x_i \leq 5, \quad i = 1, 2, 3. \end{cases} \quad (15.6)$$

NSGA-II is run with a population size of 100 and for 250 generations. Figure 15.13 shows that NSGA-II converges on the Pareto-optimal front and maintains a good spread of solutions. In comparison to NSGA-II, another competing EMO method—the Pareto archived evolution strategy (PAES) ([Knowles and Corne 2000](#))—is run for an identical overall number of function evaluations and an inferior distribution of solutions on the Pareto-optimal front is observed.

On the KUR problem, NSGA-II is compared with another elitist EMO methodology—the strength Pareto EA or SPEA ([Zitzler and Thiele 1998](#))—for an identical number of function evaluations. Figures 15.14 and 15.15 clearly show the superiority of NSGA-II in achieving both tasks of convergence and maintaining diversity of optimal solutions.

15.4.5 Other State-of-the-Art MOEAs

Besides the above elitist EMO method, there exist a number of other methods which are also quite commonly used. Of them, the strength Pareto-EA or SPEA2 ([Zitzler et al. 2001b](#)), which uses an EA population and an archive in a synergistic manner and the Pareto envelope-based selection algorithm or PESA ([Corne et al. 2000](#)),

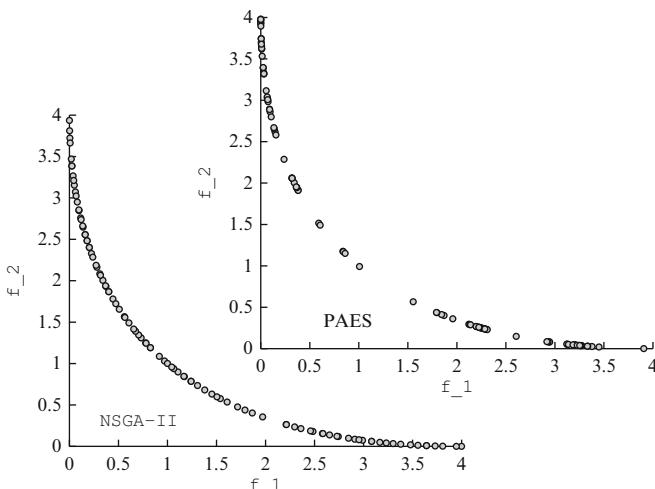


Fig. 15.13 NSGA-II finds better spread of solutions than PAES on SCH

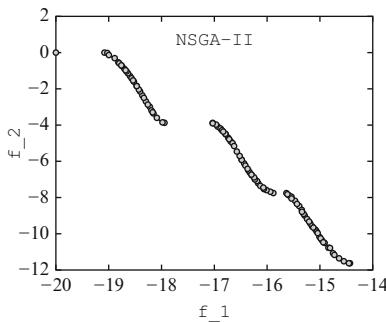


Fig. 15.14 NSGA-II on KUR

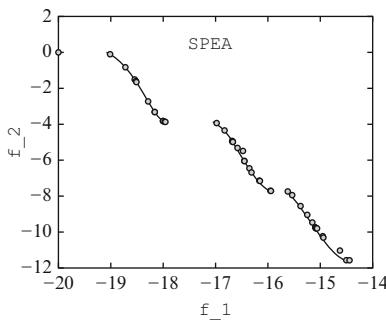


Fig. 15.15 SPEA on KUR

which emphasizes non-dominated solutions residing in a less-crowded hyper-box in both the selection and the offspring-acceptance operators, are common. The ϵ -MOEA procedure (Deb et al. 2003b) is found to be a superior version of PESA, in which only one solution is allowed to occupy a hyper-box for obtaining a better distribution of solutions. In addition, the ϵ -dominance concept (Laumanns et al. 2002a) makes the MOEA a practical approach for solving complex problems with a large number of objectives. The ϵ -MOEA is also demonstrated to find a well-converged and well-distributed set of solutions in a very small computational time (two to three orders of magnitude smaller) compared to a number of state-of-the-art MOEAs (Deb et al. 2003b), such as SPEA2 and PAES. There also exist other competent MOEAs, such as multi-objective messy GA (MOMGA) (Veldhuizen and Lamont 2000), multi-objective micro-GA (Coello and Toscano 2000), neighborhood constraint GA (Loughlin and Ranjithan 1997), and others. Further, there exist other EA-based methodologies, such as particle swarm EMO (Coello and Lechuga 2002; Mostaghim and Teich 2003), ant-based EMO (McMullen 2001; Gravel et al. 2002) and differential evolution-based EMO (Babu and Jehan 2003).

15.5 Constraint Handling

Constraints can be simply handled by modifying the definition of domination in an EMO method.

Definition 15.8. A solution $\mathbf{x}^{(i)}$ is said to “constrain-dominate” a solution $\mathbf{x}^{(j)}$ (or $\mathbf{x}^{(i)} \preceq_c \mathbf{x}^{(j)}$) if any of the following conditions are true:

1. Solution $\mathbf{x}^{(i)}$ is feasible and solution $\mathbf{x}^{(j)}$ is not.
2. Solutions $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ are both infeasible, but solution $\mathbf{x}^{(i)}$ has a smaller constraint violation.
3. Solutions $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ are feasible and solution $\mathbf{x}^{(i)}$ dominates solution $\mathbf{x}^{(j)}$ in the usual sense (see Definition 15.3).

This definition allows a feasible solution to be always dominating an infeasible solution and compares two infeasible solutions based on constraint violation values and two feasible solutions in terms of their objective values.

In the following, we show simulation results of NSGA-II applied with the above constraint handling mechanism to two test problems—the CONSTR and the problem TNK described below:

CONSTR	TNK
Minimize $f_1(\mathbf{x}) = x_1$	Minimize $f_1(\mathbf{x}) = x_1$
Minimize $f_2(\mathbf{x}) = \frac{1+x_2}{x_1}$	Minimize $f_2(\mathbf{x}) = x_2$
$x_2 + 9x_1 \geq 6$	$x_1^2 + x_2^2 - 1 - \frac{1}{10} \cos\left(16 \tan^{-1} \frac{x_1}{x_2}\right) \geq 0$
$-x_2 + 9x_1 \geq 1$	$(x_1 - 0.5)^2 + (x_2 - 0.5)^2 \leq 0.5$

With identical parameter settings as in Sect. 15.4.4, NSGA-II finds a good distribution of solutions on the Pareto-optimal front in both problems (Figs. 15.16 and 15.17, respectively).

15.6 Some Applications

Since the early development of MOEAs in 1993, they have been applied to many real-world and interesting optimization problems. Descriptions of some of these studies can be found in books (Deb 2001; Coello et al. 2002; Osyczka 2002), conference proceedings (Zitzler et al. 2001a), and domain-specific journals and conference proceedings. In this section, we describe two case studies.

15.6.1 Spacecraft Trajectory Design

Coverstone-Carroll et al. (2000) proposed a multi-objective optimization technique using the original non-dominated sorting (NSGA) (Srinivas and Deb 1994) to find

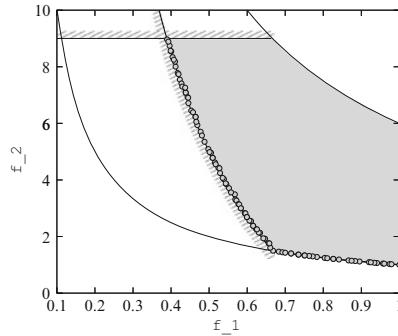


Fig. 15.16 Obtained non-dominated solutions with NSGA-II on the constrained problem CONSTR

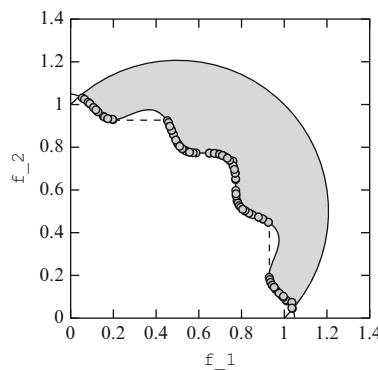


Fig. 15.17 Obtained non-dominated solutions with NSGA-II on the constrained problem TNK

multiple trade-off solutions in a spacecraft trajectory optimization problem. To evaluate a solution (trajectory), the SEPTOP software is called for, and the delivered payload mass and the total time of flight are calculated. In order to reduce the computational complexity, the SEPTOP program is run for a fixed number of generations. The MOOP had eight decision variables controlling the trajectory, as well as three objective functions, i.e. (i) maximize the delivered payload at destination, (ii) maximize the negative of the time of flight, and (iii) maximize the total number of heliocentric revolutions in the trajectory, and three constraints, i.e. (i) limiting the SEPTOP convergence error, (ii) limiting the minimum heliocentric revolutions, and (iii) limiting the maximum heliocentric revolutions in the trajectory.

On the Earth–Mars rendezvous mission, the study found interesting trade-off solutions. Using a population of size 150, the NSGA was run for 30 generations on a Sun Ultra 10 Workstation with a 333 MHz ULTRA Sparc III processor. The obtained non-dominated solutions are shown in Fig. 15.18 for two of the three objectives. It is clear that there exist short-time flights with smaller delivered payloads (solution marked as 44) and long-time flights with larger delivered payloads (solution

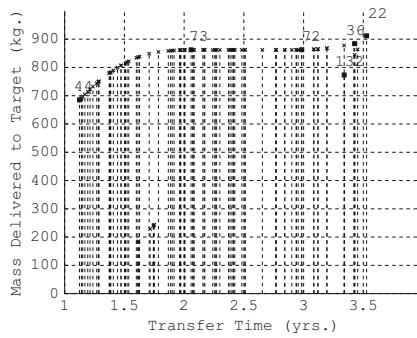


Fig. 15.18 Obtained non-dominated solutions

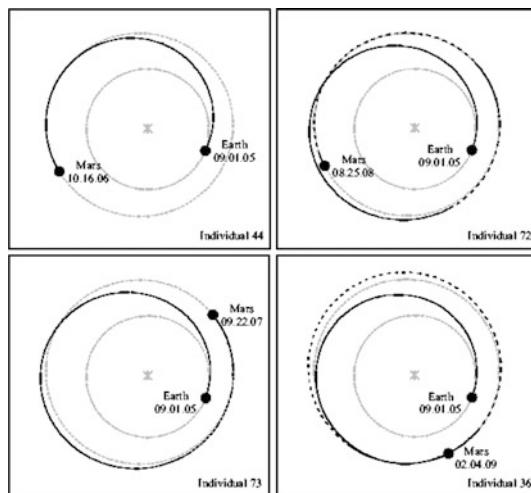


Fig. 15.19 Four trade-off trajectories

marked as 36). To the surprise of the original investigators, two different types of trajectories emerged. The representative solutions of the first set of trajectories are shown in Fig. 15.19. Solution 44 can deliver a mass of 685.28 kg and requires about 1.12 years. On the other hand, solution 72 can deliver almost 862 kg with a travel time of about 3 years. In these figures, each continuous part of a trajectory represents a thrusting arc and each dashed part of a trajectory represents a coasting arc. It is interesting to note that only a small improvement in delivered mass occurs in the solutions between 73 and 72. To move to a somewhat improved delivered mass, a different strategy for the trajectory must be found. Near solution 72, an additional burn is added, causing the trajectories to have better delivered masses. Solution 36 can deliver a mass of 884.10 kg.

The scenario as in Fig. 15.19 is what we envisaged in discovering in a MOOP while suggesting the *ideal procedure* in Fig. 15.3. Once such a set of solutions with

a good trade-off among objective values is obtained, one can then analyze them in order to choose a particular solution. For example, in this problem context, whether the wait of an extra year to be able to carry an additional 180 kg of payload is worthwhile or not would lead a decision-maker to choose between solutions 44 and 73. Without the knowledge of such a wide variety of optimal solutions, the decision-making could be difficult. Although one can set a relative weight to each objective and optimize the resulting aggregate objective function, the decision-maker will always wonder what solution would have been derived if a slightly different weight vector had been used. The ideal multi-objective optimization technique allows for a flexible and a pragmatic procedure for analyzing a well-diversified set of solutions before choosing a particular solution.

15.6.2 A Cantilever Plate Design

A rectangular plate ($1.2 \times 2 \text{ m}^2$) is fixed at one end and a 100 kN load is applied to the center element of the opposite end. The following other parameters are chosen:

- Plate thickness: 20 mm
- Yield strength: 150 MPa
- Young's modulus: 200 GPa
- Poisson ratio: 0.25.

The rectangular plate is divided into a number of grids and the presence or absence of each grid becomes a Boolean decision variable. NSGA-II is applied for 100 generations with a population size of 54 and crossover probability of 0.95. In order to increase the quality of the obtained solutions, we use an incremental grid-tuning technique. The NSGA-II and the first local search procedure are run with a coarse grid structure (6×10 or 60 elements). After the first local search procedure, each grid is divided into four equal-sized grids, thereby having a 12×20 or 240 elements. The new smaller elements inherit its parent's status of being present or absent. After the second local search is over, the elements are divided again, thereby making 24×40 or 960 elements. In all cases, an automatic mesh-generating finite element method is used to analyze the developed structure.

Figure 15.20 shows the obtained front with eight solutions: the trade-off between weight and deflection is clear. Figure 15.21 shows the shape of these eight solutions. The solutions are arranged according to increasing weight from left to right and top to bottom. Thus, the minimum-weight solution is the top-left solution and the minimum-deflection solution is the bottom-right solution.

One striking difference between single-objective optimization and multi-objective optimization is the cardinality of the solution set. In the latter, multiple solutions are the outcome and each solution is theoretically an optimal solution corresponding to a particular trade-off among the objectives. Thus, all such trade-off solutions found by an EMO are high-performing near-optimal solutions. It is intuitive to realize that these solutions will possess some common properties that qualify them to be near

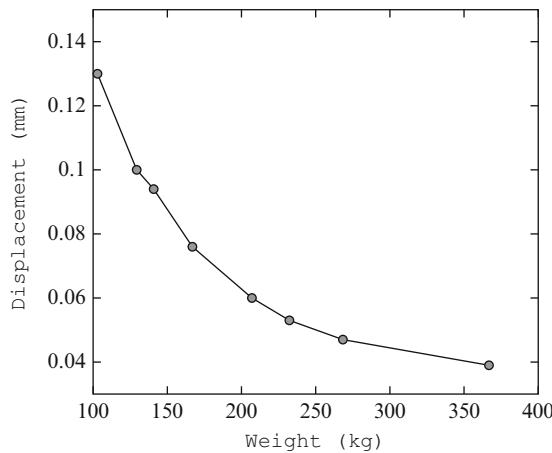


Fig. 15.20 Obtained front with eight clustered solutions shown for the cantilever plate design problem

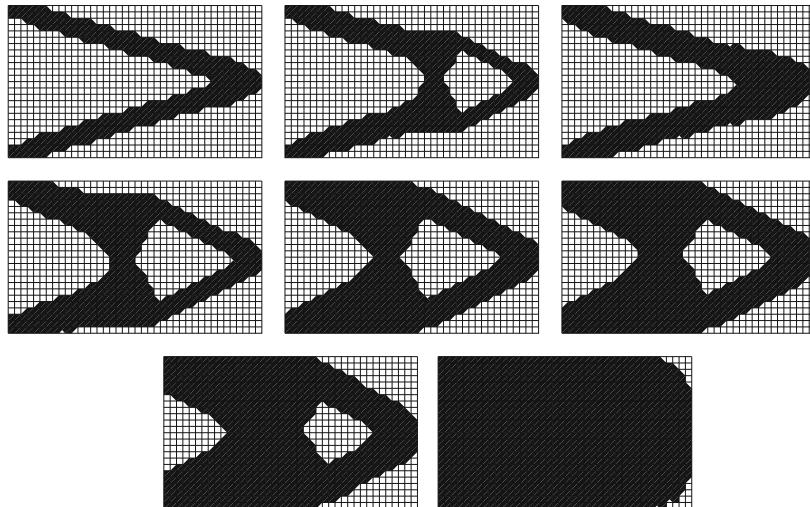


Fig. 15.21 Eight trade-off solutions of the cantilever plate design problem

Pareto-optimal. The author first proposed an analysis procedure to decipher hidden relationships common to EMO-obtained trade-off solutions in 1993 ([Deb 2003](#)) and later ([Deb and Srinivasan 2006](#)) termed the task as *innovation* procedure—revealing innovation through optimization. Such useful properties are expected to exist in practical problems, as they follow certain scientific and engineering principles at the core. In the recent past, the author and his student have devised several automated innovation procedures that take EMO solutions and churn out multiple hidden mathematical relationships of certain structure through an optimization task ([Bandaru and Deb 2010, 2011a,b](#)).

We consider the cantilever plate design problem to illustrate the usefulness of the innovation task. The obtained nine solutions are manually analyzed and the following interesting insights are revealed as properties of those solutions:

1. First, all nine solutions seem to be symmetric about the middle row of the plate. Since the loading and support are symmetrically placed around the middle row, the resulting optimum solution is also likely to be symmetric. Although this information is not explicitly coded in the hybrid NSGA-II procedure, this emerges as one of the features in all optimal solutions. Although in this problem, it is difficult to know the true Pareto-optimal solutions, the symmetry achieved in these solutions is an indication of their proximity to the true Pareto-optimal solutions.
2. The minimum-weight solution simply extends two arms from the extreme support nodes to reach to the element carrying the load. Since a straight line is the shortest way to join two points, this solution can be easily conceived as one close to the minimum-weight feasible solution.
3. Thereafter, to have a reduction in deflection, the weight has to be increased. This is where the hybrid procedure discovers an innovation. For a particular sacrifice in the weight, the procedure finds that the maximum reduction in deflection occurs when the two arms are connected by a *stiffener*. This is an engineering trick often used to design a stiff structure. Once again, no such information was explicitly coded in the hybrid procedure. By merely making elements on or off, the procedure has resulted in a *design innovation*.
4. Interestingly, the third solution is a thickened version of the minimum-weight solution. By making the arms thicker, the deflection can be increased maximally for a fixed change in the weight from the previous solution. Although not intuitive, this thick-arm solution is not an immediate trade-off solution to the minimum-weight solution. Although the deflection of this solution is smaller compared to the second solution, the stiffened solution is a good compromise between the thin- and thick-armed solutions.
5. Thereafter, any increase in the thickness of the two-armed solution turns out to be a suboptimal proposition and the stiffened solution is rediscovered instead. From the support to the stiffener, the arms are now thicker than before, providing better stiffness than before.
6. In the remaining solutions, the stiffener and arms get wider and wider, finally leading to the complete plate with rounded corners. This solution is, no doubt, close to the true minimum-deflection solution.

The transition from a simple thin two-armed cantilever plate having a minimum-weight solution to a complete plate with edges rounded off having a minimum-deflection solution proceeds by discovering a vertical stiffener connecting the two arms and then by widening the arms and then by gradually thickening the stiffener. The symmetric feature of the solutions about the middle row of the plate has emerged to be a *common* property of all obtained solutions. Such information about the trade-off solutions is very useful to a designer. Importantly, it is not obvious how such vital design information can be obtained by any other means and in a single simulation run.

15.7 Tricks of the Trade

Here, we discuss how to develop an ideal multi-objective optimization algorithm in a step-by-step manner. Since they can be developed by using classical or evolutionary optimization methods, we discuss each of these in turn.

15.7.1 Classical Multi-objective Optimization

We assume here that the user knows a classical optimization method to optimize a single-objective optimization problem P with constraints ($\mathbf{x} \in S$) and can find a near-optimum solution \mathbf{x}^* . Let us assume that the user desires to find K different efficient solutions.

- Step 1 Find individual optimum solutions \mathbf{x}^{*i} for all objectives, $i = 1, 2, \dots, M$.
- Step 2 Choose K points $\varepsilon^{(k)}$ uniformly in the $(M - 1)$ -dimensional plane having objectives $i = 1$ to $i = M - 1$ as coordinate directions.
- Step 3 Solve each subproblem ($k = 1, 2, \dots, K$) as follows:

$$\begin{aligned} & \text{Minimize } f_M(\mathbf{x}) \\ & \text{subject to } f_i(\mathbf{x}) \leq \varepsilon_i^{(k)}, \quad i = 1, 2, \dots, (M - 1) \\ & \quad \mathbf{x} \in S. \end{aligned} \tag{15.7}$$

Call the optimal solution $\mathbf{x}^{*(k)}$ and corresponding objective vector $\mathbf{f}^{*(k)}$.

- Step 4 Declare the non-dominated set, $F = \text{non-dominated}(\mathbf{f}^{*(1)}, \dots, \mathbf{f}^{*(K)})$, as the set of efficient solutions.

If desired, the above ε -constraint method can be replaced by other conversion methods, such as the weighted-sum method or the Tchebyshev metric method (Deb 2001; Miettinen 1999; Chankong and Haimes 1983).

15.7.2 Evolutionary Multi-objective Optimization (EMO)

The bottleneck of the above method is Step 3, in which a single-objective minimizer needs to be applied K times (where K is the number of desired efficient solutions). Here, we discuss an evolutionary search principle to find a set of efficient solutions simultaneously in a synergistic manner. It must be kept in mind that the main aim in an ideal multi-objective optimization is to (i) converge close to the true Pareto-optimal front and (ii) maintain a good diversity among them. Thus, an EMO method must use specific operations to achieve each of the above goals. Usually, an emphasis on non-dominated solutions is performed to achieve the first goal and a niching operation is performed to achieve the second goal. In addition, an elite-preserving operation is used to speed up convergence.

Again, we assume that the user is familiar with a particular population-based evolutionary algorithm, in which in each generation one or more new offspring are created by means of recombination and mutation operators. We describe here a generic archive-based EMO strategy.

Step 1 A population P and an empty archive A are initialized. The non-dominated solutions of P are copied in A . Steps 2 and 3 are iterated till a termination criterion is satisfied.

Step 2 A set of λ new offspring solutions are created using P and A .

Step 3 Every new offspring solution is checked for its inclusion in A by using a archive-acceptance criterion C_A and for its inclusion in P by using a population-acceptance criterion C_P . If an offspring is not to be included to either P or A , it is deleted.

Step 4 Before termination, the non-dominated set of the archive A is declared as the efficient set.

In Step 2, random solutions from the combined set $P \cup A$ can be used to create an offspring solution or some solution from P and some other solutions from A can be used to create the offspring solution. Different archive-acceptance and population-acceptance criteria can be used. Here, we propose one criterion each. Readers can find another implementation elsewhere ([Deb et al. 2003b](#)).

15.7.2.1 Archive Acceptance Criterion $C_A(c, A)$

The archive A has a maximum size K , but at any iteration it may not have all K members. This criterion required domination check and a niching operator which computes the niching of the archive with respect to a particular solution. For example, the crowding distance metric for a solution i in a subpopulation (suggested in Sect. 15.4.3) measures the objective-wise distance between two neighboring solutions of solution i in the subpopulation. The larger the crowding distance, the less crowded is the solution and the higher is probability of its existence in the subpopulation.

The offspring c is accepted in the archive A if any of the following conditions is true:

1. Offspring c dominates any archive member A . In this case, delete those archive members and include c in A .
2. Offspring c is non-dominated with all archive members and the archive is not *full* (that is, $|A| < K$). In this case, c is simply added to A .
3. Offspring c is non-dominated with all archive members, the archive is full, and crowding distance of c is larger than that of an archive member. In this case, that archive member is deleted and c is included in A .

15.7.2.2 Population Acceptance Criterion $C_P(c, P)$

If the offspring is a good solution compared to the current archive, it will be included in A by the above criterion. The inclusion of the offspring in the population must be made mainly from the point of view of keeping diversity in the population. Any of the following criteria can be adopted to accept c in P :

1. Offspring c replaces the most old (in terms of its time of inclusion in the population) population member.
2. Offspring c replaces a random population member.
3. Offspring c replaces the least-used (as a parent) population member.
4. Offspring c introduces more diversity of the population compared to an existing population member. Here the crowding distance or an entropy-based metric can be used to compute the extent of diversity.
5. Offspring c replaces a population member *similar* (in terms of its phenotype or genotype) to itself.
6. Offspring c replaces a population member dominated by c .

It is worth the effort to investigate which of the above criteria works the best in standard test problems, but the maintenance of diversity in the population and search for widespread non-dominated solutions in the archive are two activities which should allow the combined algorithm to reach the true efficient frontier quickly and efficiently.

In the following, we suggest some important post-optimality studies which are equally important to the optimality study and are often ignored in EMO studies.

15.7.3 Post-optimality Studies

It should be well understood that an EMO method (no matter whether it is the above one or any of the existing ones) does not have a guaranteed convergence properties; nor do they have any guaranteed proof for finding a well-diversed set of solutions. Thus, there is an onus on the part of EMO researchers/practitioners to perform a number of post-optimality studies to ensure (or build confidence about) convergence and achievement of diversity in obtained solutions. Here, we make some suggestions.

1. Use a hybrid EMO-local search method. From each of the obtained EMO solution, perform a single-objective search by optimizing a combined objective function—see Chap. 9.6 in [Deb \(2001\)](#) for more details. This will cause the EMO solutions to reach near to the true efficient frontier.
2. Obtain individual optimum solutions and compare the obtained EMO solutions with the individual optima on a plot or by means of a table. Such a visual comparison will indicate the extent of convergence as well as the extent of diversity in the obtained solutions.

3. Perform a number of ε -constraint studies for different values of the ε -vector, given in Eq. (15.7) and obtain efficient solutions. Compare these solutions with the obtained EMO solutions to get further visual confirmation of the extent of convergence and diversity of obtained EMO solutions.
4. Finally, it is advisable to also plot the initial population on the same objective space showing the efficient solutions, as this will depict the extent of optimization performed by the EMO local search approach.

For such a post-optimality study, refer to any of the application studies performed by the author ([Deb and Jain 2003](#); [Deb et al. 2004b](#); [Deb and Tiwari 2004](#); [Deb et al. 2004a](#)).

For practical problems, we also suggest performing an *innovation* study after multiple trade-off solutions are found to reveal useful properties that are common to them. As shown in Sect. 15.6.2 and in other studies ([Deb and Srinivasan 2006](#); [Bandaru and Deb 2011b](#)), such a task elicits useful knowledge that is usually more valuable than just the discovery of a set of trade-off solutions.

15.7.4 Evaluating a Multi-objective Optimization Algorithm

When a new algorithm is suggested to find a set of Pareto-optimal solutions in a MOOP, the algorithm has to be evaluated by applying them on standard test problems and compared with existing state-of-the-art EMO algorithms applied to the identical test problems. Here, we suggest a few guidelines in this direction.

1. Test problems with 20–50 variables must be included in the test set.
2. Test problems with three or more objectives must be included in the test set. For scalable test problems, readers may refer to WFG ([Huband et al. 2005](#)) and DTLZ ([Deb et al. 2005](#)) test problems.
3. Test problems must include some non-linear constraints, making some portion of the unconstrained Pareto-optimal front infeasible. For a set of constrained test problems, see the CTP problems ([Deb 2001](#)) or DTLZ test problems.
4. Standard EMO algorithms such as NSGA-II, SPEA2, PESA, ε -MOEA, and others must have to be used for comparison purposes. See the section *Sources of Additional Information* for some freely downloadable codes of these algorithms.
5. A proper criterion for the comparison must be chosen. Often, the algorithms are compared based on the fixed number of evaluations. They can also be compared based on some other criterion ([Deb 2001](#)).
6. Besides static performance metrics which are applied to the final solution set, *running metrics* ([Deb and Jain 2002](#)) may also be computed, plotted with generation number, and compared among two algorithms. The running metrics provide a dynamic (generation-wise) evaluation of the algorithm, rather than what had happened at the end of a simulation run.

15.8 Research Challenges

With the growing interest in the field of multi-objective optimization, particularly using evolutionary algorithms, there exist a number of research directions:

EMO and decision making: EMO methodologies have amply shown that multiple and well-spread Pareto-optimal solutions can be obtained in a single simulation run for a few objectives (four or less). We discuss solving *many-objective* problems later in this section. However, finding a set of trade-off solutions is only a part of the whole story. In practice, one needs to choose only a single preferred solution. Such a task requires one to use a multiple-criterion decision-making (MCDM) technique (Miettinen 1999; Belton and Stewart 2002; Chankong and Haimes 1983; Collette and Siarry 2004; Tzeng and Huang 2011). The combination of EMO and MCDM can, in principle, be used in three possible ways:

1. *A priori* approach, in which preference information can be determined before any optimization task is performed, like in scalarization methods, such as weighted-sum, ϵ -constraint, and other methods (Chankong and Haimes 1983). As discussed above, determining a preference information without any knowledge of trade-off solutions becomes a difficult task. However, if a priori information is used to find a preferred region of the Pareto-optimal front, instead of a single preferred solution, MCDM techniques can be used with an EMO to find a predefined focussed region (Deb et al. 2006; Deb and Kumar 2007a,b).
2. *A posteriori* approach, in which a set of trade-off solutions is first found and then a MCDM technique is used to analyse the solutions to choose a single preferred solution (Deb 2001; Coverstone-Carroll et al. 2000). Although the decision-making becomes relatively meaningful when a set of trade-off solutions are available, the approach becomes difficult to apply in problems having five or more objectives, as finding a set of trade-off solutions for a large set of objectives is still a difficult task.
3. *Interactive* approach, in which preference information is used during the EMO process iteratively, so that the combined approach is directed towards the preferred part of the Pareto-optimal region. A couple of such hybrid methods have been suggested recently (Deb et al. 2010; Branke et al. 2009).

In one of the interactive studies (Deb et al. 2010), after every 10 or 20 generations, a few (four or five) clustered non-dominated solutions are presented to the decision-maker. Using MCDM techniques, the decision-maker then provides partial or complete preference information by performing pair-wise comparisons of these solutions. These information are then used to build a mathematical utility function honoring the decision-maker's preference information. For the next 10 or 20 generations the EMO modifies its domination principle with the developed utility function so as to focus its search towards the preferred part of the search space. These approaches are practical and promise to handle many-objective problems using a combined EMO and MCDM approach.

Handling many objectives: So far, most studies using EMO strategies have been restricted to two- or three-objective problems. In practice, there exist a considerable number of problems in which 10 or 15 objectives are commonplace. Existing domination-based EMO approaches have difficulties in solving such large-dimensional problems due to the following reasons:

- A large fraction of the population becomes non-dominated to each other for a large number of objectives, as there are many ways a solution can become non-dominated. In an EMO approach emphasizing all non-dominated solutions, such a method does not keep many population slots free for new solutions, thereby slowing the search.
- A diversity preservation procedure becomes computationally expensive to determine the extent of crowding of solutions.
- An exponentially large number of solutions are required to represent a large-dimensional Pareto-optimal front, thereby requiring an exponentially large population (or an archive) to store trade-off solutions.
- Comparison of two sets of trade-off solutions for set-based EMO approaches (Zitzler et al. 2010; Zitzler and Künzli 2004) becomes a difficult task for a large-dimensional problem. Estimation of set-based metrics, such as hypervolume, becomes computationally expensive.
- Although not specific to EMO algorithms, visualization of a large-dimensional dataset becomes difficult.

However, recent studies on many objective optimization problems (Zhang and Li 2007; Knowles and Corne 2007; López and Coello Coello 2009; Corne and Knowles 2007; Hughes 2005; Ishibuchi et al. 2008; Deb and Jain 2012) have shown that computationally tractable EMO algorithms can be developed by using fixed search directions or fixed reference points to handle large-dimensional problems.

Non-evolutionary multi-objective optimization: EMO methods include principles of genetic algorithms, evolution strategy, genetic programming, particle swarm optimization, differential evolution and others. But other non-traditional optimization techniques such as ant colony optimization, tabu search and simulated annealing can also be used for solving MOOPs. Although there has been some research and application in this direction (Hansen 1997; Khor et al. 2001; Balicki and Kitowski 2001; Bandyopadhyay et al. 2008; McMullen 2001; Gravel et al. 2002; Kumral 2003; Parks and Suppapitnarm 1999; Chattopadhyay and Seeley 1994), more rigorous studies are called for, and such techniques can also be suitably used to find multiple Pareto-optimal solutions.

Performance metrics: For M objectives, the Pareto-optimal region will correspond to at most an M -dimensional surface. To compare two or more algorithms, it is then necessary to compare M -dimensional data sets which are partially ordered. It is not possible to have only one performance metric to compare such multi-dimensional data sets in an unbiased manner. A study has shown that at least M performance metrics are necessary to properly compare such data sets (Zitzler et al. 2003). An alternative pragmatic suggestion was to compare the

data sets from a purely *functional* point of view of (i) measuring the extent of convergence to the front and (ii) measuring the extent of diversity in the obtained solutions (Deb 2001). It then becomes a challenge to develop performance metrics for both functional goals for problems having any number of objectives. The hypervolume metric (Zitzler and Thiele 1999) has received a lot of attention among the EMO researchers, due to its ability to provide a combined estimate of convergence and diversity of a set of solutions. However, the computation of hypervolume for more than three or four objective problems is time-consuming. Researchers have devised approximate procedures for estimating hypervolume in higher dimensions (Bradstreet et al. 2008; While et al. 2006; Bader et al. 2010).

Test problem design: When new algorithms are designed, they need to be evaluated on test problems for which the desired Pareto-optimal solutions are known. Moreover, the test problems must be such that they are controllable to test an algorithm's ability to overcome a particular problem difficulty. Although there exist a number of such test problems (Deb 2001; Deb et al. 2005), more such problems providing different kinds of difficulties must be developed. Care should be taken to make sure that the test problems are scalable to any number of objectives and decision variables, so that systematic evaluation of an algorithm can be performed. Recent test problems (Zhang et al. 2008; Huband et al. 2005) are some efforts in this direction.

Parallel EMO methodologies: With the availability of parallel or distributed processors, it may be wise to find the complete Pareto-optimal front in a distributed manner. A study (Deb et al. 2003a) has suggested such a procedure based on a guided-domination concept, in which one processor focuses on finding only a portion of the Pareto-optimal front. With intermediate cross-talks between the processors, the procedure has shown that the complete Pareto-optimal front can be discovered by concatenating the solutions from a number of processors. Since each processor works on a particular region in the search space and processors communicate between themselves, a faster and parallel search is expected from such an implementation. Other similar parallelization techniques must be attempted and evaluated. Parallel EMO algorithms are also required to be developed for GPU-based computing platforms to take advantage of the recent enhancement of GPU technology. Some efforts in this direction are by Sharma and Collet (2010) and Wong (2009).

Multi-objectivization using EMO principle: Over the past few years and since the development of EMO methodologies, they have been also used to help solve a number of other optimization problems, such as (i) in reducing bloating problems commonly found in genetic programming applications (Bleuler et al. 2001), (ii) in goal programming problems (Deb 1999), (iii) in maintaining diversity in a single-objective EA (Jensen 2003a), (iv) single-objective constraint-handling problems (Coello 2000; Surry et al. 1995), (v) solving constrained optimization problems (Deb and Datta 2010), (vi) solving multimodal problems (Deb and Saha 2012), and others. Because of the use of additional objectives signifying a desired effect to be achieved, the search procedure becomes more flexible. More such problems, which reportedly perform poorly due to some fixed or rigid

solution procedures, must be tried using a multi-objective approach. A recent edited book ([Knowles et al. 2008](#)) presents many such recently proposed multi-objectivization studies.

EMO for redundant objectives: Many practical problems may have a large number of objectives, but the Pareto-optimal front of the problem may be lower-dimensional. In such problems, certain objectives get correlated to each other as the solutions approach the Pareto-optimal front. In such methods, the redundancy in objectives are determined by various means—through a principal component analysis (PCA) ([Deb and Saxena 2006](#)), nonlinear PCA analysis ([Saxena et al. 2013](#)), and other means ([Brockhoff and Zitzler 2006, 2007](#))—as the algorithms progress and the redundant objectives are eliminated to reduce the cardinality of the objectives. In problems having 50 objectives, the PCA-based NSGA-II procedure ([Deb and Saxena 2006](#)) was shown to reduce 48 correlated objectives successively with generations. Further such studies with better computational efficiency are needed for application to real-world problems.

Theoretical developments: One aspect for which EMO can be criticized is the lukewarm interest among its researchers to practice much theory related to their working principles or convergence behaviors. Apart from a few studies ([Rudolph 1998; Rudolph and Agapie 2000](#)), this area still remains a fertile field for theoretically oriented researchers to dive into and suggest algorithms with a good theoretical basis. Algorithms with a time complexity analysis on certain problems have been started ([Giel 2003; Laumanns et al. 2002b, 2004](#)) and research in this area should grow more popular in trying to devise problem–algorithm combinations with an estimated computational time for finding the complete Pareto-optimal set.

EMO on dynamic problems: Dynamic optimization involves objectives, constraints or problem parameters which change over time ([Branke 2001](#)). This means that as an algorithm is approaching the optimum of the current problem, the problem definition changes and now the algorithm must solve a new problem. Often, in such dynamic optimization problems, an algorithm is usually not expected to find the optimum, instead it is expected to track the changing optimum with iteration. A study ([Deb et al. 2007](#)) proposed the following procedure for dynamic optimization involving single or multiple objectives. Let $\mathcal{P}(t)$ be a problem which changes with time t (from $t = 0$ to $t = T$). Despite the continual change in the problem, we assume that the problem is fixed for a time period τ , which is not known a priori and the aim of the (offline) dynamic optimization study is to identify a suitable value of τ for an accurate as well as computationally faster approach. For this purpose, an optimization algorithm with τ as a fixed time period is run from $t = 0$ to $t = T$ with the problem assumed fixed for every τ time period. A measure $\Gamma(\tau)$ determines the performance of the algorithm and is compared with a pre-specified and expected value Γ_L . If $\Gamma(\tau) \geq \Gamma_L$, for the entire time domain of the execution of the procedure, we declare τ to be a permissible length of stasis. Then, we try with a reduced value of τ and check if a smaller length of stasis is also acceptable. If not, we increase τ to allow the optimization problem to remain static for a longer time so that the chosen algorithm can now

have more iterations (time) to perform better. Such a procedure will eventually come up with a time period τ^* which would be the smallest time of stats allowed for the optimization algorithm to work based on the chosen performance requirement. Based on this study, a number of test problems and a hydro-thermal power dispatch problem have been tackled (Deb et al. 2007).

In the case of dynamic multi-objective problem-solving tasks, there is an additional difficulty which is worth mentioning here. Not only does an EMO algorithm needs to find or track the changing Pareto-optimal fronts, but in a real-world implementation, it must also make an immediate decision about which solution to implement from the current front before the problem changes to a new one. Decision-making analysis is considered to be time-consuming, involving execution of analysis tools, higher-level considerations, and sometimes group discussions. If dynamic EMO is to be applied in practice, *automated* procedures for making decisions must be developed. Although it is not clear how to generalize such an automated decision-making procedure in different problems, problem-specific tools are possible and certainly a worthwhile and fertile area for research.

Real-world applications: Although the usefulness of EMO and classical multi-objective optimization methods are increasingly being demonstrated by solving real-world problems (Coello and Lamont 2004), more complex and innovative applications would not only demonstrate the widespread applicability of these methods but also may open up new directions for research.

15.9 Conclusions

For the past two decades, the usual practice of treating MOOPs by scalarizing them into a single objective and optimizing it has been seriously questioned. The presence of multiple objectives results in a number of Pareto-optimal solutions, instead of a single optimum solution. In this tutorial, we have discussed the use of an ideal multi-objective optimization procedure which attempts to find a well-distributed set of Pareto-optimal solutions first. It has been argued that choosing a particular solution as a post-optimal event is a more convenient and pragmatic approach than finding an optimal solution for a particular weighted function of the objectives. Besides introducing the multi-objective optimization concepts, this tutorial also has also presented two commonly used MOEAs.

Besides finding the multiple Pareto-optimal solutions, the suggested ideal multi-objective optimization procedure has another unique advantage. Once a set of Pareto-optimal solutions are found, they can be analyzed. The principle behind the *transition* from the optimum of one objective to that of other objectives can be investigated as a post-optimality analysis. Since all such solutions are optimum with respect to certain trade-off between objectives, the transition should reveal interesting knowledge on an optimal process of sacrifice of one objective to get a gain in other objectives.

The field of MOEAs has now matured. Nevertheless, there exist a number of interesting and important research topics which must be investigated before their full potential is unearthed. This tutorial has suggested a number of salient research topics to motivate newcomers to pay further attention to this growing field of importance.

Sources of Additional Information

Here, we outline some dedicated literature in the area of evolutionary multi-objective optimization and decision-making.

Books in Print

- C. A. C. Coello, D. A. VanVeldhuizen, and G. Lamont (2002). *Evolutionary algorithms for solving multi-objective problems*. Kluwer, Boston—a good reference book with a good citation of most EMO studies up to 2001.
- A. Osyczka (2002). *Evolutionary algorithms for single and multicriteria design optimization*. Physica-Verlag, Heidelberg—a book describing single and multi-objective EAs with lots of engineering applications.
- K. Deb (2001). *Multi-objective optimization using evolutionary algorithms*. Wiley, Chichester (2nd edn, with exercise problems)—a comprehensive book introducing the EMO field and describing major EMO methodologies and some research directions.
- K. Miettinen (1999). *Nonlinear multiobjective optimization*. Kluwer, Boston—a good book describing classical multi-objective optimization methods and a extensive discussion on interactive methods.
- M. Ehrgott (2000). *Multicriteria optimization*. Springer, Berlin—a good book on the theory of multi-objective optimization.

Conference Proceedings

The following six conference proceedings spanning from 2001 to 2013 are most useful on the theory, algorithms, and application of EMO.

- Purshouse et al., eds (2013). *Evolutionary Multi-Criterion Optimization (EMO-13) Conference Proceedings*, LNCS 7811. Springer, Berlin.
- Takahashi et al., eds (2011). *Evolutionary Multi-Criterion Optimization (EMO-11) Conference Proceedings*, LNCS 6576. Springer, Berlin.
- Ehrgott et al., eds (2009). *Evolutionary Multi-Criterion Optimization (EMO-09) Conference Proceedings*, LNCS 5467. Springer, Berlin.

- Obayashi et al., eds (2007). *Evolutionary Multi-Criterion Optimization (EMO-07) Conference Proceedings*, LNCS 4403. Springer, Berlin.
- Coello et al., eds (2005). *Evolutionary Multi-Criterion Optimization (EMO-05) Conference Proceedings*, LNCS 3410. Springer, Berlin.
- Fonseca et al., eds (2003). *Evolutionary Multi-Criterion Optimization (EMO-03) Conference Proceedings*. LNCS 2632. Springer, Berlin.
- Zitzler et al., eds (2001). *Evolutionary Multi-Criterion Optimization (EMO-01) Conference Proceedings*. LNCS 1993. Springer, Berlin.

Additionally,

- GECCO (Springer LNCS) and CEC (IEEE Press) annual conference proceedings feature numerous research papers on EMO theory, implementation, and applications.
- MCDM conference proceedings (Springer) publish theory, implementation, and application papers in the area of classical multi-objective optimization.

Mailing Lists

- emo-list@ualg.pt (EMO methodologies)
- <http://lists.jyu.fi/mailman/listinfo/mcdm-discussion> (MCDM related queries)

Public-Domain Source Codes

- NSGA-II in C: <http://www.iitk.ac.in/kangal/soft.htm>
- SPEA2 in C++: <http://www.tik.ee.ethz.ch/~zitzler>
- MOEA/D in C++: <http://dces.essex.ac.uk/staff/zhang/webofmoead.htm>
- Other codes: <http://www.lania.mx/~ccoello/EMOO/>
- MCDM softwares: <http://www.mit.jyu.fi/MCDM/soft.html>
- JMetal software: <http://jmetal.sourceforge.net/>

References

- Babu B, Jehan ML (2003) Differential evolution for multi-objective optimization. In: Proceedings of the CEC'2003, Canberra, vol 4. IEEE, Piscataway, pp 2696–2703
- Bader J, Deb K, Zitzler E (2010) Faster hypervolume-based search using Monte Carlo sampling. In: Proceedings of the MCDM 2008, Auckland. LNEMS 634. Springer, Heidelberg, pp 313–326

- Bagchi T (1999) Multiobjective scheduling by genetic algorithms. Kluwer, Boston
- Balicki J, Kitowski Z (2001) Multicriteria evolutionary algorithm with tabu search for task assignment. In: Proceedings of the EMO-01, Zurich, pp 373–384
- Bandaru S, Deb K (2010) Automated discovery of vital knowledge from pareto-optimal solutions: first results from engineering design. In: Proceedings of the WCCI-2010, Barcelona. IEEE, Piscataway
- Bandaru S, Deb K (2011a) Automated innovation for simultaneous discovery of multiple rules in bi-objective problems. In: Proceedings of the EMO-2011, Ouro Preto. Springer, Heidelberg, pp 1–15
- Bandaru S, Deb K (2011b) Towards automating the discovery of certain innovative design principles through a clustering based optimization technique. Eng Optim 43:911–941
- Bandyopadhyay S, Saha S, Maulik U, Deb K (2008) A simulated annealing-based multiobjective optimization algorithm: Amosa. IEEE Trans Evol Comput 12:269–283
- Belton V, Stewart TJ (2002) Multiple criteria decision analysis: an integrated approach. Kluwer, Boston
- Bleuler S, Brack M, Zitzler E (2001) Multiobjective genetic programming: reducing bloat using SPEA2. In: Proceedings of the CEC-2001, Seoul, pp 536–543
- Bradstreet L, While L, Barone L (2008) A fast incremental hypervolume algorithm. IEEE Trans Evol Comput 12:714–723
- Branke J (2001) Evolutionary optimization in dynamic environments. Springer, Heidelberg
- Branke J, Greco S, Slowinski R, Zielniewicz P (2009) Interactive evolutionary multiobjective optimization using robust ordinal regression. In: Proceedings of the EMO-09, Nantes. Springer, Berlin, pp 554–568
- Brockhoff D, Zitzler E (2006) Are all objectives necessary? On dimensionality reduction in evolutionary multiobjective optimization. In: PPSN IX, Reykjavik. LNCS 4193, pp 533–542
- Brockhoff D, Zitzler E (2007) Dimensionality reduction in multiobjective optimization: the minimum objective subset problem. In: Waldmann KH, Stocker UM (eds) OR proceedings 2006, Karlsruhe, Germany. Springer, Berlin, pp 423–429
- Chankong V, Haimes YY (1983) Multiobjective decision making theory and methodology. North-Holland, New York
- Chattopadhyay A, Seeley C (1994) A simulated annealing technique for multiobjective optimization of intelligent structures. Smart Mater Struct 3:98–106
- Coello CAC (2000) Treating objectives as constraints for single objective optimization. Eng Optim 32:275–308
- Coello CAC (2003) <http://www.lania.mx/~ccoello/EMOO/>
- Coello CAC, Lamont GB (2004) Applications of multi-objective evolutionary algorithms. World Scientific, Singapore
- Coello CAC, Lechuga MS (2002) MOPSO: a proposal for multiple objective particle swarm optimization. In: Proceedings of the CEC 2002, vol 2. IEEE, Piscataway, Honolulu, USA, pp. 1051–1056

- Coello CAC, Toscano G (2000) A micro-genetic algorithm for multi-objective optimization. Technical report Lania-RI-2000-06, Laboratoria Nacional de Informatica Avanzada, Xalapa, Veracruz
- Coello CAC, Van Veldhuizen DA, Lamont G (2002) Evolutionary algorithms for solving multi-objective problems. Kluwer, Boston
- Coello CAC, Aguirre AH, Zitzler E (eds) (2005) Evolutionary multi-criterion optimization (EMO-2005). LNCS 3410. Springer, Berlin
- Collette Y, Siarry P (2004) Multiobjective optimization: principles and case studies. Springer, Berlin
- Cormen TH, Leiserson CE, Rivest RL (1990) Introduction to algorithms. Prentice-Hall, New Delhi
- Corne DW, Knowles JD (2007) Techniques for highly multiobjective optimization: some nondominated points are better than others. In: Proceedings of the GECCO-07, London. ACM, New York, pp 773–780
- Corne DW, Knowles JD, Oates M (2000) The Pareto envelope-based selection algorithm for multiobjective optimization. In: Proceedings of the PPSN-VI, Paris, pp 839–848
- Coverstone-Carroll V, Hartmann JW, Mason WJ (2000) Optimal multi-objective low-thrust spacecraft trajectories. Comput Methods Appl Mech Eng 186:387–402
- Deb K (1995) Optimization for engineering design: algorithms and examples. Prentice-Hall, New Delhi
- Deb K (1999) Solving goal programming problems using multi-objective genetic algorithms. In: Proceedings of the CEC, Washington, pp 77–84
- Deb K (2001) Multi-objective optimization using evolutionary algorithms. Wiley, Chichester
- Deb K (2003) Unveiling innovative design principles by means of multiple conflicting objectives. Eng Optim 35:445–470
- Deb K, Datta R (2010) A fast and accurate solution of constrained optimization problems using a hybrid bi-objective and penalty function approach. In: Proceedings of the IEEE WCCI 2010, Barcelona, pp 165–172
- Deb K, Jain S (2002) Running performance metrics for evolutionary multi-objective optimization. In: Proceedings of the 4th Asia-Pacific conference on simulated evolution and learning (SEAL-02), Singapore, pp 13–20
- Deb K, Jain S (2003) Multi-speed gearbox design using multi-objective evolutionary algorithms. ASME Trans Mech Des 125:609–619
- Deb K, Jain H (2012) Handling many-objective problems using an improved NSGA-II procedure. In: Proceedings of the CEC 2012, Brisbane
- Deb K, Kumar A (2007a) Interactive evolutionary multi-objective optimization and decision-making using reference direction method. In: Proceedings of the GECCO 2007, London. ACM, New York, pp 781–788
- Deb K, Kumar A (2007b) Light beam search based multi-objective optimization using evolutionary algorithms. In: Proceedings of the CEC-07, Singapore, pp 2125–2132
- Deb K, Saha A (2012) Multimodal optimization using a bi-objective evolutionary algorithms. Evol Comput J 20:27–62

- Deb K, Saxena D (2006) Searching for Pareto-optimal solutions through dimensionality reduction for certain large-dimensional multi-objective optimization problems. In: Proceedings of the WCCI 2006, Vancouver, pp 3352–3360
- Deb K, Srinivasan A (2006) Innovization: innovating design principles through optimization. In: Proceedings of the GECCO-2006, Seattle. ACM, New York, pp 1629–1636
- Deb K, Tiwari S (2004) Multi-objective optimization of a leg mechanism using genetic algorithms. Technical report KanGAL 2004005, Kanpur Genetic Algorithms Laboratory (KanGAL), IIT, Kanpur
- Deb K, Agrawal S, Pratap A, Meyarivan T (2002) A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput* 6:182–197
- Deb K, Zope P, Jain A (2003a) Distributed computing of pareto-optimal solutions using multi-objective evolutionary algorithms. In: Proceedings of the EMO-03, Faro. LNCS 2632, pp 535–549
- Deb K, Mohan M, Mishra S (2003b) Towards a quick computation of well-spread pareto-optimal solutions. In: Proceedings of the EMO-03, Faro. LNCS 2632, pp 222–236
- Deb K, Jain P, Gupta N, Maji H (2004a) Multi-objective placement of electronic components using evolutionary algorithms. *IEEE Trans Compon Packag Technol* 27:480–492
- Deb K, Mitra K, Dewri R, Majumdar S (2004b) Towards a better understanding of the epoxy polymerization process using multi-objective evolutionary computation. *Chem Eng Sci* 59:4261–4277
- Deb K, Thiele L, Laumanns M, Zitzler E (2005) Scalable test problems for evolutionary multi-objective optimization. In: Abraham A et al (eds) *Evolutionary multiobjective optimization*. Springer, London, pp 105–145
- Deb K, Sundar J, Uday N, Chaudhuri S (2006) Reference point based multi-objective optimization using evolutionary algorithms. *Int J Comput Intell Res (IJCIR)* 2:273–286
- Deb K, Rao UB, Karthik S (2007) Dynamic multi-objective optimization and decision-making using modified NSGA-II: a case study on hydro-thermal power scheduling bi-objective optimization problems. In: Proceedings of the EMO-2007, Matsushima
- Deb K, Sinha A, Korhonen P, Wallenius J (2010) An interactive evolutionary multi-objective optimization method based on progressively approximated value functions. *IEEE Trans Evol Comput* 14:723–739
- Ehrhart M (2000) *Multicriteria optimization*. Springer, Berlin
- Ehrhart M, Fonseca CM, Gandibleux X, Hao JK, Sevaux M (eds) (2009) Proceedings of the EMO-2009, Nantes. LNCS 5467. Springer, Heidelberg
- Fonseca C, Fleming P, Zitzler E, Deb K, Thiele L (eds) (2003) *Proceedings of the EMO-2003*, Faro. LNCS 2632. Springer, Heidelberg
- Giel O (2003) Expected runtimes of a simple multi-objective evolutionary algorithm. In: Proceedings of the CEC-2003, Canberra. IEEE, Piscataway, pp 1918–1925

- Goh CK, Tan KC (2009) Evolutionary multi-objective optimization in uncertain environments: issues and algorithms. Springer, Berlin
- Goldberg DE (1989) Genetic algorithms for search, optimization, and machine learning. Addison-Wesley, Reading
- Gravel M, Price WL, Gagné C (2002) Scheduling continuous casting of aluminum using a multiple objective ant colony optimization metaheuristic. *Eur J Oper Res* 143:218–229
- Haines YY, Lasdon LS, Wismer DA (1971) On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Trans Syst Man Cybern* 1:296–297
- Hansen MP (1997) Tabu search in multiobjective optimization: MOTS. Paper presented at MCDM'97, University of Cape Town
- Huband S, Barone L, While L, Hingston P (2005) A scalable multi-objective test problem toolkit. In: Proceedings of the EMO-2005, Guanajuato. Springer, Berlin
- Hughes EJ (2005) Evolutionary many-objective optimization: many once or one many? In: Proceedings of the CEC-2005, Edinburgh, pp 222–227
- Ishibuchi H, Tsukamoto N, Nojima Y (2008) Evolutionary many-objective optimization: a short review. In: Proceedings of the CEC-2008, Hong Kong, pp 2424–2431
- Jensen MT (2003a) Guiding single-objective optimization using multi-objective methods. In: Raidl G et al (eds) Applications of evolutionary computing. EvoWorkshops 2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, and EvoSTIM, Essex. LNCS 2611. Springer, Berlin, pp 199–210
- Jensen MT (2003b) Reducing the run-time complexity of multiobjective EAs. *IEEE Trans Evol Comput* 7:503–515
- Khor EF, Tan KC, Lee TH (2001) Tabu-based exploratory evolutionary algorithm for effective multi-objective optimization. In: Proceedings of the EMO-01, Zurich, pp 344–358
- Knowles JD, Corne DW (2000) Approximating the non-dominated front using the Pareto archived evolution strategy. *Evol Comput J* 8:149–172
- Knowles J, Corne D (2007) Quantifying the effects of objective space dimension in evolutionary multiobjective optimization. In: Proceedings of the EMO-2007, Matsushima. LNCS 4403, pp 757–771
- Knowles JD, Corne DW, Deb K (2008) Multiobjective problem solving from nature. Natural computing series. Springer, Berlin
- Kumral M (2003) Application of chance-constrained programming based on multi-objective simulated annealing to solve a mineral blending problem. *Eng Optim* 35:661–673
- Kung HT, Luccio F, Preparata FP (1975) On finding the maxima of a set of vectors. *J Assoc Comput Mach* 22:469–476
- Laumanns M, Thiele L, Deb K, Zitzler E (2002a) Combining convergence and diversity in evolutionary multi-objective optimization. *Evol Comput* 10:263–282
- Laumanns M, Thiele L, Zitzler E, Welzl E, Deb K (2002b) Running time analysis of multi-objective evolutionary algorithms on a simple discrete optimization problem. In: Proceedings of the PPSN-VII, Granada, pp 44–53

- Laumanns M, Thiele L, Zitzler E (2004) Running time analysis of multiobjective evolutionary algorithms on pseudo-Boolean functions. *IEEE Trans Evol Comput* 8:170–182
- López JA, Coello Coello CA (2009) Some techniques to deal with many-objective problems. In: Proceedings of the 11th annual conference companion on genetic and evolutionary computation, Montreal. ACM, New York, pp 2693–2696
- Loughlin DH, Ranjithan S (1997) The neighborhood constraint method: a multiobjective optimization technique. In: Proceedings of the 7th international conference on genetic algorithms, East Lansing, pp 666–673
- McMullen PR (2001) An ant colony optimization approach to addressing a JIT sequencing problem with multiple objectives. *Artif Intell Eng* 15:309–317
- Miettinen K (1999) Nonlinear multiobjective optimization. Kluwer, Boston
- Mostaghim S, Teich J (2003) Strategies for finding good local guides in multi-objective particle swarm optimization (MOPSO). In: Proceedings of the 2003 IEEE symposium on swarm intelligence, Indianapolis. IEEE, Piscataway, pp 26–33
- Obayashi S, Deb K, Poloni C, Hiroyasu T, Murata T (eds) (2007) Proceedings of the EMO-2007, Matsushima. LNCS 4403. Springer, Berlin
- Osyczka A (2002) Evolutionary algorithms for single and multicriteria design optimization. Physica-Verlag, Heidelberg
- Parks G, Suppapitnarm A (1999) Multiobjective optimization of PWR reload core designs using simulated annealing. In: Aragones JM (eds) Mathematics and computation, reactor physics and environmental analysis in nuclear applications, vol 2. Senda Editorial, Madrid, pp 1435–1444
- Rudolph G (1998) Evolutionary search for minimal elements in partially ordered finite sets. In: Proceedings of the 7th annual conference on evolutionary programming, San Diego. Springer, Berlin, pp 345–353
- Rudolph G, Agapie A (2000) Convergence properties of some multi-objective evolutionary algorithms. In: Proceedings of the CEC 2000, San Diego, pp 1010–1016
- Saxena D, Duro JA, Tiwari A, Deb K, Zhang Q (2013) Objective reduction in many-objective optimization: linear and nonlinear algorithms. *IEEE Trans Evol Comput* 17(1):77–99
- Sharma D, Collet P (2010) GPGPU compatible archive based stochastic ranking evolutionary algorithm (G-ASREA) for multi-objective optimization. In: Proceedings of the PPSN-2010, Kraków. Springer, Berlin, pp 111–120
- Srinivas N, Deb K (1994) Multi-objective function optimization using non-dominated sorting genetic algorithms. *Evol Comput J* 2:221–248
- Surry PD, Radcliffe NJ, Boyd ID (1995) A multi-objective approach to constrained optimization of gas supply networks: the COMOGA method. In: Evolutionary computing. AISB workshop, Sheffield. Springer, Berlin, pp 166–180
- Takahashi RHC, Deb K, Wanner EF, Greco S (2011) Proceedings of the EMO-2011, Ouro Preto. LNCS 6576. Springer, Berlin
- Tzeng GH, Huang J-J (2011) Multiple attribute decision making: methods and applications. CRC, Boca Raton

- Veldhuizen DV, Lamont GB (2000) Multiobjective evolutionary algorithms: analyzing the state-of-the-art. *Evol Comput J* 8:125–148
- While L, Hingston P, Barone L, Huband S (2006) A faster algorithm for calculating hypervolume. *IEEE Trans Evol Comput* 10:29–38
- Wong ML (2009) Parallel multi-objective evolutionary algorithms on graphics processing units. In: Proceedings of the GECCO-2009, Montreal, pp 2515–2522
- Zhang Q, Li H (2007) MOEA/D: a multiobjective evolutionary algorithm based on decomposition. *IEEE Trans Evol Comput* 11:712–731
- Zhang Q, Zhou A, Zhao SZ, Suganthan PN, Liu W, Tiwari S (2008) Multiobjective optimization test instances for the CEC-2009 special session and competition. Technical report, Nanyang Technological University, Singapore
- Zitzler E (1999) Evolutionary algorithms for multiobjective optimization: methods and applications. PhD thesis, Swiss Federal Institute of Technology ETH, Zürich
- Zitzler E, Künzli S (2004) Indicator-based selection in multiobjective search. In: Proceedings of the PPSN VIII, Birmingham. LNCS 3242. Springer, Berlin, pp 832–842
- Zitzler E, Thiele L (1998) An evolutionary algorithm for multiobjective optimization: the strength Pareto approach. Technical report 43, Computer Engineering and Networks Laboratory, Switzerland
- Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Trans Evol Comput* 3:257–271
- Zitzler E, Deb K, Thiele L, Coello CAC, Corne DW (2001a) Proceedings of the EMO-2001, Zurich. LNCS 1993. Springer, Berlin
- Zitzler E, Laumanns M, Thiele L (2001b) SPEA2: improving the strength Pareto evolutionary algorithm for multiobjective optimization. In: Giannakoglou KC, Tsahalis DT, Périaux J, Papailiou KD, Fogarty T (eds) Evolutionary methods for design optimization and control with applications to industrial problems, Athens. International Center for Numerical Methods in Engineering (CIMNE), pp 95–100
- Zitzler E, Thiele L, Laumanns M, Fonseca CM, da Fonseca VG (2003) Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Trans Evol Comput* 7:117–132
- Zitzler E, Thiele L, Bader J (2010) On set-based multiobjective optimization. *IEEE Trans Evol Comput* 14:58–79

Chapter 16

Sharpened and Focused No Free Lunch and Complexity Theory

Darrell Whitley

16.1 Introduction

This tutorial reviews basic concepts in complexity theory, as well as various *No Free Lunch* results and how these results relate to computational complexity. The tutorial explains basic concepts in an informal fashion that illuminates key concepts. “No Free Lunch” theorems for search can be summarized by the following result:

For all possible performance measures, no search algorithm is better than another when its performance is averaged over all possible discrete functions.

Note that No Free Lunch is often referred to simply as NFL within the heuristic search community (despite copyrights and trademarks held by the National Football League). Two more recent variants of NFL, the Sharpened NFL, and the Focused NFL are also reviewed. There has been a significant amount of confusion in the literature about the meaning of No Free Lunch, and differences between Sharpened NFL and Focused NFL have not been well understood in the literature. This tutorial attempts to resolve some of this confusion. The reader familiar with basic complexity theory might wish to skip to Sect. 16.3 on No Free Lunch. Section 16.4 explains new results based on the distinction between Sharpened and Focused No Free Lunch.

16.2 Complexity: P and NP

No Free Lunch relates to complexity theory in as much as complexity theory addresses the time and space costs of algorithms; complexity theory is also concerned with key classes of problems, such as the class of NP-complete problems that are also of interest to researchers designing search algorithms.

D. Whitley (✉)

Department of Computer Science, Colorado State University, Fort Collins, CO, USA

The complexity classes denoted by P and NP are the most famous classes of problems in complexity theory. The problem class P is the set of problems that can be solved in polynomial time on a deterministic Turing machine. For current purposes, we can think of any computer as a surrogate for a Turing machine (except that Turing machines are assumed to have infinite memory). The P stands for polynomial. In practice, we generally think of P as representing those problems that are *tractable*, i.e. problems that can be solved in reasonable computation time. What we generally mean when we say that a problem is not tractable is that the computational costs grow exponentially with problem size and that relatively modest sized problems can result in computation times involving years, or hundreds of years, or trillions of years.

The problem class NP is the set of problems that can be solved in polynomial time on a nondeterministic Turing machine. The NP stands for nondeterministic polynomial (which should *not* be confused with “not polynomial”). Nondeterminism is a bit strange. In a nondeterministic machine, choices are allowed in the computation, so that some things need not be computed. In effect, the computation itself becomes a search tree with paths and decision points in the computation. Each path in the tree corresponds to a computation that results in one possible solution, but only one path or a small number of paths yields an exact and optimal solution. We say that a problem is in NP if this search tree is polynomial in height, while the number of nodes in the search tree might be exponential. Thus, if we could explore all computational paths in parallel, we arrive at a solution in polynomial time. Alternatively, if we “magically” make the right choice at each decision node in the tree, then we again arrive at the desired solution in polynomial time. If we can *deterministically* find a path to a solution in polynomial time in every case, then the problem is in P. All problems in P are also in NP, because a nondeterministic Turing machine can do all the computations in polynomial time that can be done by a deterministic Turing machine in polynomial time.

Another hallmark of the class NP is that the correctness of solutions can be verified in *deterministic* polynomial time. Note that this is true, because if we have the solution in hand, we then know how to make the right choice at each decision node without needing any magical guidance.

Can all the problems that are solved by a nondeterministic Turing machine in polynomial time be solved by a deterministic Turing machine in polynomial time using another, more clever algorithm? What we are really asking is whether the complexity class P = NP. The answer is unknown and is considered to be one of the most important theoretical questions in Computer Science. It is an equally important question in Operations Research. While the answer is unknown, it is widely thought that P \neq NP.

Researchers have identified a very important subset of the class NP known as the class NP-complete. A problem class, R , is NP-complete if (1) it is NP-hard and (2) $R \in NP$. Informally, a problem is NP-hard if it is *at least* as hard as any other problem in NP. More formally, a problem R is NP-hard if there exists an NP-complete problem R_0 such that every instance of R_0 can be reformulated into an

instance of R in deterministic polynomial time. More formally, it is said that R_0 reduces to R in polynomial time. Therefore R must be just as hard as R_0 since R includes R_0 in some sense.

In a famous theorem, Cook (1971) established that Boolean satisfiability is NP-complete by showing it is in NP and by showing that *every* problem in NP can be expressed as a Boolean satisfiability problem (also just called SAT). Of course SAT is a member of the set of NP problems: the nondeterministic Turing machine just selects the right assignment to the Boolean variables to make the expression true, if it is possible to do so.

Other problems in NP have been shown to be NP-complete by showing that every SAT problem can be converted into an instance of that particular problem class. Thus, every instance of SAT can be converted into an instance of the 3-CNF-SAT problem. A 3-CNF-SAT problem is a satisfiability problem where the Boolean expressions is made up of conjunctive normal form clauses. Each clause contains three Boolean literals, where a literal is a variable or its negation, such as x_1 or $\neg x_1$. All of the clauses of a 3-CNF-SAT problem must be satisfied for the Boolean expression to be satisfied. Every instance of a 3-CNF-SAT, in turn, can be converted into an instance of a Hamiltonian circuit problem, and every Hamiltonian circuit can be converted into an instance of the traveling salesman problem (TSP) (Cormen et al. 1990). This means all of these problems are NP-hard. Showing that they are all also in the class NP makes them NP-complete. Technically, to be NP-complete, a problem must be a decision problem. A decision problem is a problem that has a yes or no answer. Therefore, the TSP is NP-complete when expressed as a decision problem (i.e. is there a tour with length $\leq k$?), but the TSP is still said to be NP-hard when expressed as an optimization problem.

Given the interrelated nature of the NP-complete problems, if researchers ever discover a polynomial-time algorithm for any NP-complete problem, then it would follow that *every* problem in NP could be solved in polynomial time. In an abstract sense, this means that all problems in the NP-complete are all of comparable difficulty, and that the NP-complete problems are (within a polynomial difference) the most difficult problems in the set made up of all problems in NP.

16.2.1 Complexity, Search and Optimization

Since we don't know how to compute the exact solution to NP-hard problems in polynomial time, we sometimes have to settle for approximate solutions. In some cases approximate methods can guarantee a solution which is within some ratio constant of the optimal solution; in other cases this is not possible and we use heuristics search methods to find the best solutions possible. It can be useful to think of these search methods as exploring the decision tree that is magically navigated by a nondeterministic Turing machine. The solutions that are found using heuristic search methods often are not optimal, but finding satisfactory or sufficiently good solutions can be important for many applications.

A basic distinction can be made between search problems that are discrete versus problems that are continuous. This distinction can also be related to the difference between integers and real-valued numbers. If we ask how many integers there are in the (inclusive) interval between 1 and 10, the answer is obviously 10 different and discrete values. But if we asked how many real-valued numbers there are between 1 and 10, the answer is infinitely many.

The nondeterministic Turing machine is clearly solving a discrete problem, because there are a fixed number of decisions that must be made to reach an optimal solution. By definition, the number of decisions that must be made by the Nondeterministic Turing Machine must be polynomial if it is solving an NP-hard problem.

Some problems cannot be solved in polynomial time by a nondeterministic Turing machines and therefore are not in NP; we can loosely think of such problems as requiring exponential time, although in complexity theory one must worry about both space (memory) and time and balance trade-offs between space and time costs.

Consider a parameter optimization problem such that there is a function f that takes k parameters as inputs and returns a single value that evaluates the usefulness or goodness of those k parameters. The space of possible inputs is known as the *domain* and the space of possible outputs as *co-domain* of the function. For example, we might have a parameter optimization problem that used temperature and pressure as two input control parameters for a process that produces some material (e.g. paper), where the output of the function might be the cost of the material, or some measurement of its quality.

If a parameter can be assigned any continuous real-valued number, then the input space is theoretically infinite. We will limit our attention to problems that are discrete such that the domain and therefore the co-domain are finite. Discrete parameter optimization problems are part of a larger set of discrete problems referred to as combinatorial optimization problems. Combinatorial optimization problems include many different types of problems, such as scheduling and resource allocation, as well as problems in graph theory and Boolean logic.

For example, we might have a scheduling problem where we want to optimize the order in which tasks are carried out. The goal might be to minimize total processing time, or to maximize work done per unit of time. For N tasks, there could be $N!$ ways to order those tasks. Or, we might want to assign truth values (0 or 1) to a Boolean expression, in which case there are 2^k assignments if there are k Boolean variables in the expression. In the first case, an input could be a permutation of tasks of length N and the evaluation might be how long it takes to process all of the N tasks. In the second case, an input might be a bit-string of length k representing the assignments made to the k Boolean variables, and the output might be a true or false (0 or 1) evaluation of the overall Boolean expression. For classic NP-hard problems, the search space is typically modeled in a general way so that the search space is exponentially large in relationship to the size of an input.

Parameter optimization problems can also be discretized. For example, a single input parameter can be restricted to a value between 1 and 100 (inclusive) where we only consider values that are increments of 0.01. In this case, there are only 10,000 possible assignments for that particular input parameter. If all of the parameters of a

parameter optimization problem are discretized in this way, then the overall search problem is discrete as well. There are a number of reasons that one might want to look at parameter optimization problems as discrete search spaces. In some cases, sensors for the inputs and/or outputs have limited precision and it does not make sense to represent and reason about extremely high precision numbers—we simply can't measure the world that precisely. And, in general, as soon as anything is represented in a computer program it is discrete. Infinite precision is a fiction, although it is often an extremely useful fiction (for example, when we can still use mathematical methods that exploit properties of continuous functions). But as soon as we decide to represent a parameter using a fixed-length floating point representation, the optimization problem is actually discrete.

This leads to the following observation. If the set of possible inputs is discrete, we can enumerate the set of inputs and label each possible input with a unique integer. We will also sort the inputs in some principled manner, so that the i th possible input is uniquely identified. This is a familiar concept in complexity, since it allows us to count all of the inputs. Thus, any particular instance of a discrete search problem using any given discrete representation can be abstractly modeled by a function

$$f(i) = j$$

where i is an integer that labels the i th input (i.e. the i element of the domain) and j is a member of the set of values that make up the co-domain. This perspective also provides a general foundation for discussing the concept of No Free Lunch.

16.3 No Free Lunch

In 1995, a paper by David Wolpert and William Macready caused a good deal of excitement in the search community. Their technical report *No Free Lunch Theorems for Search* presents proofs that can be summarized by the following No Free Lunch result:

For all possible performance measures, no search algorithm is better than another when its performance is averaged over all possible discrete functions.

First, note that we only consider discrete functions. A performance measure includes any measurement of the quality of the solution (or set of solutions) found after sampling some fixed number of points in the search space, or how long it takes to find a solution of a particular quality. It is also implied that a performance measure is taken over the set of domain and associated co-domain values that have been sampled so far.

An updated version of the original report appeared in 1997. A key assumption behind this result is that resampling is ignored: this means that if a search algorithm samples point i and evaluates the objective function $f(i)$ then that point is never sampled again. In reality, heuristic search algorithms concentrate search in particular regions of the search space: in other words, in a concentrated search more time is

spent sampling points that are near to previously observed good solutions. This is sometimes described as intensification or exploitation. Consequently, a concentrated search is one that is more likely to also resample previously visited points. Search algorithms that are more likely to resample points in the search space than others are in some sense worse than algorithms that resample less.

One of the most basic and least intelligence forms of search is random enumeration. Random enumeration means that we sample the search space randomly without replacement; this can be done using clever bookkeeping, or simply by keeping a list of visited points so that none are evaluated again. In practice, random sampling explores only a limited amount of the search space, and it is reasonable to allow sampling with replacement because resampling is unlikely if the sampling is random. When random sampling is used as a search algorithm, it provides a minimal baseline against which the performance of heuristic search algorithms can be judged. Clearly, we would expect any useful heuristic search algorithm to outperform random enumeration. However, a startling and powerful consequence of No Free Lunch is that *no* heuristic search algorithm is better than random enumeration when compared over all possible discrete functions.

Useful search algorithms do not exhaustively enumerate the entire search space. [Wolpert and Macready \(1995, 1997\)](#) model a search algorithm as a procedure that searches for m steps. However, this does not restrict any of the No Free Lunch results.

Another issue relating to No Free Lunch involves deterministic versus stochastic search algorithms. Some algorithms make deterministic decisions, such as a steepest-ascent local search algorithm: when started from the same point, steepest ascent always yields the same solution. Other search algorithms are stochastic—meaning that the search utilizes random numbers and makes stochastic decisions and therefore different runs will typically produce different solutions. Wolpert and Macready present arguments showing that the No Free Lunch theorems hold for both stochastic and deterministic search algorithms. [Radcliffe and Surry \(1995\)](#) also point out that in practice stochastic algorithms typically employ pseudo-random number generators. Thus, if we include the random number generator and initial seed in the specification of the search algorithm, then these stochastic algorithms, in effect, are also deterministic.

Immediately following its introduction, researchers had two general reactions to the No Free Lunch results.

- *Reaction 1.* Many researchers simply dismissed No Free Lunch, arguing that results concerning the set of all possible discrete functions are not applicable in the real world because this set is not representative of real-world problems. Some researchers pointed out that the set of all possible discrete functions is infinitely large and most functions are *incompressible* in that there is not a representation whose size is significantly less than the size of the function when fully enumerated. For example, if there are N values in the co-domain of a function, then writing down all of these values requires $N \log_2(N)$ bits (i.e. N values, $\log_2(N)$ bits per value). In effect, this representation of the function is just a look-up table where the i th entry is the co-domain value associated with $f(i)$. If there exists

no representation of a function that uses less than $O(N \log_2(N))$ bits, then that function is incompressible. Even if an evaluation function only returns 0 or 1, it still requires $O(N)$ bits to construct a look-up table or to enumerate the function; in this case, the look-up table is still exponentially large when N is exponentially large in relationship to the size of an input string to the evaluation function.

Of course, there are more random functions than non-random functions ([English 2000a](#)). Furthermore, most standard textbooks on computability discuss the well-known result that the set of all possible functions is uncountably infinite (as can be shown using diagonalization arguments), while the set of all possible programs (which are just bit-strings at the lowest level) is only countably infinite ([Sudcamp 1997](#)). So the set of all possible cost functions that can be implemented on a computer is a tiny subset of the set of all possible functions. Thus, the space of all possible discrete functions is largely composed of incompressible functions. Given these observations, “No Free Lunch is No Big Deal” seemed to be the conclusion of this point of view.

- *Reaction 2.* The other reaction to No Free Lunch was to acknowledge that researchers trying to develop the best possible algorithm for a particular application typically need to leverage extensive problem-specific knowledge. Consequently, the *No Free Lunch* result seemed to be an intuitive affirmation of the idea that there are no general-purpose search methods (at least none that are very effective) and that the business of developing search algorithms is one of building special-purpose methods to solve application-specific problems. This point of view echoes a refrain from the Artificial Intelligence community: “Knowledge is Power.”

Of course, there is truth in both of these views. It has taken several years for the research community to gain a deeper understanding of No Free Lunch. These investigations have led to some surprising and even fruitful results along the way. In [1998](#) Joe Culberson published an algorithmic view of No Free Lunch that added perspective to the debate; Culberson makes two important points.

First, all of this looks at search as a blind process. This means that we are doing black-box optimization and the only information we have is the evaluation of particular points in the space. We do not have information about what a solution might look like or information about how the evaluation function is constructed that might allow us to search more intelligently. Blind search is extremely weak. Using an adversarial argument we can think of blind search as the process of asking an adversary to sample a point of some objective function and then return an answer. In the space of all possible discrete functions, however, the adversary is free to return any value whatsoever without regard to those values of the search space that have already been examined. In the worst case, the previously sampled points from the search space tell us nothing about the remaining points in the search space. (As we will see later in this paper, for Focused No Free Lunch, we are not restricted to blind black-box optimization.)

Second, Culberson points out that search is often not blind. If we construct an algorithm for the TSP, for example, we usually exploit application-specific operators

and representations. But we do not completely give up generality; our algorithms are designed to solve a particular problem, but should be general enough to solve different instances of that problem.

[Radcliffe and Surry \(1995\)](#) first formalized the idea that we can also include representations under No Free Lunch. That is, when we consider all possible representations of a function, No Free Lunch still holds: no search algorithm is better than another when applied to all possible representations of a function. In effect, a representation just transforms one function into another.

Not surprisingly, No Free Lunch also holds when comparing the set of possible representations under Gray codes and binary bit encodings. However, Whitley and Rana (1997) pointed out that if one selected particular subsets of problems of bounded complexity, then No Free Lunch no longer holds; [Whitley \(1999\)](#) provided proofs of this for binary representations. [Droste et al. \(1999\)](#) also made similar observations, indicating that one can define sets of reasonable and interesting functions where one algorithm can consistently outperform another.

If we go back in time, No Free Lunch observations were made by Greg Rawlins at the *Foundations of Genetic Algorithms* (FOGA) workshops in 1990 and 1992. In the preface to the proceedings of the 1990 FOGA workshop [Rawlins \(1991\)](#) makes the following observations:

[I]t is sometimes suggested that GAs [Genetic Algorithms] are universal in that they can be used to optimize any function. These statements are true in only a very limited sense; any algorithm satisfying [these] claims can expect to do no better than random search over the space of all functions ([Rawlins 1991](#), p. 7).

It is now apparent that for a *fixed universal* algorithm, restricted to strings ... over the set of all possible domain functions ... it does not matter which encoding we use, since for every domain function which the encoding makes easier to solve there is another domain function that makes it more difficult to solve. Thus, changing the encoding does not affect the *expected difficulty* of solving a randomly chosen domain function.

Equivalently, assume that we have a *fixed* domain function f and suppose that we choose the encoding, e , at random. That is, we pick one of the ... possible encodings. Then, no search algorithm can expect to do better than random search, since no information is carried by e about f , except that for each string there is a value ([Rawlins 1991](#), p. 8).

Rawlins anticipated several of the consequences of No Free Lunch. Nevertheless, it was Wolpert and Macready who provided the first detailed proof of No Free Lunch for search.

16.3.1 No Free Lunch: Variations on a Theme

Two variants of NFL are as follows:

- The aggregate behavior of any two search algorithms is equivalent when compared over all possible discrete functions.

- The aggregate behavior of all possible search algorithms is equivalent when compared over any two discrete functions that share the same co-domain values.

At the root of these observations is another, more concise result. Consider any algorithm A_i applied to function f_j . Let $\text{Apply}(A_i, f_j, m)$ represent a *meta-level* algorithm that outputs the order in which A_i visits m elements in the co-domain of f_j after m steps. For every pair of algorithms A_k and A_i and for any function f_j , there exists another function f_l such that

$$\text{Apply}(A_i, f_j, m) \equiv \text{Apply}(A_k, f_l, m).$$

The equivalence operator \equiv denotes that the ordered sequence of co-domain values that is return by “Apply” will be equivalent. We could interpret this result in another way. For every pair of functions f_j and f_l that share the same co-domain values and for any algorithm A_i , there exists another algorithm A_k such that $\text{Apply}(A_i, f_j, m) \equiv \text{Apply}(A_k, f_l, m)$. In fact, if we consider the algorithms and the functions as variables that are supplied to the Apply function, then when any three of the “variables” are known, the fourth is immediately determined, assuming we restrict the functions to the same set of co-domain values.

This also implies that we can talk about No Free Lunch in a much smaller context: for example, we can talk about exactly two search algorithms applied to exactly two carefully chosen paired functions.

This perspective on No Free Lunch has some rather counterintuitive implications. Consider a *Best-First* version of steepest-ascent local search which restarts when a local optimum is encountered. Also consider a *Worst-First* steepest-ascent local search, also with restarts. We incorporate restarts so that these algorithms continue searching for an arbitrary number of steps. Then, for every function f_j there exists a function f_l such that

$$\text{Apply}(\text{Best-First}, f_j, m) \equiv \text{Apply}(\text{Worst-First}, f_l, m).$$

Virtually all researchers would accept that Best-First local search is a reasonable search algorithm and that it is useful on many real-world problems. In other words, there is a subset of problems where Best-First search is effective, relative to some performance measure. But there is a corresponding set of functions where Worst-First local search is equally effective. What do these functions look like? They probably are “structured” in some sense, and might be compressible. Also note that if we are minimizing a function, then a Worst-First local search is one that simply maximizes at each step, instead of minimizing. So on some functions, we find a good *minimal* solution by using an algorithm that maximizes. Why is Best-First search generally viewed as a reasonable algorithm and Worst-First as an unreasonable algorithm? This is a nagging question for which, at least formally, there are currently no good answers except that we expect functions representing real applications to have a structure that is better explored by Best-First search.

16.3.2 No Free Lunch and Permutation Closure

Whitley et al. (1997, 2000) first explored the idea that permutations could be used to represent both algorithms and functions—and thus produce an NFL result over a finite set. However, this idea is also implicit in the work of Radcliffe and Surry (1995) on NFL and representations.

Consider the following example. Assume that the co-domain of our objective function consists of the set of values $\{A, B, C\}$. Let the permutation $\langle A, B, C \rangle$ represent a canonical ordering of these values. We can start by considering bijective functions, those that are one-to-one and onto: an important implication of this is that each value in the co-domain is unique. To construct a function, we need to assign values to $f(1), f(2)$ and $f(3)$. Exactly $3!$ bijective functions can be constructed given three possible co-domain values. Additionally, only $3!$ behaviors are possible for any search algorithm, assuming that an algorithm does not resample points. Let an algorithm's behavior be represented by a permutation over the set of numbers $\{1, 2, 3\}$ which will serve as indices into the canonical permutation of co-domain values $\{A, B, C\}$. Let s_i be the i th value sampled by a search algorithm. Thus, the permutation $\langle 2, 1, 3 \rangle$ defined with respect to the canonical ordering $\langle A, B, C \rangle$ represents a search algorithm whose behavior can be described by the following sampling behavior

$$s_1 = f(2) = B, \quad s_2 = f(1) = A, \quad s_3 = f(3) = C.$$

Note that we don't actually need to specify a particular function to talk about behavior, we just need to define the co-domain values. In the following table, we enumerate all possible permutations over all possible functions over the co-domain $\{A, B, C\}$ as well as all possible permutations over the set of algorithm behaviors over the set of indices denoted by $\{1, 2, 3\}$:

POSSIBLE BEHAVIORS	POSSIBLE FUNCTIONS
B1: $\langle 1, 2, 3 \rangle$	F1: $\langle A, B, C \rangle$
B2: $\langle 1, 3, 2 \rangle$	F2: $\langle A, C, B \rangle$
B3: $\langle 2, 1, 3 \rangle$	F3: $\langle B, A, C \rangle$
B4: $\langle 2, 3, 1 \rangle$	F4: $\langle B, C, A \rangle$
B5: $\langle 3, 1, 2 \rangle$	F5: $\langle C, A, B \rangle$
B6: $\langle 3, 2, 1 \rangle$	F6: $\langle C, B, A \rangle$

The implications of No Free Lunch start to become clear when one asks basic questions about the set of behaviors and the set of functions.

If we apply any two sets of behaviors to all functions, each behavior generates a set of $3!$ possible search behaviors which is the same as the set of all possible functions. If we apply all possible search behaviors to any two functions, for each function we again obtain a set of behaviors which, after the indices are translated into co-domain values, is the same as the set of all possible functions.

We need to be careful to distinguish between algorithms and their behaviors. There exist many algorithms (perhaps infinitely many) but once the values of the co-domain are fixed, there are only a finite number of behaviors.

[Schumacher \(2000\)](#) and [Schumacher et al. \(2001\)](#) make the No Free Lunch theorem more precise by formally relating it to the *permutation closure* of a set of functions. The result is what is now referred to as the Sharpened No Free Lunch theorem. Let \mathcal{X} and \mathcal{Y} denote finite sets and let $f: \mathcal{X} \rightarrow \mathcal{Y}$ be a function where $f(x_i) = y_i$. Let σ be a permutation such that $\sigma: \mathcal{X} \rightarrow \mathcal{X}$. We can permute functions as follows:

$$\sigma f(x) = f(\sigma^{-1}(x)).$$

Since $f(x_i) = y_i$, the permutation $\sigma f(x)$ can also be viewed as a permutation over the values that make up the co-domain (the output values) of the objective function.

We next define the permutation closure $P(F)$ of a set of functions F :

$$(F) = \{\sigma f : f \in F \text{ and } \sigma \text{ is a permutation}\}.$$

Informally, $P(F)$ is constructed by taking each function in F and re-ordering its co-domain values to produce a new function. This process is repeated until no new functions can be generated. This produces *closure* since every re-ordering of the co-domain values of any function in $P(F)$ will produce a function that is already a member of $P(F)$. Therefore, $P(F)$ is closed under permutation.

The Sharpened No Free Lunch theorem is often informally expressed by saying that when comparing search methods, the No Free Lunch theorem holds if and only if the set of functions used to compare the algorithms is closed under permutations. However in Sect. 16.4 we will see that we should really be more precise about what the Sharpened No Free Lunch theorem really means.

Sharpened NFL does make it clear that No Free Lunch theorems for search apply to finite sets. These sets can in fact be quite small. Proofs are given by [Schumacher et al. \(2001\)](#). Intuitively, that NFL should hold over a set closed under permutations can be seen from Culberson's adversarial argument: any possible remaining value of the co-domain that has not yet been sampled can occur at the next time step of search. To see why this is true, assume we have a bijective function, and we place the N values of the co-domain in a grab-bag. By drawing values of the grab-bag we can construct $N!$ different functions. Denote this set $P(F)$ since it is the permutation closure of some seed function. Assume f_j is one of these functions. Next (using our Apply meta-function) execute algorithm A_1 on function f_j :

$$\text{Apply}(A_1, f_j, m).$$

Next, assume we want to compare the behavior of algorithm A_2 against that of A_1 . For every $f_j \in P(F)$, there exists a function $f_l \in P(F)$ such that

$$\text{Apply}(A_1, f_j, m) = \text{Apply}(A_2, f_l, m).$$

It also follows that for every $f_l \in P(F)$, there exists a function $f_i \in P(F)$ where this statement also holds. This is the essence of Sharpened No Free Lunch. The statement remains true even if the functions are not bijections. But this only establishes one direction of the *if and only if*.

Proving that the connection between algorithm behavior and permutation closure is an *if and only if* relationship is much stronger than the observation that No Free Lunch holds over the permutation closure of a function. But if every remaining value is not equally likely at each time step, then the set of functions we are sampling from is not closed under permutation and No Free Lunch is not guaranteed to hold for arbitrarily chosen search methods. Similar observations have also been made by [Droste et al. \(2002\)](#). As we will see later, the fact that algorithms A_1 and A_2 can be chosen arbitrarily is critical to Sharpened No Free Lunch.

We can now make a more precise statement about the zero-sum nature of No Free Lunch. If algorithm **K** outperforms algorithm **Z** on any subset of functions denoted by β , then algorithm **Z** will outperform algorithm **K** over $P(\beta) - \beta$.

[English \(2000a\)](#) first pointed out that NFL can hold over sets of functions such as needle-in-a-haystack functions. A needle-in-a-haystack function is one that has the same evaluation for every point in the space except one; in effect, searching a needle-in-a-haystack function is necessarily random since there is no information about how to find the needle until after it has been found.

In the following example, NFL holds over just three functions:

$$\begin{aligned} f &= \langle 0, 0, 3 \rangle \\ P(f) &= \{ \langle 0, 0, 3 \rangle, \langle 0, 3, 0 \rangle, \langle 3, 0, 0 \rangle \}. \end{aligned}$$

Clearly, NFL does not just hold over sets that are incompressible. All needle-in-a-haystack functions have a compact representation of size $O(\lg N)$, where $N = |\mathcal{X}|$. In effect, the evaluation function needs to indicate when the needle has been found and return a distinct evaluation.

Generally, we like to construct evaluation functions that are capable of producing a rich and discriminating set of outputs: that is, we like to have evaluation functions that tell us point i is better than point j . But it also seems reasonable to conjecture that if NFL holds over a set that is compressible, then that set has low information measure.

[Schumacher et al. \(2001\)](#) also note that the permutation closure has the following property:

$$P(F \cup F') = P(F) \cup P(F').$$

Given a function f and a function g , where $g \notin P(f)$, we can then construct three permutation closures: $P(f), P(g), P(f \cup g)$. For example, this implies that NFL holds over the following sets which are displayed in table format:

Set 1: {< 3, 0, 0 >, < 0, 3, 0 >, < 0, 0, 3 >}	Set 3: {< 3, 0, 0 >, < 0, 3, 0 >, < 0, 0, 3 >,
Set 2: {< 1, 3, 2 >, < 2, 1, 3 >, < 2, 3, 1 >, < 3, 1, 2 >, < 3, 2, 1 >}	< 1, 3, 2 >, < 2, 1, 3 >, < 2, 3, 1 >, < 3, 1, 2 >, < 3, 2, 1 >

We can also ask about NFL and the probability of sampling a particular function in $P(f)$. For NFL to hold, we must insist that all members of $P(f)$ for a specific function f are uniformly sampled. Otherwise, some functions are more likely to be sampled than others, and NFL breaks down. For NFL to hold over $P(g)$ the probability of sampling a function in $P(g)$ must also be uniform. But [Igel and Toussaint \(2004\)](#) point out that we can also have a uniform sample over $P(g)$ and a (different) uniform sample over $P(f)$ and NFL still holds. Thus, sampling need not be uniform over $P(f \cup g)$.

16.3.3 Free Lunch and Compressibility

[Whitley \(2000\)](#) presents the following observation (the current form is expanded to be more precise):

Theorem: Let $P(f)$ represent the permutation closure of the function f . If f is a bijection, or if any fixed fraction of the co-domain values of f are unique, then $|P(f)| = O(N!)$ and the functions in $P(f)$ have a description length of $O(N \lg N)$ bits on average, where N is the number of points in the search space.

The proof, which is sketched here, follows the well known proof demonstrating that the best sorting algorithms have complexity $O(N \log N)$. We first assume that the function is a bijection and that $|P(f)| = N!$. We would like to tag each function in $P(f)$ with a bit string that uniquely identifies that function. We then make each of these tags a leaf in a binary tree. The tag acts as an address that tells us to go left or right at each point in the tree in order to reach a leaf node corresponding to that function. But the tag also uniquely identifies the function. The tree is constructed in a balanced fashion so that the height of the tree corresponds to the number of bits needed to tag each function. Since there are $N!$ leaves in the tree, the height of the tree must be $O(\log N!) = O(N \log N)$. Thus $O(N \log N)$ bits are required to uniquely label each function. Standard binary labels can be compressed by dropping leading zeros, but only 1/2 of the strings can be compressed, so the complexity is still $O(N \log N)$ on average.

To construct a lookup table or a full enumeration of any permutation of N elements requires $O(N \log N)$ bits, since there are N elements and $\log N$ bits are needed to distinguish each element. Thus, most of these functions have exponential description.

This is, of course, one of the major concerns about No Free Lunch theorems. Do No Free Lunch theorems really apply to sets of functions which are of practical interest? Yet this same concern is often overlooked when theoretical researchers wish to make mathematical observations about search. For example, proofs which calculate the number of expected optima over all possible functions (Rana and Whitley 1998), or the expected path length to a local optimum over all possible functions (Tovey 1985) under local neighborhood search are computed with respect to the set of $N!$ functions.

Igel and Toussaint (2003) formalize the idea that if one considers all the possible ways that one can construct subsets over the set of all possible functions, then those subsets that are closed under permutation are a vanishing small percentage. This problem with this observation is that the a priori probability of *any* subset of problems is vanishingly small—including any set of applications we might wish to consider. On the other hand, Droste et al. (2002) have also shown that for any function for which a given algorithm is effective, there exist related functions for which performance of the same algorithm is substantially worse.

16.4 Sharpened NFL and Focused NFL

We might express the Sharpened No Free Lunch theorem more precisely as follows:

The aggregate behaviors of any two arbitrarily chosen search algorithms are guaranteed to be equivalent if and only if the algorithms are compared on a set of functions that are closed under permutation.

It is important to stress that the algorithms are arbitrarily chosen. If we do not know which algorithms are being compared, then the best we can do to ensure that the algorithms display identical behaviors is to compare them on a set of functions that are closed under permutation.

What if we wish to compare two specific search algorithms, A_i and A_k , algorithms that we have knowledge about? Does the “Sharpened No Free Lunch” result still apply?

We will again use the function $\text{Apply}(A_i, f_j, m)$. Recall that this meta-level algorithm outputs the order in which A_i visits m elements in the co-domain of f_j after m steps. We can also reconfigure Apply to be a function generator. We will call the function generator **APPLY** such that

$$f_{\text{out}} = \mathbf{APPLY}(A_i, A_k, f_{\text{in}}, m) \iff \text{Apply}(A_i, f_{\text{in}}, m) \equiv \text{Apply}(A_k, f_{\text{out}}, m).$$

We can now define a set that is closed with respect to the operation of the **APPLY** function. Assume that we will be given some as yet unknown algorithms A_i and A_k , and we start with a set F which contains a single function f_1 . We assign $f_{\text{in}} = f_1$ and we generate a function $f_{\text{out}} = f_2$.

Define the set $C(F)$ such that the set F is a subset of $C(F)$ and if f_{in} is a member of $C(F)$ then $f_{\text{out}} = \mathbf{APPLY}(A_i, A_j, f_{\text{in}})$ is also a member of $C(F)$.

Can we define $C(F)$ in advance so that any two arbitrarily chosen algorithms A_k and A_i are guaranteed to have the same behaviors? The Sharpened No Free Lunch theorem states that $C(F)$ must be a set that is closed under permutation if A_k and A_i are arbitrarily chosen (as yet unknown algorithms) and we require that algorithms A_k and A_i have identical performance when compared on all in the functions in $C(F)$.

But what if we wish to compare exactly two algorithms, A_1 and A_2 , and we are told in advance what algorithms are going to be compared? In this case we can potentially find a closure $C(F)$ defined with respect to the **APPLY** function such that the set $C(F)$ need not be closed under permutation in order for algorithms A_1 and A_2 to display the same aggregate performance over the set of function in $C(F)$ for all possible comparative measures.

Using these ideas, [Whitley and Rowe \(2008\)](#) present the key ideas behind the Focused No Free Lunch theorem:

Let A_1 and A_2 be two predetermined algorithms and let F be a set of functions. The aggregate performance of A_1 and A_2 are equivalent over the set $C(F)$; furthermore, the set $C(F)$ need not be closed under permutation.

Actually, we might also compare 3 or 4 or 20 predetermined algorithms and ask if a set exists where all of the selected algorithms have the same behavior. But for now, looking at just two algorithms is enough to establish the behavior in which we are interested. We will look at two different ways in which Focused No Free Lunch can hold.

First, assume that the two algorithms are deterministic. The search behaviors of A_1 and A_2 when executed on a function f_1 can induce a permutation group such that the orbit of the group is smaller than the permutation closure. For example, assume that A_1 is a local search algorithm that uses a binary bit encoding, and A_2 is a local search algorithm that uses a Gray code bit encoding; assume that both algorithms use the same restart mechanism. Otherwise A_1 and A_2 apply exactly the same search strategy; the only difference is that one uses the binary representation and the other uses a Gray code. Then we can prove that when $F = \{f_1\}$, then the size of $C(F)$ is less than or equal to $2L$ where L is the number of bits used to encode the search space.

[Whitley and Rowe \(2008\)](#) show that this happens because repeated application of Gray encoding induces a group whose orbit is always made up of a set of between L and $2L$ functions. To see why this is true, consider a bit string of length 3. We will take a binary string, then Gray code it, and Gray code it again. Let G^1 denote one application of Gray code, G^2 will denote two applications, and G^x will denote x applications of Gray code:

$$\begin{aligned} \text{Binary} &= 000 \quad 001 \quad 010 \quad 011 \quad 100 \quad 101 \quad 110 \quad 111 \\ G^1 &= 000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100 \\ G^2 &= 000 \quad 001 \quad 010 \quad 011 \quad 101 \quad 100 \quad 111 \quad 110 \\ G^2 &= 000 \quad 001 \quad 011 \quad 010 \quad 111 \quad 110 \quad 100 \quad 101 \\ G^4 &= 000 \quad 001 \quad 010 \quad 011 \quad 100 \quad 101 \quad 110 \quad 111 \end{aligned}$$

By the fourth application of Gray code, the encoding has cycled back to the same as the original binary encoding. Thus the performance of any algorithm using G^4 as a representation is identical to the performance of the same algorithm using the binary coding. The first 4 sets of bit strings form a group with an orbit of 4.

We will generate four functions in the following way: Gray the original function four times, then assume that the bit pattern that is produced is actually the binary encoding of a new function. In other words, each bit string in each representation will be treated as if it is a binary string, b , and the function $f(b) = i$ will return the integer corresponding to the bit string b . (Without loss of generality, we can represent the seven co-domain values as integers.) The binary representation and the Gray code representations are transformed into the following four functions:

	$f_i(1)$	$f_i(2)$	$f_i(3)$	$f_i(4)$	$f_i(5)$	$f_i(6)$	$f_i(7)$
$f_1 =$	0	1	2	3	4	5	6
$f_2 =$	0	1	3	2	6	7	5
$f_3 =$	0	1	2	3	5	4	7
$f_4 =$	0	1	3	2	7	6	4
$f_5 =$	0	1	2	3	4	5	6

Since $f_1 = f_5$ there are only four distinct functions. One can show by construction that the Gray code representation of f_i induces exactly the same search space as the binary representation of f_{i+1} .

Let A_b be any algorithm that uses a binary coding; let A_g be exactly the same algorithm except that it uses a Gray encoding. On the set of four functions we have defined, algorithms A_b and A_g will have identical performance. This implies

$$\text{Apply}(A_b, f_2, m) = \text{Apply}(A_g, f_1, m)$$

$$\text{Apply}(A_b, f_3, m) = \text{Apply}(A_g, f_2, m)$$

$$\text{Apply}(A_b, f_4, m) = \text{Apply}(A_g, f_3, m)$$

$$\text{Apply}(A_b, f_5, m) = \text{Apply}(A_g, f_4, m).$$

And since $f_5 = f_1$ we have constructed a set $C(F) = \{f_1, f_2, f_3, f_4\}$ that is not closed under permutation. Therefore a Focused No Free Lunch holds. [Whitley and Rowe \(2008\)](#) generalize this result to show that when comparing algorithm A_g and A_b for inputs of L bits, the size of the set $C(F)$ is always less than $2L$. By contrast, the size of the permutation closure $P(F)$ is $2^L!$ for a single seed function. Whitley and Rowe also show that Focused No Free Lunch that exploit the orbits of groups also holds for other classes of search algorithms. In this case Focused No Free Lunch holds even if the entire search space is exhaustively explored.

So, when comparing two specific algorithms we can sometimes look at sets of functions smaller than the permutation closure, and still observe identical performance for the two algorithms we are comparing. Furthermore, while Sharpened No Free Lunch holds for the black-box optimization method, Focused No Free Lunch does not require that the optimization method be a black-box optimizer. For example, for our algorithms A_b and A_g we can have various kinds of information about the

functions we are optimizing. We can know how many parameters there are, where the parameter boundaries are, and we might exploit domain-specific knowledge so that we apply different search strategies for different parameters. Algorithms A_b and A_g can in fact use *any* information we might want to include about the set of functions as long as the only difference between the two algorithms is that one uses a binary encoding and the other uses a Gray encoding. Search need not be blind or black box for Focused No Free Lunch results to hold.

There is a second way in which Focused No Free lunch results can occur. In all real applications the number of points that we sample, denoted by m , is polynomial with respect to input size of the problem, while the search space is exponential. Let N denote the size of the search space. Reconsider the computation using the **APPLY** function where $m \ll N$:

$$f_{\text{out}} = \text{APPLY}(A_i, A_j, f_{\text{in}}, m).$$

There now can be exponentially many functions that can play the role of f_{out} because the behavior of f_{out} is defined at only m points in the search space, and the other points in the search space can be reconfigured in any of $(N - m)!$ ways, all of which are unique if the function f_{in} is a bijection. Intuitively, we no longer need the entire permutation closure to obtain identical over some set of functions when only a tiny fraction of the search space is explored. In fact, under certain conditions one can prove that given two predetermined algorithms A_1 and A_2 there can exist functions f_1 and f_2 such that

$$\text{Apply}(A_1, f_1, m) \equiv \text{Apply}(A_2, f_2, m)$$

$$\text{Apply}(A_1, f_2, m) \equiv \text{Apply}(A_2, f_1, m)$$

so that a Focused No Free Lunch result holds over a set of only two functions such that $C(F) = \{f_1, f_2\}$. This can occur, for example, if the two search algorithm never sample the same domain values or co-domain values on any test function. (It should be noted that this requirement is sufficient, but not strictly necessary; [Whitley and Rowe \(2008\)](#) present a more general result.)

For example, consider the following functions:

$$f_1 = \langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle.$$

Assume that search algorithm A_1 samples $f_1(4) = 3$ and $f_1(8) = 7$ and halts. Assume that search algorithm A_2 samples $f_1(3) = 2$ and $f_1(6) = 5$ and halts. We can construct a second function f_2 on the fly. We will assume that each algorithm starts at the same domain value. To make their behaviors the same, we require that A_1 samples $f_2(4) = 2$ and that A_2 samples $f_2(3) = 3$. This has the following implications for function f_2 :

$$f_2 = \langle ?, ?, 3, 2, ?, ?, ?, ? \rangle.$$

In this case, a ? symbol means that the co-domain value of the function at that location has not yet been determined, and the function is under-specified. Next we

see where the algorithms sample next as search continues. Assume that A_1 decides to sample $f_2(1)$ and that A_2 decides to sample $f_2(5)$. Then we continue to define function f_2 so that $f_2(1) = 5$ and $f_2(5) = 7$. We can do this as long as they do not sample the same points. This has the following implications for the construction of function f_2 :

$$f_2 = \langle 7, ?, 3, 2, 5, ?, ?, ? \rangle.$$

Thus, after sampling only two points in the search space we see that there must exist a function f_2 such that

$$\text{Apply}(A_1, f_1, 2) \equiv \text{Apply}(A_2, f_2, 2)$$

$$\text{Apply}(A_1, f_2, 2) \equiv \text{Apply}(A_2, f_1, 2)$$

and a set $C(F) = \{f_1, f_2\}$ can be defined which is smaller than the permutation closure. It does not matter than function f_2 is under-specified, and that in fact, f_2 actually represents a family of functions, all of which produce the desired behavior.

What happens if we are not so lucky, and algorithms A_1 and A_2 sample some of the same domain values? Whitley and Rowe (2009) present a constructive algorithm that creates a set of functions that yield a set of under-specified functions which in turn creates a closure $C(F)$. We can think of the set F as either allowing under-specified functions, or we can make every function in F specific by randomly filling in the unspecified co-domain values with unused co-domain values.

16.4.1 Partitioning the Permutation Closure under Focused NFL

There can be many ways to partition a set that is closed under permutation to obtain many additional sets that display Focused No Free Lunch results. Let $P(F)$ be the permutation closure of a set of functions denoted by F . Let $C(F)$ be a set of functions such that the specific algorithms A_k and A_j have identical performance on the set of function $C(F)$. In this case, we assume that $C(F)$ contains fully specified functions. Assume that $C(F)$ is a proper subset of $P(F)$. Then a Focused No Free Lunch result holds over $C(F)$ but a Focused No Free Lunch result must also hold over the set $P(F) - C(F)$ as well. Let $\mathcal{F}_1 = C(F)$. If we can extract a second proper subset \mathcal{F}_2 from the residual set $P(F) - \mathcal{F}_1$ such that algorithms A_k and A_j have identical performance over the functions in set \mathcal{F}_2 , then the algorithms will also have identical performance over the residual set $P(F) - \mathcal{F}_1 - \mathcal{F}_2$. By recursively extending this idea, we can decompose the set $P(F)$ into subsets such that A_k and A_j will have identical performance over the functions in each subset $\mathcal{F}_i \subset P(F)$. This also means that $P(F)$ decomposes such that

$$P(F) = \bigcup \mathcal{F}_i.$$

Furthermore, recall that when we are constructing the set $C(F)$ under the conditions that we limit search to m steps, there can be exponentially many different functions that display the same behaviors for the first m steps. Thus, when constructing $C(F)$, there is not a unique way to partition $P(F)$. Assume that we pick a different set of functions, G such $G \in P(F)$ and we define $C(G)$ so that A_k and A_j have identical performance over the functions in $C(G)$. Furthermore, assume that $\forall i, C(G) \neq \mathcal{F}_i$. Let \mathcal{G}_1 denote the set $C(G)$. Then we can also define a different set of partitions where

$$P(F) = \bigcup \mathcal{G}_i = \bigcup \mathcal{F}_i.$$

However, the decomposition represented by the sets $\bigcup \mathcal{G}_i$ can be completely different from the decomposition represented by the sets $\bigcup \mathcal{F}_i$.

For example, one can construct cases where even the average size of the subsets that make up $\bigcup \mathcal{G}_i$ is different from the average size of the subsets in $\bigcup \mathcal{F}_i$. One can attempt to construct sets \mathcal{F}_i such that every set is as small as possible. On the other hand, one can allow the search to “wander” through various random functions and allow the subsets that make up $\bigcup \mathcal{G}_i$ to grow larger before attempting to construct a function to create a closure.

Again, consider the following function:

$$f_1 = \langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle.$$

Assume that search algorithm A_1 samples $f_1(4) = 3$ and $f_1(8) = 7$ and halts. Assume that search algorithm A_2 samples $f_1(3) = 2$ and $f_1(6) = 5$ and halts. Now, instead of constructing a second function f_2 that produces a closure as quickly as possible, we will add an additional random function that causes the size of the subset to become larger.

We start to construct a different function f_2^* as follows. We still require that A_2 mimic the behavior of A_1 on f_1 , thus we assume that A_2 samples $f_2^*(3) = 3$ and then $f_2^*(0) = 7$. This results in the partial construction of f_2^* :

$$f_2^* = \langle 7, ?, 3, ?, ?, ?, ?, ? \rangle.$$

This is sufficient to ensure that

$$\text{Apply}(A_1, f_1, 2) \equiv \text{Apply}(A_2, f_2, 2).$$

However, there can be up to $(N - m)!$ different ways of making an underspecified function specific. Suppose we pick the missing values in f_2^* randomly as follows:

$$f_2^* = \langle 7, 6, 3, 5, 1, 4, 0, 2 \rangle.$$

And now we execute algorithm A_1 for two steps; assume that it samples $f_2^*(4) = 5$ and $f_2^*(6) = 4$ and halts. We now are forced to construct a third function. But now, we will try to assign values to the underspecified function to create a closure. If possible, we want to create a function f_3 where

$$\text{Apply}(A_1, f_1, 2) \equiv \text{Apply}(A_2, f_2, 2)$$

$$\text{Apply}(A_1, f_2, 2) \equiv \text{Apply}(A_2, f_3, 2)$$

$$\text{Apply}(A_1, f_3, 2) \equiv \text{Apply}(A_2, f_1, 2).$$

With this goal in mind, we create construct f_3 as follows:

$$f_3 = \langle ?, 4, ?, 5, 2, ?, ?, ? \rangle.$$

Now assume that on f_3 algorithm A_1 samples $f_3(4) = 5$ and $f_3(2) = 4$ and halts and search algorithm A_2 samples $f_3(5) = 2$ and $f_3(4) = 5$ and halts. This yields the desired result. But it yields a different partitioning of the permutation space, because function f_1 is the same in both cases, but f_1 ends up in partitions of different size. Therefore, there is not a unique way to partition the permutation closure.

There can be many different ways of partitioning the permutation closure. Assume that a f_1 is given and that function f_2 is defined to satisfy the following condition:

$$\text{Apply}(A_1, f_1, m) \equiv \text{Apply}(A_2, f_2, m).$$

If after placing a function f_1 in a partition we can randomly pick any of the exponential many functions from the set of $(N - m)!$ possibilities as the second function, assume we construct them all in parallel, but we attempt to make each closure a different size. This is sufficient to create a large number of ways of partitioning the permutation closure.

16.4.2 Evaluating Search Algorithms

From a theoretical point of view, comparative evaluation of search algorithms is a dangerous, if not dubious, enterprise. But the alternative to testing is to just give up and say that all algorithms are equal—which means we have no way of recommending one algorithm over another when a search method is required to solve a problem of practical interest. The best we can do is build test functions that we believe capture some aspects of the problems we actually want to solve. But this highlights a critical question. Do benchmarks really test what we want to test? If an algorithm does well on a very simple problem—such as a linear objective function—is that good or bad? Many people have used the ONEMAX test function for testing search algorithms that use a binary representation. The objective function for ONEMAX is to maximize number of bits set to 1 in a bit string. But should we really believe that an algorithm that does well on ONEMAX generalizes to other problems of practical interest? Theory would suggest extreme caution.

Each instance of an optimization problem has an associated objective function. Let β represent a particular set of benchmark functions. NFL implies that if algorithm **K** is better than algorithm **Z** on the benchmark set β , then algorithm **Z** must be better than **K** on the instances in $P(\beta) - \beta$. NFL theorems make it clear

that comparative evaluation is really a zero-sum game. But the $P(\beta) - \beta$ might be exponentially large and uncompressible. Focused No Free Lunch indicates that the zero-sum game can be played at a much smaller scale. Focused NFL implies that if algorithm **K** is better than algorithm **Z** on the benchmark set β , then algorithm **Z** must be better than **K** on the instances in $C(\beta) - \beta$. The set $C(\beta)$ where Focused No Free Lunch plays out can be very small indeed. Thus, there is even more reason to suggest extreme caution.

So what does it mean to evaluate an algorithm on a set of benchmarks and compare it to another algorithm? Given the NFL theorems, comparison is meaningless unless we prove (which virtually never happens) or assume (an assumption which is rarely made explicit) that the benchmarks used in a comparison are somehow representative of a particular subclass of problems.

Benchmarks are commonly used for testing both optimization and learning algorithms. Often, the legitimacy of a new algorithm is “established” by demonstrating that it finds better solutions than existing algorithms when evaluated on a particular benchmark or collection of benchmarks. Alternatively, the new algorithm may find high-quality solutions faster than existing algorithms for one or more benchmarks.

What are some of the dangers associated with the use of benchmarks? Algorithms can be tuned such that they perform well on specific benchmarks, but fail to exhibit good performance on benchmarks with different characteristics. More importantly, there is no guarantee that algorithms developed and evaluated using synthetic benchmarks will perform well on more realistic problem instances. Furthermore, simple algorithms can often provide excellent performance on more realistic benchmarks (Watson et al. 1999).

While the dangers associated with benchmarks are well known, most researchers continue to use benchmarks to evaluate their algorithms. This is because researchers have few alternatives. How can one algorithm be compared to another without some form of evaluation? Evaluation requires the use of either synthetic or real-world benchmarks, or at least the use of test problems drawn from problem generators so that algorithms can be compared on sets of problem instances that have similar characteristics. Researchers who develop new algorithms and do not demonstrate their merit through some form of comparative testing can expect their work to be ignored. The compulsion to develop “a new method” has resulted in the literature being full of new algorithms, most of which are never used or analyzed by anyone other than the researchers who created them.

Hooker (1995) discusses the “evils of competitive testing” and points out the difficulty of making fair comparisons of algorithm performance. Implementation details can significantly impact algorithm performance, as can the values selected for various tuning parameters. Some algorithms have been refined for years. Other algorithms have become so specialized that they only work well on specific benchmarks. Hooker argues that the evaluation of algorithms should be performed in a more scientific, hypothesis-driven manner. Barr et al. (1995) suggest guidelines for the experimental evaluation of heuristic methods. Such guidelines are for the most part useful, although rarely followed.

While evaluation is difficult, it is also important. Too many experimental papers (especially conference papers) include no comparative evaluation; researchers may present a hard problem (perhaps newly minted) and then present an algorithm to solve the problem. The question as to whether some other algorithm could have done just as well (or better!) is ignored.

16.5 Conclusions

As in many other areas of life, extreme reactions are likely to lead to extreme errors. This is also true for No Free Lunch. It is clearly wrong to say “NFL doesn’t apply to real-world problems, so who cares?” It is also an error to give up on building general purpose search algorithms.

A careful consideration of the No Free Lunch theorems forces us to ask what set of problems we want to solve and how to solve them. More than this, it encourages researchers to consider more formally whether the methods they develop for particular classes of problems actually are better than other algorithms. This may involve proofs about performance behavior. In some ways, we are just starting to ask the right questions. And yet, researchers working in complexity and NP-completeness have long been concerned with algorithm performance for particular classes of problems.

Few researchers have attempted to formalize their assumptions about search problems and search algorithm behavior. But if we fail to do this, then we become trapped in a kind of empirical and experimental treadmill that leads nowhere: algorithms are developed that work on benchmarks, or on particular applications, without any evidence that such methods will work on the next problem we might wish to solve.

Unfortunately, it is not widely understood that there are significant differences in the Focused and Sharpened No Free Lunch results. And there are examples in the literature where the Sharpened No Free Lunch result has been overstated to imply that for any two (predetermined) algorithms, the behaviors of those algorithm will be identical if and only if the algorithms are compared over a set of functions closed under permutation. Focused No Free Lunch proves that this interpretation is incorrect as well as very misleading. Correcting this misunderstanding can only help to also clarify our understanding as to what it means to compare algorithms.

16.6 Tricks of the Trade

No Free Lunch is a theoretical result about search algorithms. As such there are no specific methods or algorithms that directly follow from NFL. Several pieces of advice do follow from No Free Lunch.

1. In most practical applications one must trade-off generality and specificity. Using simpler off-the-shelf search methods reduces time effort and cost. Simple but reasonably effective search methods, even when implemented from scratch, are often easier to work with than complex methods. Using custom-designed search methods that only work for one application will usually yield better results: but generally, one must ask how much time and money one wishes to spend and how good the solution needs to be.
2. Exploit problem-specific information when it is simple to do so. For example, most NP-complete problems have been studied for years and there are many problem-specific methods that yield good near-optimal solutions.
3. For discrete parameter optimization problems, one has a choice of using standard binary encodings, Gray codes or real-valued representations. Gray codes are often better than binary codes when some kind of neighborhood search is used either explicitly (e.g. local search) or implicitly (e.g. via a random bit flip operator). The use of Gray codes versus real-valued is less clear, and depends on other algorithm design choices.
4. Don't assume that a search method that does well on classic benchmarks will work equally well on real-world problems. Sometimes algorithms are overly tuned to do well on benchmarks and in fact don't work well on real-world applications.

16.7 Current and Future Research Directions

One body of the literature asks the question “What representation is best?” Of course, the answer is that other No Free Lunch theorems show that in the general case there is no best representation. For discrete parameter optimization problems, one might use standard binary representations, or standard binary-reflect Gray codes. Or one might use real-valued floating point representations.

Another area of research is the construction of algorithms that can provably beat random enumeration on specific subsets of problems. [Christensen and Oppacher \(2001\)](#) prove that No Free Lunch does not hold over sets of functions that can be described using polynomials of a single variable of bounded complexity. This also includes Fourier series of bounded complexity. (Also see a [2000a](#) paper by [English](#) about polynomials and No Free Lunch.) They define a minimization algorithm called SubMedian-Seeker. The algorithm assumes that the target function f is one-dimensional and bijective and that the median value of f is known and denoted by $\text{med}(f)$. The actual performance depends on $M(f)$, which measures the number of submedian values of f that have *successors* with supermedian values. They also define M_{crit} as the critical value of $M(f)$ such that when $M(f) < M_{\text{crit}}$ SubMedian-Seeker is better than random search. Christensen and Oppacher then prove:

If f is a uniformly sampled polynomial of degree at most k and if $M_{\text{crit}} > k/2$ then SubMedian-Seeker beats random search.

The SubMedian-Seeker is not a practical algorithm. The importance of Christensen and Oppacher's work is that it sets the stage for proving that there are algorithms that are generally (if perhaps weakly) effective over a very broad class of interesting, nonrandom functions. More recently, [Whitley et al. \(2004\)](#) have generalized these concepts to outline conditions which allow local neighborhood bit climbers to display SubThreshold-Seeker behavior and then show that in practice such algorithms spend most of their time exploring the best points in the search space on common benchmarks and are obviously better than random search.

Sources of Additional Information

The classic textbook *Introduction to Algorithms* by Cormen et al. has a very good discussion of NP-completeness and approximate algorithms for some well-studied NP-hard problems.

Joe Culberson's [1998](#) paper *On the Futility of Blind Search: An Algorithmic View of No Free Lunch* helps to relate complexity theory to No Free Lunch in simple and direct terms. Tom English has contributed several good papers to the NFL discussion ([English 2000a,b](#)). Igel and Toussaint have also contributed notable papers. Chris Schumacher's [2000](#) PhD dissertation, *Fundamental Limitations on Search Algorithms*, deals with various issues related to No Free Lunch.

Work by Ingo Wegener and colleagues has focused on showing when particular methods work on particular general classes of problems, (e.g. [Storch and Wegener 2003](#); [Fischer and Wegener 2004](#)) or showing the inherent complexity of particular problems for black-box optimization ([Droste et al. 2003](#)).

[Corne and Knowles \(2003\)](#) examine questions about No Free Lunch in the space of multi-objective optimization.

[Auger and Teytaud \(2008\)](#) look at the question of whether No Free Lunch applies to continuous functions and conclude that “continuous lunches are free”. A paper by [Rowe et al. \(2009\)](#) entitled *Reinterpreting No Free Lunch* presents a general set-theoretic interpretation of Sharpened No Free Lunch which examines symmetries over arbitrary domains and co-domain values. Whether No Free Lunch holds over continuous parameter optimization problems in practice may depend on what assumptions one makes about these spaces.

References

- Auger A, Teytaud O (2008) Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica* 57:121–146
- Barr R, Golden B, Kelly J, Resende M, Stewart W Jr (1995) Designing and reporting on computational experiments with heuristic methods. *J Heuristics* 1:9–32

- Christensen S, Oppacher F (2001) What can we learn from no free lunch? In: GECCO 2001, San Francisco. Morgan Kaufmann, San Mateo, pp 1219–1226
- Cook S (1971) The complexity of theorem proving procedures. In: 3rd annual ACM symposium on theory of computing, pp 151–158. ACM, New York
- Cormen T, Leiserson C, Rivest R (1990) Introduction to algorithms. McGraw-Hill, New York
- Corne D, Knowles J (2003) Real royal road functions for constant population size. In: Evolutionary multi-criterion optimization. LNCS 2632. Springer, Berlin
- Culberson J (1998) On the futility of blind search. *Evol Comput* 6:109–127
- Droste S, Jansen T, Wegener I (1999) Perhaps not a free lunch, but at least a free appetizer. In: GECCO 1999, Orlando. Morgan Kaufmann, San Mateo, pp 833–839
- Droste S, Jansen T, Wegener I (2002) Optimization with randomized search heuristics; the (A)NFL theorem, realistic scenarios and difficult functions. *Theor Comput Sci* 287:131–144
- Droste S, Jansen T, Tinnefeld K, Wegener I (2003) A new framework for the valuation of algorithms for black-box optimization. Foundations of genetic algorithms. Morgan Kaufmann, San Mateo
- English T (2000a) Practical implications of new results in conservation of optimizer performance. In: Proceedings of the PPSN 6, Paris. Springer, Berlin, pp 69–78
- English T (2000b) Optimization is easy and learning is hard in the typical function. Proceedings of the CEC 2000, pp 924–931
- Fischer S, Wegener I (2004) The Ising model on the ring: mutation versus recombination. In: Proceedings of the GECCO 2004, Seattle. Springer, Berlin, pp 1113–1124
- Hooker JN (1995) Testing heuristics: we have it all wrong. *J Heuristics* 1:33–42
- Horowitz E, Sahni S (1978) Fundamentals of computer algorithms. Computer Science Press, Washington, DC
- Igel C, Toussaint M (2003) On classes of functions for which No Free Lunch results hold. *Inf Process Lett* 86:317–321
- Igel C, Toussaint M (2004) A no-free-lunch theorem for non-uniform distributions of target functions. *J Math Model Algorithms* 3:313–322
- Kauffman SA (1989) Adaptation on rugged fitness landscapes. In: Stein DL (ed) Lectures in the science of complexity, pp 527–618. Addison-Wesley, Reading
- Radcliffe NJ, Surry PD (1995) Fundamental limitations on search algorithms: evolutionary computing in perspective. In: van Leeuwen J (ed) Computer science today. LNCS 1000. Springer, Berlin
- Rana S, Whitley D (1997) Representations, search and local optima. In: Proceedings of the AAAI 1997, Providence. MIT, Providence, pp 497–502
- Rana S, Whitley D (1998) Search, representation and counting optima. In: Davis L, De Jong K, Vose M et al (eds) Proceedings of the IMA workshop on evolutionary algorithms. Springer, Berlin
- Rawlins G (ed) (1991) Foundations of genetic algorithms. Morgan Kaufmann, San Mateo
- Rowe J, Vose M, Wright A (2009) Reinterpreting no free lunch. *Evol Comput* 17:117–129

- Schumacher C (2000) Fundamental limitations of search. PhD thesis, University of Tennessee
- Schumacher C, Vose M, Whitley D (2001) The no free lunch and problem description length. In: Proceedings of the GECCO 2001, San Francisco. Morgan Kaufmann, San Mateo, pp 565–570
- Storch T, Wegener I (2003) Real royal road functions for constant population size. In: Proceedign of the GECCO 2003, Chicago. Springer, Berlin, pp 1406–1417
- Sudcamp T (1997) Languages and machines, 2nd edn. Addison-Wesley, Reading
- Tovey CA (1985) Hill climbing and multiple local optima. SIAM J Algebr Discret Methods 6:384–393
- Watson JP, Barbulescu L, Whitley D, Howe A (1999) Algorithm performance and problem structure for flow-shop scheduling. In: Proceeding of the AAAI 1999, Orlando, pp 688–695
- Whitley D (1999) A free lunch proof for Gray versus binary encodings. In: Proceedings of the GECCO 1999, Orlando. Morgan Kaufmann, San Mateo, pp 726–733
- Whitley D (2000) Functions as permutations: regarding no free lunch, walsh analysis and summary statistics. In: Schoenauer M et al (eds) Proceedings of the PPSN 6, Paris. LNCS 1917. Springer, Berlin, pp 169–178
- Whitley D, Rowe J (2008) Focused no free lunch theorems. In: Proceedings of the GECCO 2008, Atlanta. ACM, New York
- Whitley D, Rana S, Heckendorf R (1997) Representation issues in neighborhood search and evolutionary algorithms. In: Poloni C et al (eds) Genetic algorithms and evolution strategies in engineering and computer science. Wiley, New York, pp 39–57
- Whitley D, Rowe J, Bush K (2004) Subthreshold seeking behavior and robust local search. In: Proceedings of the GECCO 2004, Seattle. Springer, Berlin, pp 282–293
- Wolpert DH, Macready WG (1995) No free lunch theorems for search. Technical report SFI-TR-95-02-010, Santa Fe Institute
- Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 4:67–82

Chapter 17

Machine Learning

Xin Yao and Yong Liu

17.1 Introduction

Machine learning is a very active sub-field of artificial intelligence concerned with the development of computational models of learning. Machine learning is inspired by the work in several disciplines: cognitive sciences, computer science, statistics, computational complexity, information theory, control theory, philosophy and biology. Simply speaking, machine learning is learning by machine. From a computational point of view, machine learning refers to the ability of a machine to improve its performance based on previous results. From a biological point of view, machine learning is the study of how to create computers that will learn from experience and modify their activity based on that learning as opposed to traditional computers whose activity will not change unless the programmer explicitly changes it.

17.1.1 Learning Models

A machine learning model has two key components: a learning element and a performance element, as shown in Fig. 17.1. The environment supplies some information to the learning element. The learning element then uses the information to modify the performance element so that it can make better decisions. The performance element selects actions to perform its task.

X. Yao (✉)

School of Computer Science, University of Birmingham, Birmingham, UK
e-mail: X.Yao@cs.bham.ac.uk

Y. Liu

University of Aizu, Aizuwakamatsu, Fukushima, Japan
e-mail: yliu@u-aizu.ac.jp

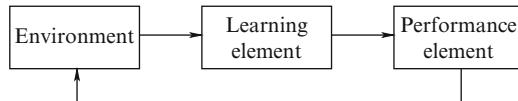


Fig. 17.1 A machine learning model

A large variety of learning elements have been proposed by researchers in the machine learning field. Based on the representation, there is symbolic and subsymbolic learning. Based on the algorithms, there are many different types of machine learning, such as decision tree, inductive logic programming, Bayesian learning, artificial neural networks, evolutionary learning and reinforcement learning. Based on the feedback available, there are three different types of machine learning: supervised, unsupervised and reinforcement learning.

The problem of supervised learning involves learning a function from a set of input–output examples. The general supervised learning model consists of two components:

1. A probability space (\mathcal{E}, Pr) in which we associate each elementary event with two random variables, the input pattern \mathbf{x} and the desired output y , where \mathcal{E} is the event set, Pr is the probability distribution, $\mathbf{x} \in R^P$, and y is a scalar. The assumption that the output y is a scalar has been made merely to simplify exposition of ideas without loss of generality.
2. A learning machine, capable of implementing a set of functions $F(\mathbf{x}, \mathbf{w}), \mathbf{w} \in W$, where W is a set of, in general, real-valued parameters.

The purpose of supervised learning is to find the function $F(\mathbf{x}, \mathbf{w})$ so that the expected squared error

$$R(\mathbf{w}) = E[(F(\mathbf{x}, \mathbf{w}) - y)^2] \quad (17.1)$$

is minimized, where E represents the expectation value over the probability space (\mathcal{E}, Pr) .

In unsupervised learning, there is no specific output supplied. In the context of pattern classification, unsupervised learning learns to discover the statistical regularities of the patterns in the input, form internal representations for encoding features of the input, and thereby to create new classes automatically. In reinforcement learning, rather than being told what to do by a teacher, the learning of an input–output mapping is performed through continued interaction with the environment in order to minimize a scalar index of performance.

The environment can be either fully observable or partially observable. In the first case, the machine can observe the effects of its action and hence can use supervised learning methods to learn to predict them. In the second case, the immediate effects might be invisible so that reinforcement learning or unsupervised learning should be adopted.

17.1.2 Learning Tasks and Issues in Machine Learning

Machine learning can be applied to tasks in many domains. This section presents some important learning tasks and issues in machine learning.

17.1.2.1 Classification

A classification task in machine learning is to take each instance and assign it to a particular class. For example, in an optical character recognition task, the machine is required to scan an image of a character and output its classification. In the English language recognition, the task involves learning the classification of the digits 0...9 and the characters A...Z.

17.1.2.2 Regression, Interpolation and Density Estimation

In regression, the aim is to learn some functional description of data in order to predict values for new input. An example of learning a regression function is predicting the future value of a share index in the stock market. In interpolation, the function for certain ranges of input is known. The task is to decide the function for intermediate ranges of input. In density estimation, the task is to estimate the density or probability that a member of a certain category will be found to have particular features.

17.1.2.3 Learning a Sequence of Actions

In robot learning and chess play learning, the task is to find the best strategies that can choose the optimal actions. In an example of robot navigation, a robot is assigned a task to track a colored object within a limited number of actions while avoiding obstacles and walls in an environment. There are obstacles of different shapes in the environment enclosed by the walls. To perform its task, the robot must learn the basic behavior of obstacle avoidance and moving to the target. It must also learn to co-ordinate the behavior of obstacle avoidance and the behavior of moving to the target to avoid becoming stuck due to repetition of an identical sensor-motion sequence. In chess playing, machine must decide an action based on the state of the board to move a piece in which the action will maximize its chance of winning the game.

17.1.2.4 Data Mining

The problem of data mining is of searching for interesting patterns and important regularities in large databases. Many learning methods have been developed

for determining general descriptions of concepts from examples in the form of relational data tables. Machine learning plays an important role in discovering and presenting potentially useful information from data in a form which is easily comprehensible to humans.

17.1.2.5 Issues in Machine Learning

There are many issues that need to be solved in machine learning. For example, which learning algorithm performs best for a particular learning task and representation? How many training samples are sufficient? How fast can the learning algorithms converge? When and how can prior knowledge be used in the learning process? Can a machine learn in real-time or only via offline learning? How do we choose from among multiple learning models that are all consistent with the data? Among all these issues, generalization is a key issue for any learning system. There are often two phases to design a learning system. The first phase is learning. The second phase is a generalization test. The term generalization is borrowed from psychology. In neural network learning, a model is said to generalize well when it can produce correct input–output mapping for unseen test data that have not been used in the learning phase.

17.1.3 *Organization of the Chapter*

The reminder of this chapter is organized as follows. Section 17.2 introduces a number of learning algorithms in order to give a breadth of coverage of machine learning. Section 17.3 addresses evolution and learning. Three levels of evolution can be introduced in neural network learning: the evolution of weight, the evolution of architectures and the evolution of learning rules. Section 17.5 points out some promising areas in machine learning. Section 17.6 provides a guideline for implementing machine learning algorithms. Section 17.7 concludes with a summary of the chapter and a few remarks.

17.2 Overview of Learning Algorithms

This section explores the basic ideas and the principles of a number of learning algorithms that are used for real-world applications.

17.2.1 *Learning Decision Trees*

The task of inductive learning is to find a function h that approximates f given a collection of examples of f . The function h is called a hypothesis. An example is

a pair $(x, f(x))$, where x is the input, and $f(x)$ is the output of the function applied to x . In decision-tree learning, hypotheses are represented by decision trees.

A decision tree is a diagram representing a classification system or a predictive system. The structure of the system is a tree generated based on a sequence of simple questions. The answers to these questions trace a path down the tree. As a result, a decision tree is a collection of hierarchical rules that segment the data into groups, where a decision is made for each group. The hierarchy is called a tree, and each segment is called a node. The original segment that contains the entire data set is referred to as the root node of the tree. A node with all of its successors forms a branch of the tree. The terminal nodes are called leaves that return a decision, i.e. the predicted output value for the input. The output value can be either discrete or continuous. A classification tree is used to learn a discrete-valued function, while a regression tree is used to learn a continuous function. Most decision learning algorithms are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees.

A very effective decision learning algorithm, called ID3, was developed by Quinlan (1986). In ID3, classification trees are built by starting with the set of examples and an empty tree. An attribute test is chosen for the root of the tree, and examples are partitioned into disjoint subsets depending on the outcome of the test. The learning is then applied recursively to each of these disjoint subsets. The learning process stops when all the examples within a subset belong to the same class. At this learning stage, a leaf node is created and labeled with the class.

The method used to choose the attribute test is designed to minimize the depth of the final tree. The idea is to select the attribute that can lead to an exact classification of examples as far as possible. In ID3, a statistical property, called information gain, was introduced to measure how well a given attribute separates the examples according to their target classification.

For decision-tree learning, a learned classification tree has to predict what the correct classification is for a given example. Given a training set S , containing p positive examples and n negative examples, the entropy of S to this Boolean classification is

$$E(S) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}. \quad (17.2)$$

In information theory, the entropy of S gives an estimate of the information contained in a correct answer before any of the attributes have been tested. Information theory measures information content in bits. After a test on a single attribute A , attribute A divides the training set S into subsets S_i , $i = 1, \dots, v$, where A can have v distinct values. The information gain $G(S, A)$ of an attribute A , relative to a training set S , is defined as

$$G(S, A) = E(S) - \sum_{i=1}^v \frac{p_i + n_i}{p+n} E(S_i) \quad (17.3)$$

where each subset S_i has p_i positive examples and n_i negative examples. The second term in (17.3) is the expected value of entropy after S is partitioned using attribute A .

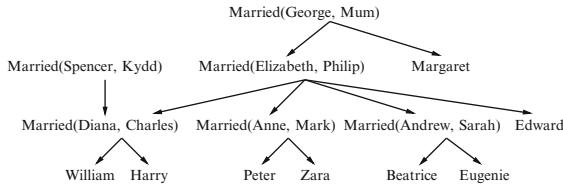


Fig. 17.2 A typical family tree

The expected entropy is the sum of entropies of each subset S_i , weighted by the fraction of examples in S_i . $G(S, A)$ is therefore the expected reduction in entropy caused by knowing the value of attribute A .

ID3 provides a simple and effective approach to decision-tree learning. However, for real-world applications, the algorithm needs to cope with problems such as a noisy data set, missing attribute values and attributes with continuous values. Dealing with these problems was studied with ID3's successor C4.5 (Quinlan 1993).

The first decision-tree learning system, called the Elementary Perceiver and Memorizer, was proposed by Feigenbaum (1961). It was studied as a cognitive-simulation model of human concept learning. The concept learning system developed by Hunt et al. (1966) used a heuristic look-ahead method to grow decision trees. ID3 (Quinlan 1986) introduced the information content as a heuristic search. The classification and regression tree system is a widely used statistical procedure for producing classification and regression (Breiman et al. 1984). Many practical issues of decision-tree induction can be found in C4.5, a decision-tree learning package by Quinlan (1993).

The advantages of decision-tree learning are its comprehensibility, fast classification and mature technology. However, by using only one attribute at each internal node, decision-tree learning can construct monothetic trees, which are limited to axis-parallel partitions of the instance space, rather than polythetic trees. Polythetic trees can use more than two attributes at each internal node, but are expensive to induce. The next section will introduce inductive logic programming, which combines inductive learning with the power of first-order representations.

17.2.2 Inductive Logic Programming

Inductive logic programming is a combination of knowledge-based inductive learning and logic programming (Russell and Norvig 2002). A general knowledge-based inductive learning is a kind of algorithm that satisfies the entailment constraint

$$\text{Background} \wedge \text{Hypothesis} \wedge \text{Descriptions} \models \text{Classifications}, \quad (17.4)$$

where *Descriptions* denote the conjunction of all the example classifications, and *Classifications* denote the conjunction of all the example classifications. Given the

Background knowledge and examples described by *Descriptions* and *Classifications*, the induction problem of knowledge-based inductive learning is to solve the entailment constraint (17.4) for the unknown *Hypothesis*.

In order to see how the background knowledge can be combined with the new hypothesis to explain examples, consider a problem of learning family relationships from examples in an extended family tree given in Fig. 17.2 (Russell and Norvig 2002).

The descriptions will be in the terms of Mother, Father, and Married relations and Male and Female properties, such as Father(Philip, Charles), Mother(Mum, Margaret), Married(Diana, Charles), Male(Philip), Female(Beatrice). Classifications depend on the target concept being learned. For learning the target concept of Grandfather, the complete set of Classifications contains $20 \times 20 = 400$ conjuncts of the form

$$\begin{aligned} & \text{Grandparent(Mum, Charles)} \quad \text{Grandparent(Elizabeth, Beatrice)} \dots \\ & \neg\text{Grandparent(Mum, Harry)} \quad \neg\text{Grandparent(Spencer, Peter)} \dots \end{aligned}$$

Without the background knowledge, inductive learning can find a possible *Hypothesis*:

$$\begin{aligned} \text{Grandparent}(x,y) \Leftrightarrow & [\exists z \ \text{Mother}(x,z) \wedge \text{Mother}(z,y)] \\ & \vee [\exists z \ \text{Mother}(x,z) \wedge \text{Father}(z,y)] \\ & \vee [\exists z \ \text{Father}(x,z) \wedge \text{Mother}(z,y)] \\ & \vee [\exists z \ \text{Father}(x,z) \wedge \text{Father}(z,y)]. \end{aligned} \quad (17.5)$$

With the help of the background knowledge represented by the sentence

$$\text{Parent}(x,y) \Leftrightarrow [\text{Mother}(x,y) \vee \text{Father}(x,y)]. \quad (17.6)$$

Hypothesis can be simply defined by

$$\text{Grandparent}(x,y) \Leftrightarrow [\exists z \ \text{Parent}(x,z) \wedge \text{Parent}(z,y)]. \quad (17.7)$$

By using background knowledge, we can reduce the size of hypotheses greatly.

There are two basic approaches to inductive logic programming: the top-down learning of refining a very general rule and the bottom-up learning of inverting the deductive process. A top-down approach will typically begin with a general clause and search the clause by adding literals so that only positive examples are entailed. First-order inductive learning (Quinlan 1990) is such a top-down induction algorithm.

Suppose the task is to learn a definition of *Grandfather*(x,y) predicate in the family tree shown in Fig. 17.2. Examples can be divided into positive and negative ones as in decision-tree learning: 12 positive examples are

$$\langle \text{George}, \text{Charles} \rangle, \langle \text{George}, \text{Anne} \rangle, \langle \text{George}, \text{Andrew} \rangle, \dots$$

and 388 negative examples are

$\langle \text{George}, \text{Spencer} \rangle, \langle \text{George}, \text{Kydd} \rangle, \langle \text{George}, \text{Elizabeth} \rangle, \dots$

First-order inductive learning constructs a set of clauses that must classify the positive examples while ruling out the negative examples. First-order inductive learning starts with the initial clause with $\text{Grandfather}(x,y)$ as the head, and an empty body

$$\Rightarrow \text{Grandfather}(x,y). \quad (17.8)$$

All examples are classified as positive by this clause. To specialize it, first-order inductive learning adds literals one at a time to the clause body. Look at two clauses constructed by such addition:

$$\text{Parent}(x,z) \Rightarrow \text{Grandfather}(x,y) \quad (17.9)$$

$$\text{Father}(x,z) \Rightarrow \text{Grandfather}(x,y). \quad (17.10)$$

Although both clauses agree with all of 12 positive examples, the first allows both fathers and mothers to be grandfathers and makes larger misclassification on negative examples. The second clause is chosen to be further specialized. By adding the single literal $\text{Parent}(z,y)$, first-order inductive learning can find

$$\text{Father}(x,z) \wedge \text{Parent}(z,y) \Rightarrow \text{Grandfather}(x,y), \quad (17.11)$$

which successfully classifies all the examples. This example gives a simple explanation how first-order inductive learning works. In real applications, first-order inductive learning generally has to search through a large number of unsuccessful clauses before finding the correct one.

Whereas first-order inductive learning ([Quinlan 1990](#)) is a top-down approach, Cigol (logic, spelled backwards), which [Muggleton and Buntine \(1988\)](#) developed for inductive logic programming, worked bottom-up. Cigol incorporated a slightly incomplete version of inverse resolution and was capable of generating new predicates. A hybrid (top-down and bottom-up) approach was chosen in Progol ([Muggleton 1995](#)) that inverse entailment and had been applied to a number of practical problems. A large collection of papers on inductive logic programming can be found in [Lavrač and Džeroski \(1994\)](#).

Inductive logic programming provides a practical approach to the general knowledge-based inductive learning problem. Its strengths lie in its firm theoretical foundations, richer hypothesis representation language, and explicit use of background knowledge. The limitations of inductive logic programming are its weak numeric representations and large search spaces.

17.2.3 Bayesian Learning

In practice, there are cases when more than one hypothesis satisfy a given task. Because it is not certain how those hypotheses perform on unseen data, it is hard to choose the best hypothesis. Bayesian learning gives a probabilistic framework for

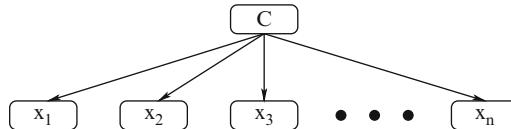


Fig. 17.3 The naive Bayes model

justification. By calculating explicit probabilities for hypotheses, Bayesian learning provides a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

Let \mathbf{d} represent all the data, and H the set of all the hypotheses h_i . The probability of each hypothesis with observed \mathbf{d} can be calculated by Bayes' rule:

$$P(h_i | \mathbf{d}) = \alpha P(\mathbf{d} | h_i)P(h_i), \quad (17.12)$$

where $P(h_i)$ is the prior probability, $P(\mathbf{d} | h_i)$ denotes the probability of observed \mathbf{d} given h_i , and $P(h_i | \mathbf{d})$ is the posterior probability of h_i .

A practical Bayesian learning used in machine learning is the naive Bayes model shown in Fig. 17.3, where each instance \mathbf{x} is described by a conjunction of attribute values $\langle x_1, x_2, \dots, x_n \rangle$. In this model, the class variable C is the root, and the attribute values \mathbf{x} are leaves.

According to (17.12), the probability of each class from a set of S is given by

$$P(C | x_1, x_2, \dots, x_n) = \alpha P(x_1, x_2, \dots, x_n | C)P(C). \quad (17.13)$$

In the naive Bayes model, a simplified assumption is made that the attributes are conditionally independent of each other given the class. That is, the probability of the observed conjunction x_1, x_2, \dots, x_n is just the product of probabilities for the individual attributes:

$$P(x_1, x_2, \dots, x_n | C) = \prod_i P(x_i | C). \quad (17.14)$$

From (17.13) and (17.14), the naive Bayes model makes the prediction by choosing the most likely class:

$$C_{NB} = \operatorname{argmax}_{C \in S} P(C) \prod_i P(x_i | C), \quad (17.15)$$

where C_{NB} denotes output class by the naive Bayes model.

Consider a medical diagnosis problem with three possible diagnoses (*well*, *cold*, *allergy*) based on three symptoms (*sneeze*, *cough*, *fever*). In this example, there are three attributes in which x_1 can be *sneeze* or *not sneeze*, x_2 *cough* or *not cough*, and x_3 *fever* or *not fever*, and three classes: *well*, *cold* and *allergy*. The probabilities for the three attributes and three prior class probabilities are given in Table 17.1.

Table 17.1 An example for the naive Bayes model

Diagnosis	Well	Cold	Allergy
$P(C)$	0.9	0.05	0.05
$P(\text{sneeze} C)$	0.1	0.9	0.9
$P(\text{cough} C)$	0.1	0.8	0.7
$P(\text{fever} C)$	0.01	0.7	0.4

Given a new $\mathbf{x} = \langle \text{sneeze}, \text{cough}, \text{not fever} \rangle$, which class of diagnoses is it mostly like to be? First, the posterior probability $P(\text{well} | \text{sneeze}, \text{cough}, \text{not fever})$ of *well*, *cold* and *allergy* are calculated by the product of $P(\text{sneeze} | \text{well})$, $P(\text{cough} | \text{well})$, $P(\text{not fever} | \text{well})$, and $P(\text{well})$:

$$\begin{aligned} P(\text{well} | \text{sneeze}, \text{cough}, \text{not fever}) &= 0.1 \times 0.1 \times (1 - 0.01) \times 0.9 \\ &= 0.00891. \end{aligned} \quad (17.16)$$

Similarly we can obtain the posterior probability of *cold*:

$$P(\text{cold} | \text{sneeze}, \text{cough}, \text{not fever}) = 0.216 \quad (17.17)$$

and the posterior probability of *allergy*:

$$P(\text{allergy} | \text{sneeze}, \text{cough}, \text{not fever}) = 0.378. \quad (17.18)$$

Finally, we compare three posterior probabilities and generate output class *allergy* because the probability of allergy for the data $\mathbf{x} = \langle \text{sneeze}, \text{cough}, \text{not fever} \rangle$ is the largest one.

The naive Bayes model has been compared with C4.5 on 28 benchmark tasks ([Domingos and Pazzani 1996](#)). The results show that the naive Bayes model performs surprisingly well in a wide range of applications. Except for a few domains where the naive Bayes model performs poorly, it is comparable to or better than C4.5.

This section just uses the naive Bayes model to introduce the idea of Bayesian learning. [Heckerman \(1998\)](#) gives an excellent introduction on general learning with Bayesian networks. Bayesian learning has had successful applications in pattern recognition and information retrieval. Algorithms based on Bayesian learning won the 1997 and 2001 KDD Cup data mining competitions ([Elkan 1997; Cheng et al. 2002](#)). Experimental comparisons between Bayesian learning, decision-tree learning and other algorithms have been made on a wide range of applications ([Michie et al. 1994](#)).

17.2.4 Reinforcement Learning

Reinforcement learning concerns learning how to map situations to actions so as to maximize a numerical reward signal ([Sutton and Barto 1998](#)). Unlike supervised learning, the machine is not told which actions to take but has to discover which

actions yield the most reward by trying them. In the most practical cases, actions may affect both the immediate reward and the next situation and thus all subsequent rewards. Trial-and-error search and delayed reward are the two most important unique characteristics of reinforcement learning.

A central and novel idea of reinforcement learning is temporal-difference learning (Sutton and Barto 1998). Temporal-difference learning is a combination of Monte Carlo ideas and dynamic programming ideas. Like Monte Carlo methods, temporal-difference learning methods can learn directly from the raw experience without a model of the environment's dynamics. Like dynamic programming methods, temporal-difference learning methods update estimates based in part on other learned estimates, without waiting for a final outcome. Temporal-difference learning works because it is possible to make local improvements. At every point in the state space, the Markov property allows actions to be chosen based only on knowledge about the current state and the states reachable by taking the actions available at that state.

Temporal-difference learning methods fall into two classes: on-policy and off-policy (Sutton and Barto 1998). One of the most important breakthroughs in reinforcement learning was the development of an off-policy temporal-difference learning control algorithm known as Q -learning. The learned action-value function $Q(s, a)$ directly approximates the optimal action-value function, independent of the policy being followed. The major steps of Q -learning are (Sutton and Barto 1998):

1. Initialize $Q(s, a)$ values arbitrarily.
2. Initialize the environment.
3. Choose action a using the policy derived from $Q(s, a)$ (e.g., ϵ -greedy).
4. Take action a ; Observe reward r and the next state s' .
5. Update the $Q(s, a)$ as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]. \quad (17.19)$$

6. Let $s \leftarrow s'$. Go to the next step if the state s is a terminal state. Otherwise, go to Step 3.
7. Repeat Steps 2–6 for a certain number of episodes.

The Sarsa learning algorithm is an on-policy temporal-difference learning method in which the action-value function Q is updated after every transition from a nonterminal state. The major steps of Sarsa learning are (Sutton and Barto 1998):

1. Initialize $Q(s, a)$ values arbitrarily.
2. Initialize the environment.
3. Choose action a using the policy derived from $Q(s, a)$ (e.g., ϵ -greedy).
4. Take action a ; Observe reward r and the next state s' ; Choose the next action a' using the policy derived from Q (e.g. ϵ -greedy).
5. Update the $Q(s, a)$ as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]. \quad (17.20)$$

6. Let $s \leftarrow s'$ and $a \leftarrow a'$. Go to the next step if the state s is a terminal state. Otherwise, go to Step 3.
7. Repeat Steps 2–6 for a certain number of episodes.

Sutton and Barto assessed a Sarsa learning example ([Sutton and Barto 1998](#)). The results showed that the online performance of Q -learning is worse than that of Sarsa learning.

The strengths of reinforcement learning come from its firm theoretical foundation, its ability to solve broad tasks, and its easy usage of background knowledge. Work in reinforcement learning dates back to the earliest days of machine learning when Turing proposed the reinforcement learning approach ([Turing 1950](#)), and Samuel developed his famous checkers learning program that contained most of the modern ideas of reinforcement learning, including temporal differencing and function approximation ([Samuel 1959](#)). Three threads contributed towards the modern field of reinforcement learning. The first thread is about learning by trial and error and had its origin in the psychology of animal learning, which led to the popularity of reinforcement learning in the early 1980s. The second thread arose from the problem of optimal control and its solution using value functions and dynamic programming. The third thread concerns temporal-difference methods. The survey by [Kaelbling et al. \(1996\)](#) provides a good starting point in the literature. The text *Reinforcement Learning: An Introduction* by Sutton and Barto, two of the field's pioneers, shows architectures and algorithms of reinforcement learning in the context of learning, planning and acting ([Sutton and Barto 1998](#)).

17.2.5 Neural Networks

Artificial neural networks, commonly referred to as neural networks, try to simulate biological brains. However, neural networks have been simplified greatly from biological brains. A neural network is a parallel computational system consisting of many processing elements connected with each other in a certain way in order to perform a task. Neural networks have gained popularity because they are adaptive, robust, fault tolerant, noise tolerant, and massively parallel.

Among the many tasks that neural networks perform, the most important one is learning. A neural network can improve its performance via learning. Perceptron learning is one of the earliest learnings developed for neural networks ([Rosenblatt 1962](#)). Perceptrons are often used to refer to feed-forward neural networks consisting of McCulloch–Pitts (MP) neurons ([McCulloch and Pitts 1943](#)):

$$y_i = \text{sgn} \left(\sum_j w_{ij} x_j - \theta_i \right) \quad (17.21)$$

where the w_{ij} are called *weights* (synapses) and θ is the *threshold*. The x_j and y_i are input and output. The signum function $\text{sgn}(x)$ is defined as

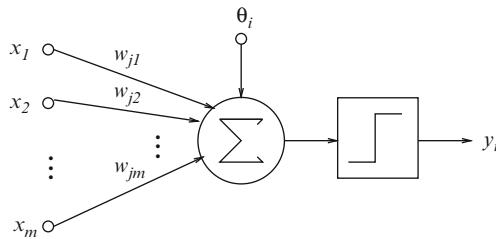


Fig. 17.4 Nonlinear model of a neuron

$$\text{sgn}(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases} \quad (17.22)$$

It is also known as the threshold function or Heaviside function and is described in Fig. 17.4.

After presenting each example to a perceptron that has one layer of neurons, perceptron learning adjusts the weights until the weights converge (i.e. $\Delta w_j(t) = 0$):

$$w_j(t+1) = w_j(t) + \Delta w_j(t), \quad (17.23)$$

where

$$\Delta w_j(t) = \eta(y^p - O^p)x_j^p, \quad (17.24)$$

where η is the learning rate, x_j^p is the j th input of the p th example, y^p is the target (desired) output of the p th example, and O^p is the actual output of the p th example:

$$O^p = \text{sgn} \left(\sum_j w_j x_j^p - \theta \right). \quad (17.25)$$

The convergence theorem of perceptron learning states that if there exists a set of weights for a perceptron which solves a problem correctly, the perceptron learning rule will find them in a finite number of iterations (Rosenblatt 1962). If a problem is linearly separable, then the perceptron learning rule will find a set of weights in a finite number of iterations that solves the problem correctly. A pair of linearly separable patterns means that the patterns to be classified must be sufficiently separated from each other to ensure that the decision surface consists of a hyperplane.

The perceptron learning rule, $\Delta w_j(t) = \eta(y^p - O^p)x_j^p$, is related to the Hebbian learning rule (Hebb 1949). Hebb's postulate of learning (Hebb 1949) states that

When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.

In other words, if two neurons on either side of a synapse (connection) are activated simultaneously (i.e. synchronously), then the strength of that synapse is selectively increased. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

It is clear from perceptron learning that the algorithm tries to minimize the difference between the actual and desired output. We can define an error function to represent such a difference:

$$E(\mathbf{w}) = \frac{1}{2} \sum_p (y^p - O^p)^2 \quad (17.26)$$

or

$$E(\mathbf{w}) = \frac{1}{2N} \sum_p (y^p - O^p)^2, \quad (17.27)$$

where N is the number of patterns. The second error function above is called the mean square error. Learning minimizes this error by adjusting weights \mathbf{w} .

One advantage of introducing the error function is that it can be used for any type of transfer function, discrete or continuous. The aim of learning is to adjust \mathbf{w} such that the error E is minimized, i.e. the network output is as close to the desired output as possible. There exist mathematical tools and algorithms which tell us how to minimize the error function E , such as the gradient descent algorithm which is based on partial derivatives. Given a set of training examples, $\{(x_1^p, \dots, x_m^p; y_1^p, \dots, y_n^p)\}_p$, the gradient descent learning algorithm can be summarized as follows:

1. Construct a neural network with m inputs and n outputs.
2. Select learning rate η and the gain parameter a .
3. Generate initial weights at random in a small range, e.g. $[-0.5, 0.5]$. Note that thresholds are regarded as weights here.
4. While the neural network has not converged do: For each training example p ,
 - (a) Compute O_i^p . ($O_i^p = f(u_i)$)
 - (b) Compute $\delta_i^p = (y_i^p - O_i^p) f'(u_i)$, where $f'(u_i) = af(u_i)(1 - f(u_i))$ if the transfer function is

$$f(u_i) = \frac{1}{1 + \exp(-au_i)}, \quad (17.28)$$

where a is a parameter determined by the user.

- (c) Compute $\Delta w_{ij} = \eta \delta_i^p x_i^p$.
- (d) Update w_{ij} for all i, j . (All weights will be updated.)

There are two modes of the gradient descent learning algorithm. One is the sequential mode of training which is also known as online, pattern or stochastic mode. In this mode, weights are updated after the presentation of each example. The other is the batch mode of training in which weights are updated only after the complete presentation of all examples in the training set, i.e. only after each epoch.

The idea of the gradient descent learning algorithm for a single-layer neural network can be generalized to find weights for multilayer neural networks. Multilayer feedforward neural networks can solve nonlinear problems. In fact, there are mathematical theorems which show that multilayer feedforward neural networks can approximate any input–output mapping. The backpropagation algorithm can be used to train multilayer feedforward neural networks (Rumelhart et al. 1986). Its forward pass propagates the activation values from input to output. Its backward

pass propagates the errors from output to input. Backpropagation is still a gradient descent algorithm. It uses gradient information to figure out how the weights should be adjusted so that the output error can be reduced. Mathematically, backpropagation uses the chain rule to figure out how to change weights in order to minimize the error.

Consider a network with M layers $m = 1, 2, \dots, M$ and use V_i^m to represent the output of the i th unit in the m th layer. $V_i^0 = x_i$ is the i th input. Backpropagation can be described as follows:

1. Initialize the weights to small random values.
2. Choose a pattern and apply it to the input layer so that $V_i^0 = x_i^p$.
3. Propagate the signal forwards through the network using

$$V_i^m = f(u_i^m) = f\left(\sum_j w_{ij}^m V_j^{m-1}\right) \quad (17.29)$$

for each i and m until the final outputs V_i^M have all been calculated.

4. Compute the deltas for the output layer:

$$\delta_i^M = f'(u_i^M)(y_i^p - V_i^M). \quad (17.30)$$

5. Compute the deltas for the preceding layers by propagating errors backwards:

$$\delta_i^{m-1} = f'(u_i^{m-1}) \sum_j w_{ij}^m \delta_j^m \quad (17.31)$$

for $m = M, M-1, \dots, 2$.

6. Update the weights according to

$$w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ij}, \quad (17.32)$$

where

$$\Delta w_{ij} = \eta \delta_i^m V_j^{m-1}. \quad (17.33)$$

7. Goto step 2 and repeat for the next pattern. The algorithm stops when no weight changes were made for a complete epoch or the maximum number of iterations has been reached.

The study of neural networks started with the work of MP neuron models proposed by McCulloch and Pitts (1943). The Hebbian learning rule was studied by Hebb (1949). Rosenblatt (1962) proposed perceptrons and proved the perceptron convergence theory. After Minsky and Papert's book (Minsky and Papert 1969) which showed the limitation of single-layer perceptrons, the field of neural networks was almost deserted during the 1970s. Then Hopfield published a series of papers on Hopfield networks that used the idea of an energy function to formulate a new way of understanding the computation performed by recurrent networks with symmetric synaptic connections (Hopfield 1982; Hopfield and Tank 1985). The two-volume “bible” *Parallel Distributed Processing: Explorations in*

the Microstructures of Cognition, edited by Rumelhart and McClelland attracted a great deal of attention ([Rumelhart and McClelland 1986](#)). In the mid-1980s, the field of neural networks really took off.

Neural networks have been applied to solve a wide range of problems such as pattern recognition and classification, time-series prediction, function approximation, system identification, and control. Neural network applications often include two phases. The first phase is learning. The task performed by a neural network is often represented as a set of examples. The neural network is expected to learn more general concepts from these examples. The steps involved include:

1. Select a neural network architecture, where the number of input and output nodes are determined by the task. Hidden nodes and network connectivity need to be designed mostly by trial and error.
2. Train the network using a suitable training algorithm.

The second phase is a generalization test. After the neural network is trained, it will be tested with new (never seen before) examples to see how well it generalizes.

17.2.6 Evolutionary Learning

Evolutionary learning includes many topics, such as learning classifier systems, evolutionary neural networks, evolutionary fuzzy logic systems, co-evolutionary learning and self-adaptive systems. The primary goal of evolutionary learning is the same as that of machine learning in general. Evolutionary learning can be regarded as the evolutionary computation approach to machine learning. It has been used in the framework of supervised learning, reinforcement learning and unsupervised learning, although it appears to be most promising as a reinforcement learning method. Evolutionary computation encompasses major branches, i.e. evolution strategies, evolutionary programming, genetic algorithms and genetic programming, due largely to historical reasons. At the philosophical level, they differ mainly in the level at which they simulate evolution. At the algorithmic level, they differ mainly in their representations of potential solutions and their operators used to modify the solutions. From a computational point of view, representation and search are two key issues.

Evolution strategies were first proposed by Rechenberg and Schwefel in the mid-1960s for numerical optimization. Real-valued vectors are used to represent individuals. Evolution strategies use both recombination and self-adaptive mutations. The original evolution strategy did not use populations. A population was introduced into evolution strategies later ([Schwefel 1981, 1995](#)).

Evolutionary programming was first proposed by Fogel et al. in the mid-1960s for simulating intelligence ([Fogel et al. 1966](#)). Finite-state machines were used to represent individuals, although real-valued vectors have always been used in numerical optimization. Search operators (mutations only) are applied to the phenotypic representation of individuals. There is no recombination in evolutionary programming. Tournament selection is often used in evolutionary programming.

Genetic algorithms and genetic programming are introduced in Chaps. 4 and 5 of this book, respectively. Although genetic algorithms, evolutionary programming, evolution strategies and genetic programming are different, they are all variants of population-based generate-and-test algorithms:

Generate: Mutate and/or recombine individuals in a population.

Test: Select the next generation from the parents and offsprings.

They share more similarities than differences. A better and more general term to use is evolutionary algorithms. All evolutionary algorithms have two prominent features which distinguish themselves from other search algorithms. First, they are all population-based. Secondly, there are communications and information exchange among individuals in a population. Such communications and information exchange are the result of selection and/or recombination in evolutionary algorithms. A general framework of evolutionary algorithms can be summarized as follows:

1. Generate the initial population $P(0)$ at random, and set $i \leftarrow 0$;
2. Repeat
 - (a) Evaluate the fitness of each individual in $P(i)$;
 - (b) Select parents from $P(i)$ based on their fitness in $P(i)$;
 - (c) Generate offspring from the parents using crossover and mutation to form $P(i+1)$;
 - (d) $i \leftarrow i + 1$;
3. Until halting criteria are satisfied

where the search operators are also called genetic operators for genetic algorithms. They are used to generate offspring (new individuals) from parents (existing individuals).

Learning classifier systems, also known as classifier systems, are probably the oldest and best known evolutionary learning systems, although they did not work very well in their classical form. Some of the recent systems have improved this situation. Due to its historical importance, a brief introduction to the classical learning classifier systems will be introduced here.

Learning classifier systems are a particular class of message-passing, rule-based systems. They can also be regarded as a type of adaptive expert system that uses a knowledge base of production rules in a low-level syntax that can be manipulated by a genetic algorithm. In a classifier system, each low-level rule is called a classifier. A general operational cycle for the classifier system is as follows:

1. Allow the detectors (input interface) to code the current environment status and place the resulting messages on the message list.
2. Determine the set of classifiers that are matched by the current messages.
3. Resolve conflicts caused by limited message list size or contradictory actions.
4. Remove those messages which match the conditions of firing classifier from the message list.
5. Add the messages suggested by the firing messages to the list.

6. Allow the effectors (output interface) that are matched by the current message list to take actions in the environment.
7. If a payoff signal is received from the environment, assign credit to the classifiers.
8. Goto Step 1.

A genetic algorithm is used in classifier systems to discover new classifiers by crossover and mutation. The strength of a classifier updated by the credit assignment scheme is used as its fitness. A classifier's strength is based on its average usefulness in the context in which it has been tried previously. Credit assignment is a very difficult task because credit must be assigned to early-acting classifiers that set the stage for a sequence of actions leading to a favorable situation. The most well known credit assignment is the bucket brigade algorithm which uses metaphors from economics.

For a classifier called middleman, its suppliers are those classifiers that have sent messages satisfying its conditions, and its consumers are those classifiers that have conditions satisfied by its message and have won their competition in turn. When a classifier wins in competition, its bid is actually apportioned to its suppliers, increasing their strengths by the amounts apportioned to them. At the same time, because the bid is treated as a payment for the right to post a message, the strength of the winning classifier is reduced by the amount of its bid. Should the classifier bid but not win, its strength remains unchanged and its suppliers receive no payment. Winning classifiers can recoup their payments from either winning consumers or the environment payoff.

The genetic algorithm is only applied to the classifiers after certain number of operational cycles in order to approximate strengths better. There are two approaches to classifier systems; the Michigan approach and the Pitt approach. For the Michigan approach, each individual in a population is a rule. The whole population represents a complete classifier system. For the Pitt approach, each individual in a population represents a complete classifier system. The whole population includes a number of competing classifier systems.

17.3 Learning and Evolution

Learning and evolution are two fundamental forms of adaptation. There has been a great interest in combining learning and evolution with neural networks in recent years.

17.3.1 Evolutionary Neural Networks

Evolutionary neural networks refer to a special class of neural networks in which evolution is another fundamental form of adaptation in addition to learning (Yao 1991, 1993a, 1994, 1995). Evolutionary algorithms are used to perform various

tasks, such as connection weight training, architecture design, learning rule adaptation, input feature selection, connection weight initialization and rule extraction from neural networks. One distinct feature of evolutionary neural networks is their adaptability to a dynamic environment. In other words, evolutionary neural networks can adapt to an environment as well as to changes in the environment. The two forms of adaptation, i.e., evolution and learning in evolutionary neural networks, make their adaptation to a dynamic environment much more effective and efficient. In a broader sense, evolutionary neural networks can be regarded as a general framework for adaptive systems, i.e. systems that can change their architectures and learning rules appropriately without human intervention.

Evolution has been introduced into neural networks at roughly three different levels:

- Connection weights
- Architectures and
- Learning rules.

17.3.1.1 The Evolution of Connection Weights

The evolution of connection weights introduces an adaptive and global approach to training, especially in the reinforcement learning and recurrent network learning paradigm where gradient-based training algorithms often experience great difficulties.

One way to overcome gradient-descent-based training algorithms' shortcomings is to adopt evolutionary neural networks, i.e. to formulate the training process as the evolution of connection weights in the environment determined by the architecture and the learning task. Evolutionary algorithms can then be used effectively in the evolution to find a near-optimal set of connection weights globally without computing gradient information. The fitness of a neural network can be defined according to different needs. Two important factors which often appear in the fitness (or error) function are the error between target and actual outputs and the complexity of the neural network. Unlike in the gradient-descent-based case, the fitness (or error) function does not have to be differentiable or even continuous since evolutionary algorithms do not depend on gradient information. Because evolutionary algorithms can treat large, complex, nondifferentiable and multimodal spaces, which are the typical case in the real world, considerable research and application has been conducted on the evolution of connection weights.

The evolutionary approach to weight training in neural networks consists of two major phases. The first phase is to decide the representation of connection weights, i.e. whether it is to be in the form of binary strings or not. The second one is the evolutionary process simulated by an evolutionary algorithm, in which search operators such as crossover and mutation have to be decided on in conjunction with the representation scheme. Different representations and search operators can lead to

quite different training performance. A typical cycle of the evolution of connection weights is shown as follows (Yao 1999):

1. Decode each individual (genotype) in the current generation into a set of connection weights and construct a corresponding neural network with weights.
2. Evaluate each neural network by computing its total mean square error between actual and target outputs. Other error functions can also be used. The fitness of an individual is determined by the error. The higher the error, the lower the fitness. The optimal mapping from the error to the fitness is a problem dependent. A regularization term may be included in the fitness function to penalize large weights.
3. Select parents for reproduction based on their fitness.
4. Apply genetic operators, such as crossover and/or mutation, to parents to generate offspring, which form the next generation.

The evolution stops when the fitness is greater than a predefined value (i.e. the training error is smaller than a certain value) or the population has converged.

17.3.1.2 The Evolution of Architectures

The evolution of architectures enables neural networks to adapt their topologies to different tasks without human intervention and thus provides an approach to automatic neural network design as both neural network connection weights and structures can be evolved.

Architecture design is crucial in the successful application of neural networks because the architecture has significant impact on a network's information processing capabilities. Given a learning task, a neural network with only a few connections and linear nodes may not be able to perform the task at all due to its limited capability, while a neural network with a large number of connections and nonlinear nodes may overfit noise in the training data and fail to have good generalization ability.

Currently, architecture design is still very much a human expert's job. It depends heavily on expert experience and a tedious trial-and-error process. There is no systematic way to design a near-optimal architecture for a given task automatically. Design of the optimal architecture for a neural network can be formulated as a search problem in the architecture space where each point represents an architecture. Given some performance (optimality) criteria, e.g. lowest training error, lowest network complexity, etc., about architectures, the performance level of all architectures forms a discrete surface in the space. The optimal architecture design is equivalent to finding the highest point on this surface.

Similar to the evolution of connection weights, two major phases involved in the evolution of architectures are the genotype representation scheme of architectures and the evolutionary algorithm used to evolve neural network architectures. One of the key issues in encoding neural network architectures is to decide how much information about an architecture should be encoded in the chromosome. At one extreme, all the details, i.e. every connection and node of an architecture, can be specified

by the chromosome. This kind of representation scheme is called direct encoding. At the other extreme, only the most important parameters of an architecture, such as the number of hidden layers and hidden nodes in each layer, are encoded. Other details about the architecture are left to the training process to decide. This kind of representation scheme is called indirect encoding. After a representation scheme has been chosen, the evolution of architectures can progress according to the cycle as follows (Yao 1999):

1. Decode each individual in the current generation into an architecture. If the indirect encoding scheme is used, further detail of the architecture is specified by some developmental rules or a training process.
2. Train each neural network with the decoded architecture by a pre-defined learning rule (some parameters of the learning rule could be learned during training) starting from different sets of random initial weights and, if any, learning parameters.
3. Compute the fitness of each individual (encoded architecture) according to the above training result and other performance criteria such as the complexity of the architecture.
4. Select parents from the population based on their fitness.
5. Apply genetic operators to the parents and generate offspring which form the next generation.

The cycle stops when a satisfactory neural network is found.

An automatic system, EPNet (Yao and Liu 1997, 1998), based on evolutionary programming has been developed for simultaneous evolution of neural network architectures and connection weights. EPNet relies on a number of mutation operators to modify architectures and weights. Behavioral (i.e. functional) evolution, rather than genetic evolution, is emphasized in EPNet. A number of techniques were adopted to maintain the behavioral link between a parent and its offspring (Yao and Liu 1997). Figure 17.5 shows the main structure of EPNet.

EPNet uses rank-based selection (Yao 1993b) and five mutations: hybrid training, node deletion, connection deletion, connection addition and node addition (Yao and Liu 1997). EPNet uses a hybrid algorithm to train the neural network for a fixed number of epochs. Such training does not guarantee the convergence of neural network learning. Hence the training is partial. The other four mutations are used to grow and prune hidden nodes and connections.

The five mutations are attempted sequentially. If one mutation leads to a better offspring, it is regarded as successful. No further mutation will be applied. Otherwise the next mutation is attempted. The motivation behind ordering mutations is to encourage the evolution of compact neural networks without sacrificing generalization. A validation set is used in EPNet to measure the fitness of an individual, and another validation set to stop training in the final step. EPNet has been tested extensively on a number of benchmark problems, and very compact neural networks with good generalization ability have been evolved (Yao and Liu 1997).

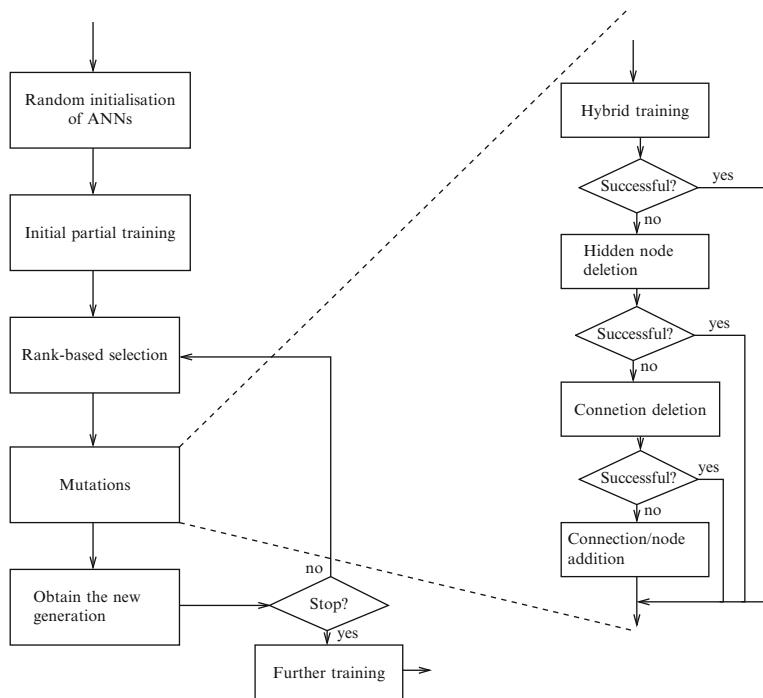


Fig. 17.5 The main structure of EPNet

17.3.2 The Evolution of Learning Rules

The evolution of learning rules can be regarded as a process of *learning to learn* in neural networks where the adaptation of learning rules is achieved through evolution. It can also be regarded as an adaptive process of automatic discovery of novel learning rules.

The relationship between evolution and learning is extremely complex. Various models have been proposed, but most of them deal with the issue of how learning can guide evolution and the relationship between the evolution of architectures and that of connection weights (Yao 1999). Research into the evolution of learning rules is still in its early stages. This research is important not only in providing an automatic way of optimizing learning rules and in modeling the relationship between learning and evolution, but also in modeling the creative process since newly evolved learning rules can deal with a complex and dynamic environment. This research will help us to understand better how creativity can emerge in artificial systems, like neural networks, and how to model the creative process in biological systems. A typical cycle of the evolution of learning rules can be described as follows (Yao 1999):

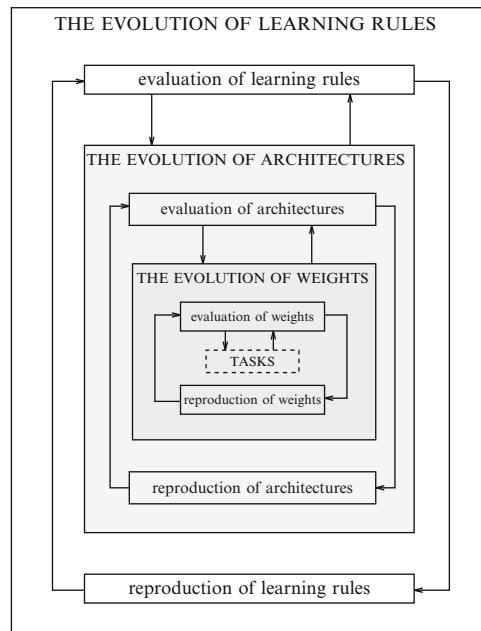


Fig. 17.6 A general framework for evolutionary neural networks

1. Decode each individual in the current generation into a learning rule.
2. Construct a set of neural networks with randomly generated architectures and initial weights, and train them using the decoded learning rules.
3. Calculate the fitness of each individual (encoded learning rule) according to the average training results.
4. Select parents from the current generation according to their fitness.
5. Apply search operators to parents to generate offspring which form the next generation.

The iteration stops when the population converges or a predefined maximum number of iterations has been reached.

17.3.3 A General Framework for Evolutionary Neural Networks

Figure 17.6 illustrates a general framework for evolutionary neural networks (Yao 1999). The evolution of connection weights proceeds at the lowest level on the fastest time scale in an environment determined by an architecture, a learning rule, and learning tasks. There are, however, two alternatives to decide the level of the evolution of architectures and that of learning rules: either the evolution of architectures is at the highest level and that of learning rules at the lower level or vice versa. The lower the level of evolution, the faster the time scale.

17.4 Ensemble Learning

17.4.1 Bias–Variance Trade-Off

In order to minimize $R(\mathbf{w})$ in Eq. (17.1) with an unknown probability distribution Pr , a training set $D = \{(\mathbf{x}(1), y(1)), \dots, (\mathbf{x}(N), y(N))\}$ is selected and the function $F(\mathbf{x}, \mathbf{w})$ is chosen to minimize $\sum_{i=1}^N E[(F(\mathbf{x}(i), \mathbf{w}) - y(i))^2]$. Thus, the training set leads to a function $F(\mathbf{x}, \mathbf{w})$ that depends on D . To be explicit about dependence on the training set D , we rewrite the function $F(\mathbf{x}, \mathbf{w})$ as $F(\mathbf{x}, D)$.

The training set D can be chosen randomly using (\mathcal{E}, Pr) by choosing N independent samples from \mathcal{E} . This can be described by a new probability space $(\mathcal{E}^{(N)}, Pr^{(N)})$ which consists of all the training sets D of given size N . Let E_D denote expectation over this space. Let E with no subscript denote the expectation over (\mathcal{E}, Pr) . Since the function $F(\mathbf{x}, D)$ is dependent on D , it differs from term to term in the sum for E_D . Consider the mean-squared error of the function $F(\mathbf{x}, D)$, which is defined by

$$E_D[(F(\mathbf{x}, D) - y)^2].$$

Taking expectations with respect to the training set D , we obtain the well-known separation of the mean-squared error (Geman et al. 1992)

$$\begin{aligned} E_D[(F(\mathbf{x}, D) - y)^2] &= E_D[F(\mathbf{x}, D)^2] - 2yE_D[F(\mathbf{x}, D)] + y^2 \\ &= (E_D[F(\mathbf{x}, D)] - y)^2 \\ &\quad + E_D[F(\mathbf{x}, D)^2] - (E_D[F(\mathbf{x}, D)])^2 \\ &= (E_D[F(\mathbf{x}, D)] - y)^2 \\ &\quad + var_D(F(\mathbf{x}, D)), \end{aligned} \tag{17.34}$$

where we have made use of the fact that y has constant expectation with respect to D , and variance definition

$$\begin{aligned} var_D(F(\mathbf{x}, D)) &= E_D[(F(\mathbf{x}, D) - E_D[F(\mathbf{x}, D)])^2] \\ &= E_D[F(\mathbf{x}, D)^2] - (E_D[F(\mathbf{x}, D)])^2. \end{aligned} \tag{17.35}$$

The first term $(E_D[F(\mathbf{x}, D)] - y)^2$ in the right-hand side of Eq. (17.34) represents the *bias* of the approximating function $F(\mathbf{x}, D)$. The bias measures how much the average function value at \mathbf{x} deviates from y . The second term $var_D(F(\mathbf{x}, D))$ represents the *variance* of the approximating function $F(\mathbf{x}, D)$. The variance measures how much the function values at \mathbf{x} vary from one training set to another.

Accordingly, Eq. (17.34) states that the expected mean-square value consists of the sum of two terms: bias and variance. Note that neither is negative. To achieve good performance, both the bias and the variance of the approximating function $F(\mathbf{x}, D)$ should be small.

If an allowed function $F(\mathbf{x}, D)$ is too simple, it will not be capable of capturing some of the aspects of the data. In particular, for a particular pair (\mathbf{x}, y) , there may be a general tendency to overestimate or a general tendency to underestimate. Both tendencies will make bias large. On the other hand, if an allowed function $F(\mathbf{x}, D)$

is too complex, it may be able to implement numerous solutions that are consistent with the training data, but most of these are likely to be poor approximations to data different from the training data. In other words, for any particular pair (\mathbf{x}, y) , a wide range of values of $F(\mathbf{x}, D)$, i.e. a large variance, may be obtained as the training set D varies.

There is usually a trade-off between bias and variance in the case of a training set with finite size (Geman et al. 1992): attempts to decrease bias by introducing more parameters often tend to increase variance; attempts to reduce variance by reducing parameters often tend to increase bias.

17.4.1.1 Bias–Variance–Covariance Trade-Off

There are many approaches to dealing with the bias–variance trade-off in neural network field. Given the training data set $D = \{(\mathbf{x}(1), y(1)), \dots, (\mathbf{x}(N), y(N))\}$, this section considers estimating y by forming a neural network ensemble whose output is a simple averaging of outputs $F_i(\mathbf{x}, D)$ of a set of neural networks. All the individual networks in the ensemble are trained on the same training data set D :

$$F(\mathbf{x}, D) = \frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D), \quad (17.36)$$

where M is the number of individual networks in the neural network ensemble. Taking expectations with respect to the training set D , the expected mean-squared error of the neural network ensemble can be written in terms of individual network output:

$$E_D [(F(\mathbf{x}, D) - y)^2] = E_D \left[\left(\frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D) - y \right)^2 \right]. \quad (17.37)$$

By use of Eq. (17.34), the right-hand side in (17.37) can be written as

$$\begin{aligned} E_D \left[\left(\frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D) - y \right)^2 \right] &= \left(E_D \left[\frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D) \right] - y \right)^2 \\ &\quad + var_D \left(\frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D) \right), \end{aligned} \quad (17.38)$$

where the first term in the right-hand side of (17.38) is the bias of the neural network ensemble, and the second term is the variance of the neural network ensemble. The variance of the neural network ensemble may be expressed as the sum of two terms:

$$\begin{aligned} var_D \left(\frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D) \right) &= E_D \left[\left(\frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D) - E_D \left[\frac{1}{M} \sum_{i=1}^M F_i(\mathbf{x}, D) \right] \right)^2 \right] \\ &= E_D \left[\frac{1}{M^2} \left(\sum_{i=1}^M (F_i(\mathbf{x}, D) - E_D[F_i(\mathbf{x}, D)]) \right)^2 \right] \end{aligned}$$

$$\begin{aligned}
&= E_D \left[\frac{1}{M^2} (\Sigma_{i=1}^M (F_i(\mathbf{x}, D) - E_D[F_i(\mathbf{x}, D)]) \right. \\
&\quad \left. (\Sigma_{j=1}^M (F_j(\mathbf{x}, D) - E_D[F_j(\mathbf{x}, D)])) \right] \\
&= E_D \left[\frac{1}{M^2} \Sigma_{i=1}^M (F_i(\mathbf{x}, D) - E_D[F_i(\mathbf{x}, D)])^2 \right] \\
&\quad + E_D \left[\frac{1}{M^2} \Sigma_{i=1}^M \Sigma_{j=1, j \neq i}^M (F_i(\mathbf{x}, D) - E_D[F_i(\mathbf{x}, D)]) \right. \\
&\quad \left. (F_j(\mathbf{x}, D) - E_D[F_j(\mathbf{x}, D)]) \right], \tag{17.39}
\end{aligned}$$

where the first term in the right-hand side of (17.39) is the weighted average of variance among the individual neural networks and the second term is the weighted average covariance among the different neural networks in the ensemble.

Similar to the bias–variance trade-off for a single neural network, there is a bias–variance–covariance trade-off for neural network ensembles. If the individual neural network F_i are highly positively correlated, for example $F_i = F, i = 1, \dots, M$, there is no reduction in the variance of ensemble in this case. If the individual neural network F_i is uncorrelated, the weighted-average covariance among the different neural networks is reduced to zero, and the variance of ensemble can be seen to decay at $\frac{1}{M}$. Both theoretical and experimental results (Clemen and Winkler 1985; Perrone and Cooper 1993) have indicated that when individual neural networks in an ensemble are unbiased, average procedures are most effective in combining them when errors in the individual neural networks are negatively correlated and moderately effective when the errors are uncorrelated. There is little to be gained from average procedures when the errors are positively correlated.

There are a number of methods of learning neural network ensembles. To summarize, there are three approaches: independent learning, sequential learning and simultaneous learning.

17.4.1.2 Independent Learning Methods

It is clear that there is no advantage of combining a set of identical neural networks. In order to create a set of neural networks which are as uncorrelated as possible, a number of methods have been proposed to train a set of neural networks independently by varying initial random weights, the architectures, the learning algorithm used, and the data (Hansen and Salamon 1990; Sarkar 1996; Rogova 1994; Battiti and Colla 1994; Kim et al. 1995).

Experimental results have showed that neural networks obtained from a given neural network architecture for different initial random weights often correctly recognize different subsets of a given test set (Hansen and Salamon 1990; Sarkar 1996). As argued by Hansen and Salamon (1990), because each neural network makes generalization errors on different subsets of the input space, the collective decision produced by the ensemble is less likely to be in error than the decision made by any of the individual neural networks.

Currently, the commonest methods for the creation of ensembles are those which involve altering the training data, such as cross-validation (Krogh and Vedelsby 1995), bootstrapping (Raviv and Intrator 1996) and different input features (Rogova 1994; Battiti and Colla 1994).

Cross-validation is a method of estimating prediction error in its original form (Stone 1974). The procedure of m -fold cross-validation is as follows:

1. Split the data into m roughly equal-sized parts.
2. For the i th part, fit the model to the other $(m - 1)$ parts of the data, and calculate the prediction error of fitted model when predicting the i th part of the data.
3. Do the above for $i = 1, \dots, m$, and combine the m estimates of prediction error.

Cross-validation can be used to create a set of neural networks. Split the data into m roughly equal-sized parts, and train each neural network on the different parts independently. As indicated by Meir (1995), when the data set is small and noisy, such independence will help to reduce the correlation among the m neural networks more drastically than in the case where each neural network is trained on the full data.

When a larger set of independent neural networks is needed, splitting the training data into non-overlapping parts may cause each data part to be too small to train each neural network if no more data are available. In this case, data reuse methods, such as bootstrap (Efron and Tibshirani 1993), can help. Bootstrap was introduced in 1979 as a computer-based method for estimating the standard error of a statistic $s(x)$ (Efron and Tibshirani 1993). Breiman (1996) used the idea of bootstrap in bagging predictors. In bagging predictors, a training set containing N patterns is perturbed by sampling with replacement N times from the training set. The perturbed data set may contain repeats. This procedure can be repeated several times to create a number of different, although overlapping, data sets.

Although an ensemble primarily combines a set of neural networks with same architectures, it has been observed that classifiers based on different classifiers and features are frequently complementary to one another (Rogova 1994; Battiti and Colla 1994). Rogova (1994) proposed to combine several different neural network classifiers. For instance, three different neural network classifiers were used in Rogova's experiment on data of hand-printed digits (Rogova 1994).

17.4.1.3 Sequential Learning Methods

Most independent learning methods emphasize independence among individual neural networks in an ensemble. One of the disadvantages of such a method is the loss of interaction between the individual neural networks during learning. There is no consideration of whether what one individual learns has already been learned by other individuals. The errors of independently trained neural networks may still be positively correlated. It has been found that the combined results are weakened if the errors of individual neural networks are positively correlated (Clemen and Winkler 1985; Perrone and Cooper 1993). In order to decorrelate the individual neural

networks, sequential learning methods train a set of networks in a particular order (Drucker et al. 1993; Opitz and Shavlik 1996; Rosen 1996). Drucker et al. (1993) suggested training the neural networks using the boosting algorithm.

The boosting algorithm was originally proposed by Schapire (1990). Schapire proved that it is theoretically possible to convert a weak learning algorithm that performs only slightly better than random guessing into one that achieves arbitrary accuracy. The proof presented by Schapire (1990) is constructive. The construction uses filtering to modify the distribution of examples in such a way as to force the weak learning algorithm to focus on the harder-to-learn parts of the distribution.

The boosting algorithm trains a set of learning machines sequentially on data that have been filtered by the previously trained learning machines (Schapire 1990). As indicated by Drucker et al. (1994), the original boosting procedure is as follows. The first machine is trained with N_1 patterns randomly chosen from the available training data. After the first machine has been trained, a second training set with N_1 patterns is randomly selected on which the first machine would have 50% error rate. That is, there are 50% of patterns in the training set which the first machine misclassifies. Once the second machine has been trained with the second training set, another set of training patterns are filtered through the first and second machines. Add the patterns on which the two machines disagree into the third training set for the third machine until there are total of N_1 patterns in it. Then the third machine is trained. During testing, each testing pattern is classified using the following voting scheme: if the first two machines agree, take their answer as the output; otherwise, assign the label as classified by the third machine.

Drucker et al. (1993) first used the idea of the boosting algorithm to improve performance of neural networks on four databases of optical character recognition problems. Drucker et al. (1994) compared the performance of the original version of boosting to that of a single neural network for an optical character recognition problem. The results showed that a single network was best for small training set size while for large training set size the original version of boosting was best. The ensemble used by Drucker et al. (1994) only consists of three individual neural networks, where the output of the ensemble is decided by adding of the outputs of the three neural networks rather than voting them.

The boosting algorithm can help to reduce the covariance between the different neural networks in an ensemble. A practical limitation of the original boosting algorithm Drucker et al. (1994) is that with a finite number of training patterns, unless the first network has very poor performance, there may not be enough patterns to generate a second or third training set. This limitation can be overcome by another boosting algorithm called AdaBoost. AdaBoost was developed by Freund and Schapire (1996) and analyzed theoretically by Schapire (1999). AdaBoost belongs to boosting by resampling that allows the training data to be reused.

Different to the boosting algorithm, Rosen (1996) proposed a decorrelation neural network training algorithm in which individual neural networks are trained not only to minimize the error between the target and its output, but also to decorrelate the errors with those from previously trained neural networks.

17.4.1.4 Simultaneous Learning Methods

Most of the independent training methods and sequential training methods follow a two-stage design process: first generating individual neural networks, and then combining them. The possible interactions between the individual neural networks cannot be exploited until the integration stage. There is no feedback from the integration stage to the individual neural network design stage. It is possible that some of the independently designed neural networks do not make much contribution to the integrated system. In order to use the feedback from the integration, simultaneous training methods train a set of neural networks together. The mixtures-of-experts (ME) architectures (Jacobs et al. 1991a,b; Jordan and Jacobs 1994) and negative correlation learning (Liu and Yao 1998a,b, 1999a,b; Chandra and Yao 2006) are two examples of simultaneous training methods.

The ME architecture is composed of multiple neural networks that combine aspects of competitive and associative learning (Jacobs et al. 1991a,b; Jordan and Jacobs 1994). The ME architecture is based on the principle of divide and conquer, in which a complex problem is decomposed into a set of simpler subproblems. It is assumed that the data can be adequately summarized by a collection of functions, each defined over a local region of the input space. The ME architecture adaptively partitions the input space into possibly overlapping regions and allocates different networks to summarize the data located in different regions. The ME architecture consists of two types of neural networks: a gating neural network and a number of expert neural networks. The ME architecture allows for all expert neural networks to look at the input and make their best guess. The gating neural network uses the normalized exponential transformation to weight the outputs of the expert neural networks to provide an overall best guess. All the parameter adjustments in the expert neural networks and gating neural network are performed simultaneously. Although the ME architecture can produce biased individual neural networks whose estimates are negatively correlated (Jacobs 1997), it does not provide a convenient way to balance the bias-variance-covariance trade-off.

17.4.1.5 Negative Correlation Learning

The idea of negative correlation learning is to introduce a correlation penalty term into the error function of each individual network so that the individual network can be trained simultaneously and interactively (Liu and Yao 1999a). Given the training data set $D = \{(\mathbf{x}(1), y(1)), \dots, (\mathbf{x}(N), y(N))\}$, we consider estimating y by forming a neural network ensemble whose output is a simple averaging of outputs F_i of a set of neural networks. All the individual networks in the ensemble are trained on the same training data set D :

$$F(n) = \frac{1}{M} \sum_{i=1}^M F_i(n), \quad (17.40)$$

where $F_i(n)$ is the output of individual network i on the n th training pattern $\mathbf{x}(n)$, $F(n)$ is the output of the neural network ensemble on the n th training pattern, and M is the number of individual networks in the neural network ensemble.

In negative correlation learning, the error function E_i for individual i on the training data set $D = \{(\mathbf{x}(1), y(1)), \dots, (\mathbf{x}(N), y(N))\}$ in negative correlation learning is defined by

$$\begin{aligned} E_i &= \frac{1}{N} \sum_{n=1}^N E_i(n) \\ &= \frac{1}{N} \sum_{n=1}^N \left[\frac{1}{2} (F_i(n) - y(n))^2 + \lambda p_i(n) \right], \end{aligned} \quad (17.41)$$

where N is the number of training patterns, $E_i(n)$ is the value of the error function of network i at presentation of the n th training pattern, and $y(n)$ is the desired output of the n th training pattern. The first term in the right-hand side of (17.41) is the mean-squared error of individual network i . The second term p_i is a correlation penalty function. The purpose of minimizing p_i is to negatively correlate each individual's error with errors for the rest of the ensemble. The parameter λ is used to adjust the strength of the penalty.

The penalty function p_i has the form

$$p_i(n) = -\frac{1}{2} (F_i(n) - F(n))^2. \quad (17.42)$$

The partial derivative of E_i with respect to the output of individual i on the n th training pattern is

$$\begin{aligned} \frac{\partial E_i(n)}{\partial F_i(n)} &= F_i(n) - y(n) - \lambda(F_i(n) - F(n)) \\ &= (1 - \lambda)(F_i(n) - y(n)) + \lambda(F(n) - y(n)), \end{aligned} \quad (17.43)$$

where we have made use of the assumption that the output of ensemble $F(n)$ has constant value with respect to $F_i(n)$. The value of parameter λ lies inside the range $0 \leq \lambda \leq 1$ so that both $(1 - \lambda)$ and λ have non-negative values.

Negative correlation learning has been analyzed in terms of mutual information on a regression task in the different noise conditions (Liu et al. 2001). Unlike independent training which creates larger mutual information among the ensemble, negative correlation learning can produce smaller mutual information among the ensemble. Through minimization of mutual information, very competitive results have been produced by negative correlation learning in comparison with independent training.

The decision boundaries and the correct response sets constructed by negative correlation learning and the independent training have been compared for two pattern classification problems (Liu et al. 2002). The experimental results show that negative correlation learning has a very good classification performance. In fact, the decision boundary formed by negative correlation learning is nearly close to the

optimum decision boundary generated by the Bayes classifier. Negative correlation learning has also been applied in online learning and class imbalance learning (Minku et al. 2010; Tang et al. 2009; Wang and Yao 2009a,b).

17.4.1.6 Evolutionary Neural Networks as Ensembles

Combining individual neural networks in a population into a neural network ensemble has a close relationship to the design of neural network ensembles. The population of neural networks can be regarded as an ensemble. The evolutionary process can be regarded as a natural and automatic way to design neural network ensembles.

Evolutionary ensembles with negative correlation learning (EENCL) were developed for automatically designing neural network ensembles (Liu et al. 2000). EENCL are studied to address the following issues: exploitation of the interaction between individual neural network design and combination and automatic determination of the number of individual neural networks. In EENCL, an evolutionary algorithm based on evolutionary programming (Fogel 1995) is used to search for a population of diverse individual neural networks that together solve a problem. To maintain a diverse population, fitness sharing (Yao et al. 1996) and negative correlation learning are used to encourage the formation of different species. In the implementation of EENCL, each neural network in the ensemble is a feedforward neural network with logistic transfer functions. The major steps of EENCL are as follows:

1. Generate an initial population of M neural networks, and set $k = 1$. The number of hidden nodes for each neural network, n_h , is specified by the user. The random initial weights are uniformly distributed inside a small range.
2. Train each neural network in the initial population on the training set for a certain number of epochs using negative correlation learning. The number of epochs, n_e , is specified by the user.
3. Randomly choose a group of n_b neural networks as parents to create n_b offspring neural networks by Gaussian mutation.
4. Add the n_b offspring neural networks to the population and train the offspring neural networks using negative correlation learning while the rest of the neural networks' weights are frozen.
5. Calculate the fitness of $M + n_b$ neural networks in the population and prune the population to the M fittest neural networks.
6. Go to the next step if the maximum number of generations has been reached. Otherwise, $k = k + 1$ and go to Step 3.
7. Form species using the k -means algorithm.
8. Combine species to form the ensembles.

There are two levels of adaptation in EENCL: negative correlation learning at the individual level and evolutionary learning based on evolutionary programming (Fogel 1995) at the population level. Negative correlation learning and fitness

sharing have been used to encourage the formation of species in the population. EENCL were tested on the Australian credit card assessment problem and the diabetes problem (Liu et al. 2000). Very competitive results have been produced by EENCL in comparison with other algorithms (Liu et al. 2000). Three combination methods have been investigated in EENCL, including simple averaging, majority voting and winner-takes-all.

Besides the k -means algorithm, regularized learning and constructive learning were also introduced in negative correlation learning for automatically determining the size of the ensembles (Chen and Yao 2009; Islam et al. 2003; Chen and Yao 2010).

17.5 Promising Areas for Future Application

Six recent trends and directions in machine learning are summarized by Dietterich (1997) and Langley (1996).

The first trend is in experimental studies of learning algorithms. Experimental studies of learning algorithms have shifted from the early study of idealized, hand-crafted examples to realistic learning tasks that involve hundreds and thousands of cases. Besides robustness and the generality test of learning algorithms on a number of different data sets, comparisons between different learning algorithms on the same task domains need to be done. It has been realized that some explicit methods for evaluating different learning algorithms should be established, and the conditions in which a learning algorithm will perform well should be identified in order to make progress in machine learning.

The second trend is in theoretical analyses of learning processes. The main goal of theoretical analysis is to find the inductive principle with the best generalization, and then to develop learning algorithms with such inductive principle. Early studies on the convergence of learning algorithms were important but showed little insight into real learning problems. A major advance was due to the introduction of the probably approximately correct model (Vapnik 1995). For the first time, this model provided theoretical accuracy guarantees that were based on a finite number of training samples. The resulting probably approximately correct model also served the rigorous framework that addressed the concerns from real-world problems.

The third trend is in applications of machine learning. Most recent successful applications are in classification or prediction tasks. Machine learning has also been applied in the areas of configuration and layout, planning and scheduling, and execution and control. In order for machine learning to play a big role in solving problems of interest to industry and commerce, many more applications need to be undertaken.

The fourth trend is on new learning algorithms. Many new learning algorithms have been studied in the past decade. For example, a support vector machine can construct a hyperplane as the decision surface in such a way that the margin of separation between positive and negative examples is maximized. A boosting algorithm

trains a set of classifiers on data sets with entirely different distributions, and combines them in an elementary way to achieve near-optimal performance. The boosting algorithm was originally proposed by [Schapire \(1990\)](#). Schapire proved that it is theoretically possible to convert a weak learning algorithm that performs only slightly better than random guessing into one that achieves arbitrary accuracy. Boosting is a general method that can be used to improve the performance of any learning algorithm. Another ensemble learning called bagging combines models built on resamplings of data to yield superior models ([Breiman 1996](#)).

The fifth trend is in unified frameworks for machine learning. Machine learning has been widely studied from a variety of backgrounds. The similarities between the various approaches have often been overlooked while the differences between them were emphasized. It is important to draw distinctions among different learning algorithms. However, an ultimate goal of machine learning is to study a unified framework that can explain different learning processes in terms of common underlying mechanisms. One route to this goal is the study of hybrid learning systems that incorporate aspects of different learning algorithms.

The sixth trend is in integrated cognitive architectures. It is related to the development of integrated architectures for cognition. A common implementation in early work was to design a separate system for each new task. These systems had little of the nature of intelligent behavior, and posed limitations on work in other domains. Now research has moved to the design of integrated architectures that make strong assumptions about the control structures that can support intelligence. It is clear that learning will continue to play an important role in the development of such cognitive architectures when it is necessary to acquire knowledge from the environment for long-term adaptive behavior.

These new areas will confront researchers with many more challenge problems, and novel directions will surely emerge when the limitations of existing learning algorithms are revealed.

17.6 Tricks of the Trade

Newcomers to the field of machine learning, applying a learning algorithm to a given problem, are often not very clear about where to start to come up with a successful implementation. The following step-by-step procedures provide a guideline for implementing machine learning algorithms ([Langley and Simon 1995](#)).

17.6.1 *Formulating the Problem*

The first step is to formulate a given problem in terms of what can be dealt with by a particular learning algorithm. Often, some real-world problems can be transformed into simple classification tasks. For an example, the breast cancer diagnosis problem

can be formulated as a classification task that classifies a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Strategies such as divide-and-conquer can be used to decompose a complex task into a set of subproblems more amenable to the chosen learning algorithm. The strengths and limitations discussed in Sect. 17.2 provide a guideline for selecting an appropriate learning algorithm. Additionally, based on the feedback available in the problem, supervised learning can be chosen when specific output is supplied; unsupervised should be adopted when there is no specific output; reinforcement learning can be applied when the environment can be either fully observable or partially observable.

17.6.2 Choosing the Representation

The second step is to choose an appropriate representation for both the data and knowledge to be learned. The representation is referred to the attributes or features describing examples rather than the representational formalism, such as decision trees or neural networks. In some real-world problems, there might be thousands of potential features describing each input. Most learning algorithms do not scale well when there are many features. Meanwhile, examples with many irrelevant and noisy input features give little information from a statistical point of view. It is essential to choose useful and important features to feed to the learning algorithms. There are three main approaches for feature selection (Dietterich 1997). The first approach is to select a subset of the features based on some initial analysis. The second approach is to test different subsets of the features on the chosen learning algorithm and select the subsets that generate the best performance. The third approach is to automate the selection and weighting of features in the learning algorithm.

17.6.3 Collecting the Data

The third step is to collect data needed for the learning algorithm. In some applications, this process may be straightforward, but in others it can be very difficult. Generally speaking, the quantity of the data is decided by the chosen learning algorithm. Data preprocessing is often necessary in the learning process.

17.6.4 Conducting the Learning Process

Once the data are ready, the learning process can be started to find the best learning model within a set of candidate model structures according to a certain criterion. A standard tool in statistics known as cross-validation provides a good guiding criterion. First the collected data are randomly partitioned into a training set and a

test set. The training set is further divided into two disjoint subsets called estimation subset and validation subset in which the estimation subset is used to induce the learning model, and the validation subset is used to validate the model. It is possible that the learned model may end up overfitting on the validation subset. Therefore, the generalization performance of the learned model is measured on the test set which is different from the validation subset. Some learning algorithms, such as inductive logic programming, rely on the background knowledge available. How to obtain such helpful background knowledge is an important issue that will affect the outcome of those learning algorithms.

17.6.5 Analyzing and Evaluating the Learned Knowledge

Empirical comparisons have often been used to evaluate the predictive performance of the various learning methods. The experiments can be conducted on simulated data sets or a real-life data set, or both. The known best predictions on some simulated data sets make it possible to compare the learned knowledge with the known knowledge. Real-life data are helpful for evaluation of the robustness and generality of different learning algorithms. Cross-validation is a method of estimating prediction error in its original form ([Stone 1974](#)). The procedure of m -fold cross-validation is as follows:

1. Split the data into m roughly equal-sized parts.
2. For the i th part, fit the model to the other $(m - 1)$ parts of the data, and calculate the prediction error of fitted model when predicting the i th part of the data.
3. Do the above for $i = 1, \dots, m$, and combine the m estimates of prediction error.

In addition to empirical comparisons, statistical learning theory can be used to analyze the generalization ability of learning algorithms. [Vapnik \(1995\)](#) argued that the Vapnik–Chervonenkis dimension of the set of functions (rather than number of parameters) is responsible for the generalization ability of learning machines. This opens remarkable opportunities to overcome the “curse of dimensionality”: to generalize well on the basis of a set of functions containing a huge number of parameters but possessing a small Vapnik–Chervonenkis dimension.

17.7 Conclusions

This chapter has been primarily concerned with the core learning algorithms including decision tree, inductive logic programming, Bayesian learning, neural networks, evolutionary learning and reinforcement learning. Inevitably, there are some important learning algorithms that have not been covered. One important learning algorithm dealing with imprecise and uncertain knowledge and data is fuzzy logic. Imprecision is treated based on probability in statistical learning. In contrast, fuzzy

logic is concerned with the use of fuzzy values that capture the meaning of words, human reasoning and decision making. At the heart of fuzzy logic lies the concept of a linguistic variable. The values of the linguistic variable are words rather than numbers.

Generalization is one of key issues in machine learning. In neural network learning, generalization was studied from the bias–variance trade-off point of view (Geman et al. 1992). There is usually a trade-off between bias and variance in the case of a training set with finite size: attempts to decrease bias by introducing more parameters often tend to increase variance; attempts to reduce variance by reducing parameters often tend to increase bias. Besides the generalization issue, how to scale up learning algorithms is another important issue. Dietterich (1997) reviewed learning with a large training set and learning with many features. Even though some learning techniques can solve very large problems with millions of training examples in a reasonable amount of computer time, it is unclear whether they can successfully be applied to those problems with billions of training examples.

This chapter has also been concerned with exploring the possible benefits arising from combining learning with evolution with neural networks. Different learning algorithms have their own strengths and weaknesses. Among all learning algorithms, there is no clear winner in terms of the best learning algorithm. The best one is always problem dependent. This is certainly true according to the no-free-lunch theorem (Wolpert and Macready 1997). In general, hybrid algorithms tend to perform better than others for a large number of problems.

Sources of Additional Information

The literature on machine learning is rather large, and has been growing rapidly.

- Mitchell's *Machine Learning* (Mitchell 1997) and Russell and Norvig's *Artificial Intelligence: A Modern Approach* (Russell and Norvig 2002) give good overviews of different types of learning algorithms.
- Machine Learning, volumes I–III, provide the early history of machine learning development (Michalski et al. 1983, 1986; Kodratoff and Michalski 1990).
- Some important papers in machine learning are collected in *Readings in Machine Learning* (Shavlik and Dietterich 1990).
- Current research in machine learning spreads out over a number of journals. Major machine learning journals include *Machine Learning*, the *Journal of Machine Learning Research*, *IEEE Transactions on Neural Networks*, *IEEE Transactions on Evolutionary Computation*, and mainstream artificial intelligence journals.
- Machine learning is also covered by a number of conferences, such as the International Conference on Machine Learning, the International Joint Conference on Neural Networks, Congress on Evolutionary Computation, the IEEE International Conference on Fuzzy Systems, and the Conference on Neural Information Processing Systems.

- Mlnet Online Information Service (<http://www.mlnet.org/>) funded by the European Commission is dedicated to the field of machine learning, knowledge discovery, case-based reasoning, knowledge acquisition and data mining.
- Machine learning topics can also be found at the website of the American Association for Artificial Intelligence: www.aaai.org/Pathfinder/html/machine.html.

References

- Battiti R, Colla AM (1994) Democracy in neural nets: voting schemes for classification. *Neural Netw* 7:691–707
- Breiman L (1996) Bagging predictors. *Mach Learn* 24:123–140
- Breiman L, Friedman J, Olshen RA, Stone PJ (1984) Classification and regression trees. Wadsworth, Belmont
- Chandra A, Yao X (2006) Ensemble learning using multi-objective evolutionary algorithms. *J Math Model Algorithms* 5:417–445
- Chen H, Yao X (2009) Regularized negative correlation learning for neural network ensembles. *IEEE Trans Neural Netw* 20:1962–1979
- Chen H, Yao X (2010) Multiobjective neural network ensembles based on regularized negative correlation Learning. *IEEE Trans Knowl Data Eng* 22:1738–1751
- Cheng J, Greiner R, Kelly J, Bell DA, Liu W (2002) Learning Bayesian networks from data: an information-theory based approach. *Artif Intell* 137:43–90
- Clemen RT, Winkler RL (1985) Limits for the precision and value of information from dependent sources. *Oper Res* 33:427–442
- Dietterich TG (1997) Machine-learning research: four current directions. *AI Mag* 18:97–136
- Domingos P, Pazzani M (1996) Beyond independence: conditions for the optimality of the simple Bayesian classifier. In: Saitta L (ed) Proceedings of the 13th international conference on machine learning, Bari. Morgan Kaufmann, San Mateo, pp 105–112
- Drucker H, Schapire R, Simard P (1993) Improving performance in neural networks using a boosting algorithm. In: Hanson SJ et al (eds) Advances in neural information processing systems 5. Morgan Kaufmann, San Mateo, pp 42–49
- Drucker H, Cortes C, Jackel LD, LeCun Y, Vapnik V (1994) Boosting and other ensemble methods. *Neural Comput* 6:1289–1301
- Efron B, Tibshirani RJ (1993) An introduction to the bootstrap. Chapman and Hall, London
- Elkan C (1997) Boosting and naive Bayesian learning. Technical report, Department of Computer Science and Engineering, University of California
- Feigenbaum EA (1961) The simulation of verbal learning behavior. In: Proceedings of the western joint computer conference, Los Angeles, pp 121–131
- Fogel DB (1995) Evolutionary computation: towards a new philosophy of machine intelligence. IEEE, New York

- Fogel LJ, Owens AJ, Walsh MJ (1966) Artificial intelligence through simulated evolution. Wiley, New York
- Freund Y, Schapire RE (1996) Experiments with a new boosting algorithm. In: Proceedings of the 13th international conference on machine learning, Bari. Morgan Kaufmann, San Mateo, pp 148–156
- Geman S, Bienenstock E, Doursat R (1992) Neural networks and the bias/variance dilemma. *Neural Comput* 4:1–58
- Hansen LK, Salamon P (1990) Neural network ensembles. *IEEE Trans Pattern Anal Mach Intell* 12:993–1001
- Hebb DO (1949) The organization of behavior: a neurophysiological theory. Wiley, New York
- Heckerman D (1998) A tutorial on learning with Bayesian networks. In: Jordan MI (ed) Learning in graphical models. Kluwer, Dordrecht
- Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. *Proc Nat Acad Sci USA* 79:2554–2558
- Hopfield JJ, Tank DW (1985) Neural computation of decisions in optimization problems. *Biol Cybern* 52:141–152
- Hunt EB, Marin J, Stone PT (1966) Experiments in induction. Academic, New York
- Islam MM, Yao X, Murase K (2003) A constructive algorithm for training cooperative neural network ensembles. *IEEE Trans Neural Netw* 14:820–834
- Jacobs RA (1997) Bias/variance analyses of mixture-of-experts architectures. *Neural Comput* 9:369–383
- Jacobs RA, Jordan MI, Barto AG (1991a) Task decomposition through competition in a modular connectionist architecture: the what and where vision task. *Cogn Sci* 15:219–250
- Jacobs RA, Jordan MI, Nowlan SJ, Hinton GE (1991b) Adaptive mixtures of local experts. *Neural Comput* 3:79–87
- Jordan MI, Jacobs RA (1994) Hierarchical mixtures-of-experts and the EM algorithm. *Neural Comput* 6:181–214
- Kaelbling LP, Littman ML, Moore AW (1996) Reinforcement learning: a survey. *J Artif Intell Res* 4:237–285
- Kim J, Ahn J, Cho S (1995) Ensemble competitive learning neural networks with reduced input dimensions. *Int J Neural Syst* 6:133–142
- Kodratoff Y, Michalski RS (eds) (1990) Machine learning—an artificial intelligence approach 3. Morgan Kaufmann, San Mateo
- Krogh A, Vedelsby J (1995) Neural network ensembles, cross validation, and active learning. In: Tesauro G et al (eds) Advances in neural information processing systems 7. MIT, Cambridge, pp 231–238
- Langley P (1996) Elements of machine learning. Morgan Kaufmann, San Francisco
- Langley P, Simon H (1995) Applications of machine learning and rule induction. *Commun ACM* 38:54–64
- Lavrač N, Džeroski S (1994) Inductive logic programming: techniques and applications. Ellis Horwood, Chichester
- Liu Y, Yao X (1998a) Negatively correlated neural networks can produce best ensembles. *Aust J Intell Inf Process Syst* 4:176–185

- Liu Y, Yao X (1998b) A cooperative ensemble learning system. In: Proceedings of the IJCNN 1998, Anchorage. IEEE, Piscataway, pp 2202–2207
- Liu Y, Yao X (1999a) Simultaneous training of negatively correlated neural networks in an ensemble. *IEEE Trans Syst Man Cybern B* 29:716–725
- Liu Y, Yao X (1999b) Ensemble learning via negative correlation. *Neural Netw* 12:1399–1404
- Liu Y, Yao X, Higuchi T (2000) Evolutionary ensembles with negative correlation learning. *IEEE Trans Evol Comput* 4:380–387
- Liu Y, Yao X, Higuchi T (2001) Ensemble learning by minimizing mutual information. In: Proceedings of the 2nd international conference on software engineer, artificial intelligence, networking and parallel/distributed computing, Nagoya. International association for computer and information science, pp 457–462
- Liu Y, Yao X, Zhao Q, Higuchi T (2002) An experimental comparison of neural network ensemble learning methods on decision boundaries. In: Proceedings of the IJCNN 2002, Honolulu. IEEE, Piscataway, pp 221–226
- McCulloch WS, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. *Bull Math Biophys* 5:115–137
- Meir R (1995) Bias, variance, and the combination of least squares estimators. In: Tesauro G, Touretzky DS, Leen TK (eds) *Advances in neural information processing systems 7*. MIT, Cambridge, pp 295–302
- Michalski RS, Carbonell JG, Mitchell TM (eds) (1983) *Machine learning—an artificial intelligence approach 1*. Morgan Kaufmann, San Mateo
- Michalski RS, Carbonell JG, Mitchell TM (eds) (1986) *Machine learning—an artificial intelligence approach 2*. Morgan Kaufmann, San Mateo
- Michie D, Spiegelhalter DJ, Taylor CC (1994) *Machine learning, neural and statistical classification*. Ellis Horwood, London
- Minku LL, White A, Yao X (2010) The impact of diversity on on-line ensemble learning in the presence of concept drift. *IEEE Trans Knowl Data Eng* 22:730–742
- Minsky ML, Papert S (1969) *Perceptrons: an introduction to computational geometry*. MIT, Cambridge
- Mitchell TM (1997) *Machine learning*. McGraw-Hill, New York
- Muggleton SH (1995) Inverse entailment and progol. *New Gener Comput* 13:245–286
- Muggleton SH, Buntine W (1988) Machine invention of first-order predicates by inverting resolution. In: Proceedings of the 5th international conference on machine learning, Ann Arbor. Morgan Kaufmann, San Mateo, pp 339–352
- Opitz DW, Shavlik JW (1996) Actively searching for an effective neural network ensemble. *Connect Sci* 8:337–353
- Perrone MP, Cooper LN (1993) When networks disagree: ensemble methods for hybrid neural networks. In: Mammone RJ (ed) *Neural networks for speech and image processing*. Chapman and Hall, London
- Quinlan JR (1986) Introduction to decision tree. *Mach Learn* 1:81–106
- Quinlan JR (1990) Learning logical definitions from relations. *Mach Learn* 5:239–266

- Quinlan JR (1993) C4.5: programs for machine learning. Morgan Kaufmann, San Mateo
- Raviv Y, Intrator N (1996) Bootstrapping with noise: an effective regularization technique. *Connect Sci* 8:355–372
- Rogova G (1994) Combining the results of several neural networks classifiers. *Neural Netw* 7:777–781
- Rosen BE (1996) Ensemble learning using decorrelated neural networks. *Connect Sci* 8:373–383
- Rosenblatt F (1962) Principles of neurodynamics: perceptrons and the theory of brain mechanisms. Spartan, Chicago
- Rumelhart DE, McClelland JL (ed) (1986) Parallel distributed processing: explorations in the microstructures of cognition. MIT, Cambridge
- Rumelhart DE, Hinton GE, Williams RJ (1986) Learning internal representations by error propagation. In: Rumelhart DE, McClelland JL (eds) *Parallel distributed processing: explorations in the microstructures of cognition I*. MIT, Cambridge, pp 318–362
- Russell S, Norvig P (2002) Artificial intelligence: a modern approach. Prentice-Hall, Englewood Cliffs
- Samuel AL (1959) Some studies in machine learning using the game of checkers. *IBM J Res Dev* 3:210–229
- Sarkar D (1996) Randomness in generalization ability: a source to improve it. *IEEE Trans Neural Netw* 7:676–685
- Schapire RE (1990) The strength of weak learnability. *Mach Learn* 5:197–227
- Schapire RE (1999) Theoretical views of boosting and applications. In: *Proceedings of the 10th international conference on algorithmic learning theory*, Tokyo. Springer, Berlin, pp 13–25
- Schwefel HP (1981) Numerical optimization of computer models. Wiley, Chichester
- Schwefel HP (1995) Evolution and optimum seeking. Wiley, New York
- Shavlik J, Dietterich T (eds) (1990) Readings in machine learning. Morgan Kaufmann, San Mateo
- Stone M (1974) Cross-validatory choice and assessment of statistical predictions. *J R Stat Soc* 36:111–147
- Sutton RS, Barto AG (1998) Reinforcement learning: an introduction. MIT, Cambridge
- Tang K, Lin M, Minku FL, Yao X (2009) Selective negative correlation learning approach to incremental learning. *Neurocomputing* 72:2796–2805
- Turing A (1950) Computing machinery and intelligence. *Mind* 59:433–460
- Vapnik VN (1995) The nature of statistical learning theory. Springer, New York
- Wang S, Yao X (2009a) Theoretical study of the relationship between diversity and single-class measures for class imbalance learning. In: *Proceedings of the IEEE international conference on data mining workshops*, Miami. IEEE Computer Society, Washington, DC, pp 76–81
- Wang S, Yao X (2009b) Diversity exploration and negative correlation learning on imbalanced data sets. In: *Proceedings of the IJCNN 2009*, Atlanta, pp 3259–3266

- Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. *IEEE Trans Evol Comput* 1:67–82
- Yao X (1991) Evolution of connectionist networks. In: Dartnall T (ed) Preprints of the international symposium on AI, reasoning and creativity, Griffith University, Queensland, pp 49–52
- Yao X (1993a) A review of evolutionary artificial neural networks. *Int J Intell Syst* 8:539–567
- Yao X (1993b) An empirical study of genetic operators in genetic algorithms. *Microprocess Microprogr* 38:707–714
- Yao X (1994) The evolution of connectionist networks. In: Dartnall T (ed) Artificial intelligence and creativity. Kluwer, Dordrecht, pp 233–243
- Yao X (1995) Evolutionary artificial neural networks. In: Kent A, Williams JG (eds) Encyclopedia of computer science and technology 33. Dekker, New York, pp 137–170
- Yao X (1999) Evolving artificial neural networks. *Proc IEEE* 87:1423–1447
- Yao X, Liu Y (1997) A new evolutionary system for evolving artificial neural networks. *IEEE Trans Neural Netw* 8:694–713
- Yao X, Liu Y (1998) Making use of population information in evolutionary artificial neural networks. *IEEE Trans Syst Man Cybern B* 28:417–425
- Yao X, Liu Y, Darwen P (1996) How to make best use of evolutionary learning. In: Stocker R, Jelinek H, Durnota B (eds) Complex systems: from local interactions to global phenomena. IOS, Amsterdam, pp 229–242

Chapter 18

Fuzzy Reasoning

Costas P. Pappis and Constantinos I. Siettos

18.1 Introduction

The derivation of mathematical models that can efficiently describe real-world problems is generally an overwhelming or even impossible task, due to the complexity and inherent ambiguity of characteristics that these problems can possess. As L. A. [Zadeh \(1973\)](#), the founder of the theory of fuzzy sets, puts it

as the complexity of a system increases, our ability to make precise and yet significant statements about its behavior diminishes until a threshold is reached beyond which precision and significance (or relevance) become almost mutually exclusive characteristics.

Fuzzy reasoning is based on the theory of fuzzy sets and it encompasses artificial intelligence, information processing and theories from logic to pure and applied mathematics, like graph theory, topology and optimization. The theory of fuzzy sets was introduced in 1965. In his introductory paper, Zadeh, while stating his intention “to explore in a preliminary way some of the basic properties and implications” of fuzzy sets, noted that

the notion of a fuzzy set provides a convenient point of departure for the construction of a conceptual framework which parallels in many respects the framework used in the case of ordinary sets, but is more general than the latter and, potentially, may prove to have a much wider scope of applicability, particularly in the fields of pattern classification and information processing.

Indeed, in subsequent years, the theory of fuzzy sets was more decisively established as a new approach to complex systems theory and decision processes. The

C.P. Pappis
University of Piraeus, Piraeus, Greece
e-mail: pappis@unipi.gr

C.I. Siettos (✉)
National Technical University, Athens, Greece
e-mail: ksiet@mail.ntua.gr

Table 18.1 A chronology of critical points in the development of fuzzy reasoning

First paper on fuzzy systems	Zadeh 1965
Linguistic approach	Zadeh 1973
Fuzzy logic controller	Assilian and Mamdani 1974
Heat exchanger control based on fuzzy logic	Østergaard 1977
First industrial application of fuzzy logic: cement kiln control	Homblad and Østergaard 1982
Self-organizing fuzzy controller	Procyk and Mamdani 1979
Fuzzy pattern recognition	Bezdek 1981
Fuzzy controllers on Tokyo subway shuttles	Hitachi 1984
Fuzzy chip	Togai and Watanabe 1986
Takagi–Sugeno fuzzy modeling	Takagi and Sugeno 1985
Hybrid neural-fuzzy systems	Kosko 1992

applications of fuzzy logic have dramatically increased since 1990, ranging from production, finance, marketing and other decision-making problems to biomedical applications, micro-controller-based systems in home appliances and large-scale process control systems (e.g. [Østergaard 1977, 1990; Sugeno and Yasukawa 1993; Karr and Gentry 1993; Lee 1990; Kunsch and Fortemps 2002; Ruan et al. 2003; Polat et al. 2006; Cheng et al. 2008](#)). For systems involving nonlinearities and lack of a reliable analytical model, fuzzy logic control has emerged as one of the most promising approaches. Definitely, fuzzy inference is a step towards the simulation of human thinking.

The main advantage of fuzzy logic techniques, i.e. techniques based on the theory of fuzzy sets, over more conventional approaches in solving complex, nonlinear and/or ill-defined problems lies in their capability of incorporating a priori qualitative knowledge and expertise about system behavior and dynamics. This renders fuzzy logic systems almost indispensable for obtaining a more transparent and tactile qualitative insight into systems whose representation with exact mathematical models is poor and inadequate. Besides, fuzzy schemes can be used either as enabling to other approaches or as self-reliant methodologies providing thereby a plethora of alternative structures and schemes.

In fact, fuzzy control theory generates nonlinear functions according to a representation theorem by [Wang \(1992\)](#), who stated that any continuous nonlinear function can be approximated as exactly as needed with a finite set of fuzzy variables, values and rules. Therefore, by applying appropriate design procedures, it is always possible to design a fuzzy controller that is suitable for the nonlinear system under control. Table 18.1 depicts some benchmarks in the history of fuzzy logic, particularly in the domain of fuzzy control.

This chapter presents an overview of the basic notions of the theory of fuzzy sets and fuzzy logic. In the next section, an introduction to the theory of fuzzy sets is presented, covering topics of the most commonly used types of membership functions, logical and transformation operators, fuzzy relations, implication and inference rules, and fuzzy similarity measures. Section 18.3 introduces the basic structure of a fuzzy inference system and its elements are described. Section 18.4 presents the topic of fuzzy control system and an example is demonstrated. In particular, a fuzzy controller is proposed for the control of a plug flow tubular reactor, which is a typical nonlinear distributed parameter. The proposed fuzzy controller is compared with a conventional PI controller. In the same section an introduction to the field of fuzzy adaptive control systems is given and the self-organizing scheme is presented. In Sect. 18.5 reviews are given on the topics of model identification and stability of fuzzy systems, respectively. Conclusions and perspectives on fuzzy reasoning are given in Sect. 18.6.

18.2 Basic Definitions of Fuzzy Set Theory

18.2.1 Fuzzy Sets and the Notion of Membership

A classical set A is defined as a collection of elements or objects. Any element or object x either belongs or does not belong to A . The membership $\mu_A(x)$ of x in A is a mapping

$$\mu_A : X \rightarrow \{0, 1\}$$

that is, it may take the value 1 or 0, which represent the truth value of x in A . It follows that, if \bar{A} is the complement set of A and \cap represents intersection of sets, then

$$A \cap \bar{A} = \emptyset.$$

Fuzzy logic is a logic based on fuzzy sets, i.e. sets of elements or objects characterized by truth values in the $[0, 1]$ interval rather than crisp 0 and 1, as in conventional set theory. The function that assigns a number in $[0, 1]$ to each element of the universe of discourse of a fuzzy set is called a membership function.

18.2.2 Membership Functions

Let X denote the universe of discourse of a fuzzy set A , which is completely characterized by its membership function μ_A :

$$\mu_A : X \rightarrow [0, 1]$$

and is defined as a set of pairs:

$$A = \{(x, \mu_A(x))\}.$$

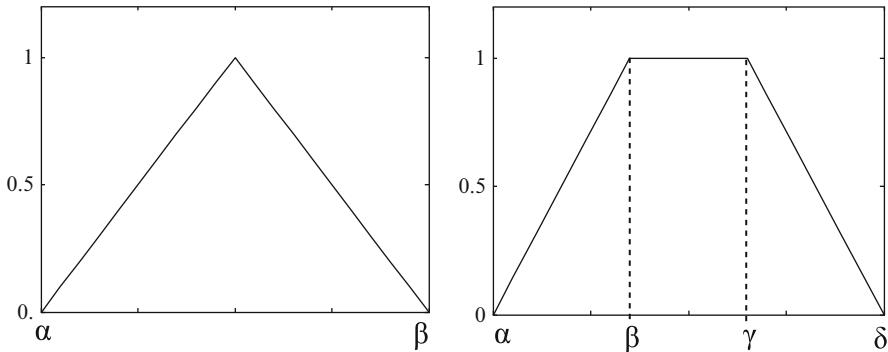


Fig. 18.1 Triangular (left) and trapezoid (right) membership functions

The most commonly used membership functions are the following (Dubois and Prade 1980; Zimmermann 1996):

- Triangular membership function
- Trapezoid membership function
- Linear membership function
- Sigmoidal membership function
- Π -type membership function
- Gaussian membership function.

The *triangular* membership function (Fig. 18.1, left) is defined as

$$\text{Tri}(x; \alpha, \beta, \gamma) = \begin{cases} 0 & x < \alpha \\ \frac{x-\alpha}{\beta-\alpha} & \alpha \leq x < \beta \\ \frac{x-\gamma}{\gamma-\beta} & \beta \leq x < \gamma \\ 0 & x \geq \gamma. \end{cases}$$

The *trapezoid* membership function (Fig. 18.1, right) is defined as

$$\text{Tra}(x; \alpha, \beta, \gamma, \delta) = \begin{bmatrix} 0 & x < \alpha \\ \frac{x-\alpha}{\beta-\alpha} & \alpha \leq x < \beta \\ 1 & \beta \leq x < \gamma \\ -\frac{x-\delta}{\delta-\gamma} & \gamma \leq x < \delta \\ 0 & x \geq \delta \end{bmatrix}.$$

The *monotonically increasing linear* membership function (Fig. 18.2, left) is given by

$$L(x; \alpha, \beta) = \begin{cases} 0 & x < \alpha \\ \frac{x-\alpha}{\beta-\alpha} & \alpha \leq x \leq \beta \\ 1 & x > \beta. \end{cases}$$

The *monotonically decreasing linear* membership function (Fig. 18.2, right) is given by

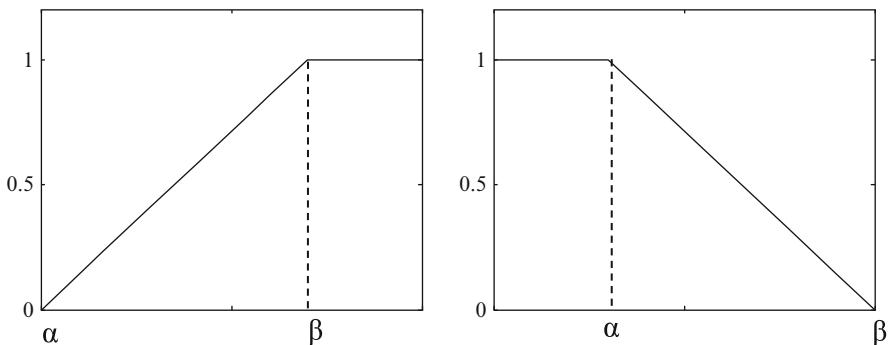


Fig. 18.2 Monotonically increasing linear (*left*) and monotonically decreasing linear (*right*) membership functions

$$L(x; \alpha, \beta) = \begin{cases} 1 & x < \alpha \\ -\frac{x-\beta}{\beta-\alpha} & \alpha \leq x \leq \beta \\ 0 & x > \beta. \end{cases}$$

The *monotonically increasing sigmoidal* membership function (Fig. 18.3, left) is given by

$$S(x; \alpha, \beta, \gamma) = \begin{cases} 0 & x < \alpha \\ 2 \left(\frac{x-\alpha}{\gamma-\alpha} \right)^2 & \alpha \leq x \leq \beta \\ 1 - 2 \left(\frac{x-\gamma}{\gamma-\alpha} \right)^2 & \beta \leq x \leq \gamma \\ 1 & x > \gamma. \end{cases}$$

The *monotonically decreasing sigmoidal* membership function (Fig. 18.3, right) reads

$$S(x; \alpha, \beta, \gamma) = \begin{cases} 1 & x < \alpha \\ 1 - 2 \left(\frac{x-\alpha}{\gamma-\alpha} \right)^2 & \alpha \leq x \leq \beta \\ 2 \left(\frac{x-\gamma}{\gamma-\alpha} \right)^2 & \beta \leq x \leq \gamma \\ 0 & x > \gamma. \end{cases}$$

The Π -membership function (Fig. 18.4, left) is defined as

$$\Pi(x; \beta, \gamma) = \begin{cases} S \left(x; \gamma - \beta, \frac{\gamma - \beta}{2}, \gamma \right) & x \leq \gamma \\ 1 - S \left(x; \gamma, \frac{\gamma + \beta}{2}, \gamma + \beta \right) & x > \gamma. \end{cases}$$

The *Gaussian* membership function (Fig. 18.4, right) is given by

$$G(x; \gamma, \sigma) = \exp \left(-(\gamma - x)^2 / 2\sigma^2 \right),$$

where σ is the standard deviation.

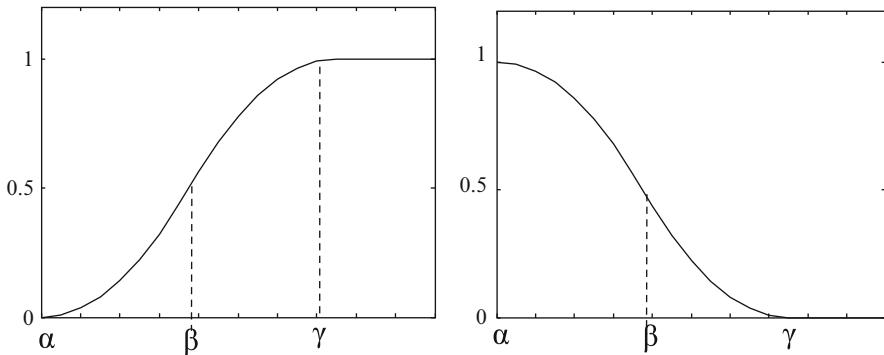


Fig. 18.3 Monotonically increasing sigmoidal (left) and monotonically decreasing sigmoidal (right) membership functions

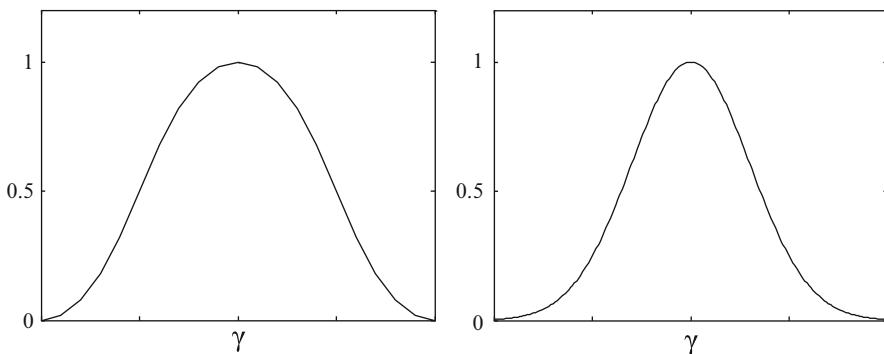
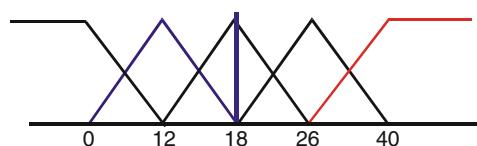


Fig. 18.4 Π (left) and Gaussian (right) membership functions

18.2.2.1 Examples of Fuzzy Sets

Maintaining a comfortable room temperature is of great importance for work productivity. Fuzzy logic climate control is one of the many commercial applications of fuzzy set theory. For example, the room temperature for low-level activities could be described by the following five fuzzy sets: a temperature around 18°C is comfortable, around 26°C is warm (though not during summer!), above 40°C is definitely too warm, while around 12°C can be characterized as cold, and below that too cold.



Fast cars can be described by their horsepower (HP) using the following membership function:

$$\mu(x) = \begin{cases} 0, & 0 \leq x \leq 75 \\ \frac{x-120}{25}, & 75 \leq x \leq 120 \\ 1, & 120 \leq x \leq 150 \\ 0, & x \geq 150 \end{cases}.$$

Notice that in the above characterization, for a horsepower above 150 HP *fast* cars have a zero membership. A nonzero membership could have been assigned in another the fuzzy set (e.g. the fuzzy set of *very fast* cars).

18.2.3 Fuzzy Set Operations

Knowledge and understanding of the operations of the theory of fuzzy sets is important for the design of fuzzy systems. The fuzzy set operations are defined with respect to the sets' membership functions.

Two fuzzy sets A and B on the universe of discourse X are *equal* if their membership functions are equal for each $x \in X$:

$$\forall x \in X : \mu_A(x) = \mu_B(x).$$

A fuzzy set A is a *subset* of B ($A \subseteq B$) if

$$\forall x \in X : \mu_A(x) \leq \mu_B(x).$$

For the operation of *intersection* \cap of two fuzzy sets A and B , there is a plethora of definitions in the references. The choice is application dependent:

$$\forall x \in X : \mu_{A \cap B} = \begin{cases} \min(\mu_A(x), \mu_B(x)) \\ \frac{\mu_A(x) + \mu_B(y)}{2} \\ \mu_A(x)\mu_B(y) \\ \dots \dots \end{cases}.$$

The *union* \cup of two fuzzy sets A and B is also defined in several ways:

$$\forall x \in X : \mu_{A \cup B} = \begin{cases} \max(\mu_A(x), \mu_B(y)) \\ \frac{2\min(\mu_A(x), \mu_B(y)) + 4\max(\mu_A(x), \mu_B(y))}{6} \\ \mu_A(x) + \mu_B(y) - \mu_A(x)\mu_B(y) \\ \dots \dots \end{cases}$$

The *complement* A' of a fuzzy set A is defined as

$$\forall x \in X : \mu'_A(x) = 1 - \mu_A(x).$$

Table 18.2 Examples of transformation operators

Very	$\mu_{\tilde{A}}(x) = (\mu_A(x))^n, n > 1$
More/less	$\mu_{\tilde{A}}(x) = (\mu_A(x))^n, 0 < n < 1$
More than (lt)	$M_{lt(A)}(x) = 0 \text{ for } x \geq x_0, x_0 : \mu_A(x_0) = \max \mu_A(x) = 1 - \mu_A(x) \text{ for } x < x_0$
More/less (mt)	$M_{mt(A)}(x) = 0 \text{ for } x \leq x_0, x_0 : \mu_A(x_0) = \max \mu_A(x) = 1 - \mu_A(x) \text{ for } x > x_0$

18.2.3.1 Examples of Fuzzy Set Operations

Let us consider the fuzzy sets A and B :

$$A = \{0/1 + 0.2/2 + 0.8/3 + 1/4 + 1/5\}$$

and

$$B = \{0.1/1 + 0.4/2 + 0.5/3 + 0.7/4 + 0.3/5\}.$$

Then, using the notation A' for the complement of A ,

$$\begin{aligned} A \cap B &= \{0/1 + 0.2/2 + 0.5/3 + 0.7/4 + 0.3/5\} \text{ using the min operator} \\ &= \{0/1 + 0.08/2 + 0.4/3 + 0.7/4 + 0.3/5\} \text{ using the product operator} \\ A \cup B &= \{0.1/1 + 0.4/2 + 0.8/3 + 1/4 + 1/5\} \text{ using the max operator} \\ A' &= \{1/1 + 0.8/2 + 0.2/3 + 0/4 + 0/5\} \\ B' &= \{0.9/1 + 0.6/2 + 0.5/3 + 0.3/4 + 0.7/5\} \\ (A \cap B)' &= A' \cup B' \\ &= \{1/1 + 0.8/2 + 0.5/3 + 0.3/4 + 0.7/5\} \text{ using the max operator.} \end{aligned}$$

18.2.4 Transformation Operators

The transformation operator (or hedge or modifier) acts on a membership function to modify the concept of the linguistic term that describes the fuzzy set. For example, in the clause “number very close to 10”, the transformation operator *very* acts on the linguistic term “close to 10” which corresponds to a fuzzy set. Examples of such operators are given in Table 18.2 (Ross 1995; Zimmermann 1996; Pappis and Mamdani 1977).

18.2.4.1 Example of Transformation Operators

Let us consider the fuzzy set *young* on the discrete set $U = \{0, 20, 40, 60, 80\}$, $F_{\text{young}} = \{(0, 1), (20, 0.75), (40, 0.52), (60, 0.23), (80, 0)\}$. Then we can derive the fuzzy set $F' = \text{very young}$ by using the relevant transformation operator. Choosing $v = 1.5$ we get $F' = \{(0, 1), (20, 0.6495), (40, 0.0.375), (60, 0.1103), (80, 0)\}$.

18.2.5 Cartesian Inner Product of Fuzzy Sets

If A_1, A_2, \dots, A_v are fuzzy sets defined in U_1, U_2, \dots, U_v , their Cartesian inner product is a fuzzy set $F = A_1 \times A_2 \times \dots \times A_v$ in $U_1 \times U_2 \times \dots \times U_v$ with membership function

$$\mu_F(u_1, u_2, \dots, u_v) = \cap_{i=1, v} \mu_{A_i}(u_i).$$

For example,

$$\mu_F(u_1, u_2, \dots, u_v) = \min\{\mu_{A_1}(u_1), \mu_{A_2}(u_2), \dots, \mu_{A_v}(u_v)\}$$

or

$$\mu_F(u_1, u_2, \dots, u_v) = \mu_{A_1}(u_1) \mu_{A_2}(u_2) \dots \mu_{A_v}(u_v).$$

18.2.5.1 Example

The objective in climate control is to find the optimum conditions in terms of both temperature T and humidity H . Suppose that the discrete sets of temperature and humidity are given by $T = \{T_1, T_2, T_3, T_4\}$, $H = \{H_1, H_2, H_3\}$ respectively, that of the desired temperature by the discrete fuzzy set

$$A = 0.12/T_1 + 0.65/T_2 + 1/T_3 + 0.25/T_4$$

and the desired level of humidity by

$$B = 0.5/H_1 + 0.9/H_2 + 0.1/H_3.$$

Then the Cartesian product $A \times B$ reads

$$A \times B = 0.12/T_1, H_1 + 0.12/T_1, H_2 + 0.1/T_1, H_3 + 0.5/T_2, H_1 + 0.65/T_2, H_2 + 0.1/T_2, H_3 + 0.5/T_3, H_1 + 0.9/T_3, H_2 + 0.1/T_3, H_3 + 0.25/T_4, H_1 + 0.25/T_4, H_2 + 0.1/T_4, H_3.$$

Then the optimum conditions are those for $T = T_3$ and $H = H_2$.

18.2.6 Fuzzy Relations

Let U_1 and U_2 be two universes of discourse and the membership function $\mu_R : U_1 \times U_2 \rightarrow [0, 1]$. Then a fuzzy relation R on $U_1 \times U_2$ is defined as (Zimmermann 1996)

$$R = \int_{U_1 \times U_2} \mu_R(u_1, u_2) / (u_1, u_2) \quad \text{if } U_1, U_2 \text{ are continuous}$$

or

$$R_d = \int_{U \times V} \mu_R(u_1, u_2) / (u_1, u_2) \quad \text{if } U_1, U_2 \text{ are discrete.}$$

18.2.6.1 Example

Consider the coordinates of three atoms, denoted by i, j, k in a cubic crystal with a lattice constant of 3 Å and their corresponding x, y, z coordinates

$$U = \{(0, 0, 0), (0.5, 0.5, 0.5), (1.2, 1.2, 1.2)\}$$

(in Å). Then the fuzzy relation near neighbors can be described by the following fuzzy relation:

$$\begin{aligned} R = & 1.0/i, i+1/j, j+1/k, k+0.9/i, j+0.1/i, k+0.9/j, i \\ & +0.6/j, k+0.1/k, i+0.6/k, j. \end{aligned}$$

Note that for the particular problem we should have excluded the pairs (i, i) , (j, j) and (k, k) , but we have kept them for the completeness of the example.

18.2.7 Fuzzy Set Composition

Let R_1 and R_2 be two fuzzy relations on $U_1 \times U_2$ and $U_2 \times U_3$ respectively, then the composition C of R_1 and R_2 is a fuzzy relation defined as follows:

$$C = R_1 \circ R_2 = \{(u_1, u_3), \cup(\mu_{R_1}(u_1, u_2) \cap \mu_{R_2}(u_2, u_3))\}, u_1 \in U_1, u_2 \in U_2, u_3 \in U_3.$$

18.2.7.1 Example

Consider the following fuzzy relations (in a matrix form):

$$R = \begin{bmatrix} 0.2 & 0.6 \\ 0.9 & 0.4 \end{bmatrix} \quad \text{and} \quad S = \begin{bmatrix} 1 & 0.4 & 0.3 \\ 0.8 & 0.5 & 0.1 \end{bmatrix}.$$

Then using a min operator for \cap and a max operator for \cup their composition reads

$$T = R \circ S = \begin{bmatrix} 0.2 & 0.6 \\ 0.9 & 0.4 \end{bmatrix} \circ \begin{bmatrix} 1 & 0.4 & 0.3 \\ 0.8 & 0.5 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.6 & 0.5 & 0.2 \\ 0.9 & 0.4 & 0.3 \end{bmatrix}.$$

18.2.8 Fuzzy Implication

Let A and B be two fuzzy sets in U_1, U_2 respectively. The implication $I : A \Rightarrow B \in U_1 \times U_2$ is defined as (Ross 1995; Zimmermann 1996)

$$I = A \times B = \int_{U_1 \times U_2} \mu_A(u_1) \cap \mu_B(u_2) / (u_1, u_2).$$

The rule “*If* the error is negative big *then* control output is positive big” is an implication: error x implies control action y .

Let the two discrete fuzzy sets be $A = \{(u_i, \mu_A(u_i)), i = 1, \dots, n\}$ defined on U and $B = \{(v_j, \mu_B(v_j)), j = 1, \dots, m\}$ defined on V . Then the implication $A \Rightarrow B$ is a fuzzy relation R :

$$R = \{((u_i, v_j), \mu_R(u_i, v_j)), i = 1, \dots, n, j = 1, \dots, m\}$$

defined on $U \times V$, whose membership function $\mu_R(u_i, v_j)$ is given by

$$\begin{aligned} & \left[\begin{matrix} \mu_A(u_1) \\ \mu_A(u_2) \\ \dots \\ \mu_A(u_n) \end{matrix} \right] \times [\mu_B(v_1) \ \mu_B(v_2) \ \dots \ \mu_B(v_m)] \\ &= \left[\begin{matrix} \mu_A(u_1) \wedge \mu_B(v_1) & \mu_A(u_1) \wedge \mu_B(v_2) & \dots & \mu_A(u_1) \wedge \mu_B(v_m) \\ \mu_A(u_2) \wedge \mu_B(v_1) & \mu_A(u_2) \wedge \mu_B(v_2) & \dots & \mu_A(u_2) \wedge \mu_B(v_m) \\ \dots & \dots & \dots & \dots \\ \mu_A(u_n) \wedge \mu_B(v_1) & \mu_A(u_n) \wedge \mu_B(v_2) & \dots & \mu_A(u_n) \wedge \mu_B(v_m) \end{matrix} \right]. \end{aligned}$$

18.2.9 Inference Rules

Let R be a fuzzy relation on $U_1 \times U_2$ and A be a fuzzy set in U_1 . The composition

$$A \circ R = B$$

is a fuzzy set in U_2 , which represents the conclusion made from the fuzzy set A (fact) based on the implication R (rule).

Let a multiple-input single-output (MISO) rule base with N rules. The i th rule is given by “*If* A_{i1} and A_{i2} and ... and A_{in} *then* B_i ”, where

- n is the number of input variables x_i
- A_{ij} is the fuzzy set of input variable x_j in the i th rule
- B_i is the fuzzy set of output variable y_j in the i th rule.

The i th rule is the implication

$$I_i = A_i \Rightarrow B_i, \quad A_i = A_{i1} \cap A_{i2} \cap \dots \cap A_{in} = \cap_{j=1}^n A_{ij}.$$

Then the implication I_{tot} of N rules is given by

$$I_{\text{tot}} = R_1 \cup R_2 \cup \dots \cup R_N = \cup_{i=1}^N R_i = \cup_{i=1}^N A_i \rightarrow B_i.$$

18.2.10 The Inverse Problem

The inverse problem is defined as follows. Given two fuzzy relations S and T find R such that $R \circ S = T$. In application terms, the problem may be defined as follows: Let S be the input–output relation describing a system and T a desired output of the system. Find input R , which produces T .

[Sanchez \(1976\)](#) showed an existence condition of the solutions associated with their least upper bound and presented a method for obtaining it analytically. [Pappis \(1976\)](#) and [Pappis and Sugeno \(1985\)](#) presented a method to obtain the whole set of solutions.

18.2.11 Fuzzy Similarity Measures

The fuzzy similarity measures introduce the notion of approximate equality (or similarity) between fuzzy sets. The most commonly used fuzzy similarity measures are the following ([Pappis 1991](#); [Pappis and Karacapilidis 1993, 1995](#); [Wang 1997](#); [Cross and Sudkamp 2002](#)):

18.2.11.1 L-Fuzzy Similarity Measure

The $L(A, B)$ similarity measure of two fuzzy sets A, B is defined as

$$L(A, B) = 1 - \max_{x \in X} |A(x) - B(x)|.$$

18.2.11.2 M-Fuzzy Similarity Measure

The $M(A, B)$ similarity measure of two fuzzy sets $A, B \in X$ is defined as

$$M(A, B) = \begin{cases} 1 & \text{if } A = B = \emptyset \\ \frac{\sum_{x \in X} \min(A(x), B(x))}{\sum_{x \in X} \max(A(x), B(x))} & \text{in every other case.} \end{cases}$$

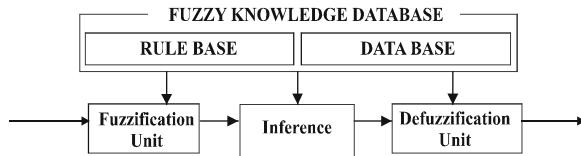


Fig. 18.5 Basic structure of a fuzzy inference system

Two fuzzy sets are ε -almost equal ($A \sim B$) if and only if $M(A, B) \leq \varepsilon$, where $\varepsilon \in [0, 1]$.

18.2.11.3 S-Fuzzy Similarity Measure

The $S(A, B)$ similarity measure of two fuzzy sets $A, B \in X$ is defined as

$$S(A, B) = \begin{cases} 1, & \text{if } A = B = \emptyset \\ 1 - \frac{\sum_{x \in X} |A(x) - B(x)|}{\sum_{x \in X} (A(x) + B(x))} & \text{in every other case.} \end{cases}$$

18.2.11.4 W-Fuzzy Similarity Measure

The $W(A, B)$ similarity measure of two fuzzy sets $A, B \in X$ is defined as

$$W(A, B) = 1 - \sum_{x \in X} |A(x) - B(x)|.$$

18.2.11.5 P-Fuzzy Similarity Measure

The $P(A, B)$ similarity measure of two fuzzy sets $A, B \in X$ is defined as

$$P(A, B) = \frac{\sum_{x \in X} A(x)B(x)}{\max(\sum_{x \in X} A(x)A(x), \sum_{x \in X} B(x)B(x))}.$$

18.3 Basic Structure of a Fuzzy Inference System

The basic structure of a fuzzy inference system consists of a fuzzification unit, a fuzzy logic reasoning unit (process logic), a knowledge base and a defuzzification unit (Fig. 18.5).

The key element of the system is the fuzzy logic reasoning unit that contains two main types of information:

- A data base defining the number, labels and types of the membership functions the fuzzy sets used as values for each system variable. These are of two types: the input and the output variables. For each one of them the designer has to define the corresponding fuzzy sets. The proper selection of these is one of the most critical steps in the design process and can dramatically affect the performance of the system. The fuzzy sets of each variable form the universe of discourse of the variable.
- A rule base, which essentially maps fuzzy values of the inputs to fuzzy values of the outputs. This actually reflects the decision-making policy. The control strategy is stored in the rule base, which in fact is a collection of fuzzy control rules and typically involves weighting and combining a number of fuzzy sets resulting from the fuzzy inference process in a calculation, which gives a single crisp value for each output. The fuzzy rules incorporated in the rule base express the control relationships usually in an IF-THEN format. For instance, for a two-input one-output fuzzy logic controller, that is the case in this work, a control rule has the general form

Rule i: IF x is A_i and y is B_i THEN z is C_i

where x and y are input variables, z is the output variable, and A_i , B_i and C_i are linguistic terms (fuzzy sets) such as *negative*, *positive* or *zero*. The *if* part of the rule is called condition or premise or antecedent, and the *then* part is called the consequence or action.

Usually the actual values acquired from or sent to the system of concern are crisp, and therefore fuzzification and defuzzification operations are needed to map them to and from the fuzzy values used internally by the fuzzy inference system.

The fuzzy reasoning unit performs various fuzzy logic operations to infer the output (decision) from the given fuzzy inputs. During fuzzy inference, the following operations are involved for each fuzzy rule:

1. Determination of the degree of match between the fuzzy input data and the defined fuzzy sets for each system input variable;
2. Calculation of the fire strength (degree of relevance or applicability) for each rule based on the degree of match and the connectives (e.g. AND, OR) used with input variables in the antecedent part of the rule;
3. Derivation of the control outputs based on the calculated fire strength and the defined fuzzy sets for each output variable in the consequent part of each rule.

Several techniques have been proposed for the inference of the fuzzy output based on the rule base. The most common used are the following:

- The Max-Min fuzzy inference method/fuzzy inference method.

Assume that there are two input variables, e (error) and ce (change of error), one output variable, cu (change of output) and two rules:

- Rule1: **If** e is A₁ **AND** ce is B₁ **THEN** cu is C₁.
- Rule2: **If** e is A₂ **AND** ce is B₂ **THEN** cu is C₂.

In the Max-Min inference method, the fuzzy operator **AND** (intersection) means that the minimum value of the antecedents is taken:

$$\mu_A \text{ AND } \mu_B = \min\{\mu_A, \mu_B\},$$

while for the Max-product one the product of the antecedents is taken:

$$\mu_A \text{ AND } \mu_B = \mu_A \mu_B$$

for any two membership values μ_A and μ_B of the fuzzy subsets A, B, respectively. All the contributions of the rules are aggregated using the union operator, thus generating the output fuzzy space C.

18.3.1 Defuzzification Unit

Defuzzification typically involves weighting and combining a number of fuzzy sets resulting from the fuzzy inference process in a calculation, which gives a single crisp value for each output.

The most commonly used defuzzification methods are those of mean of maximum, centroid and center of sum of areas ([Lee 1990; Ross 1995; Driankov et al. 1993](#)).

18.3.1.1 Mean of Maximum Defuzzification Technique

The technique of the mean value of maximum is given by the following equation ([Yan et al. 1994](#)):

$$x = \frac{\sum_{i=1}^n \alpha_i H_i x_i}{\alpha_i H_i},$$

where x is the control (output) value to be applied, n is the number of rules in a MISO system, H_i is the maximum value of the membership function of the output fuzzy set, which corresponds to rule I_i, x_i is the corresponding control (output) value, and α_i is the degree that the rule i is fired.

18.3.1.2 Centroid Defuzzification Technique

This is the most prevalent and intuitively appealing among the defuzzification methods ([Lee 1990; Ross 1995](#)).

This method takes the center of gravity of the final fuzzy space in order to produce a result (the value u of the control variable) sensitive to all rules; it is described

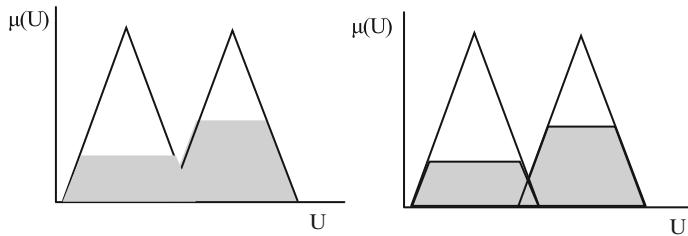


Fig. 18.6 Defuzzification techniques. *Left:* centroid; *right:* center of sums

by the following equation ([Ross 1995](#)):

$$u = \frac{\sum_{i=1}^n \alpha_i M_i}{\sum_{i=1}^n \alpha_i A_i},$$

where M_i is the value of the membership function of the output fuzzy set of rule i , A_i is the corresponding area and α_i is the degree that the rule i is fired. Note that the overlapping areas are merged (Fig. 18.6, left).

In the case of a continuous space (universe of discourse), the output value is given by [Ross \(1995\)](#) and [Taprantzis et al. \(1997\)](#)

$$u = \frac{\int_U u \mu_U(u) du}{\int_U \mu_U(u) du}.$$

18.3.1.3 Center of Sums Defuzzification Technique

A technique similar to the centroid technique but computationally more efficient, in terms of speed, is that of the center of sums. The difference is that the overlapping—between the output fuzzy sets—areas are not merged (Fig. 18.6, right). The discrete value of the output is given by [Lee \(1990\)](#) and [Driankov et al. \(1993\)](#)

$$u = \frac{\sum_{i=1}^1 u_i \cdot \sum_{k=1}^n \mu_k(u_i)}{\sum_{i=1}^1 \sum_{k=1}^n \mu_k(u_i)}.$$

18.3.2 Design of the Rule Base

There are two main approaches in the design of rule bases ([Yan et al. 1994](#)):

1. The heuristic approach
2. The systematic approach.

Table 18.3 A fuzzy rule base with two inputs and one output

		ce						
		NB	NM	NS	Z	PS	PM	PB
e		PB	ZE	PS	NM	NB	NB	NB
e		PM	PS	ZE	NS	NM	NB	NB
e		PS	PM	PS	ZE	NS	NM	NB
e		Z	PB	PM	PS	ZE	NS	NM
e		NS	PB	PM	PS	PS	ZE	NS
e		NM	PB	PB	PM	PM	PS	ZE
e		NB	PB	PB	PB	PB	PM	ZE

Heuristic approaches (Yan et al. 1994; King and Mamdani 1977; Pappis and Mamdani 1977) provide a convenient way to build fuzzy control rules in order to achieve the desired output response, requiring only qualitative knowledge for the behavior of the system under study. For a two-input (e and ce) one-output (cu) system these rules are of the form

- **IF e is P (Positive) AND ce is N (Negative) THEN cu is P (Positive).**
- **IF e is N (Negative) AND ce is P (Positive) THEN cu is N (Negative).**

The reasoning for the construction of the fuzzy control rules can be summarized as follows:

1. If the system output has the desired value and the change of the error (ce) is zero then keep the control action constant.
2. If the system output diverges from the desired value then the control action changes with respect to the sign and the magnitude of the error e and the change of error ce. Table 18.3 compresses the design of a rule base for the linguistic term sets NB (negative big), NM (negative medium), NS (negative small), ZE (zero), PS (positive small), PM (positive medium) and PB (positive big) of the fuzzy variables e, ce, cu. The input variables are laid out along the axes, and each matrix element represents the output variable.

Systematic approaches provide the decision-making strategy (Rule Base) with the aid of system identification and pattern recognition techniques from input-output data.

18.4 Case Study: A Fuzzy Control System

18.4.1 The Fuzzy Logic Control Closed Loop

Over the last 20 years, a large number of conventional modeling and control methods have been proposed to cope with nonlinear and/or time-varying systems including input-state linearization (Isidori 1995), input–output linearization (Cravaris and

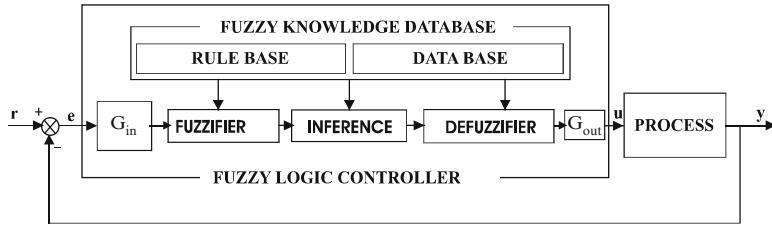


Fig. 18.7 The fuzzy logic control closed loop

(Chung 1987; Henson and Seborg 1990), model predictive schemes and various direct and indirect adaptive control schemes (Isermann 1989; Batur and Kasparian 1991).

However, the poor modeling of system uncertainties and the inherent difficulty of incorporating a priori qualitative information about the system dynamics limit the efficiency and the applicability of the classical approaches. The fuzzy logic approach to process control provides a convenient way to build the control strategy, by requiring only qualitative knowledge for the behavior of the control system. The heuristics employed offer a very attractive way of handling imprecision in the data and/or complex systems, where the derivation of an accurate model is difficult or even impossible. On the other hand, modeling and control techniques based on fuzzy logic comprise very powerful approaches of handling imprecision and non-linearity in complex systems. The basic structure of a fuzzy logic controller is given in Fig. 18.7. Usually the input and output variables are normalized through scaling factors G_{in} and G_{out} in the interval $[-1, 1]$.

18.4.2 Fuzzy Logic Controllers in Proportional-Integral (PI) and Proportional-Differential (PD) Forms

In what follows, in order to enable a comparison basis, the fuzzy logic controller (FLC) with two input variables, the error and the change of error, is represented in PI- and PD-like forms.

18.4.2.1 PI-Like Fuzzy Controller

The PI controller in the z -domain has the following form (Stephanopoulos 1984):

$$C(z) = \frac{u(z)}{e(z)} = K_c \left(1 + K \frac{1}{1 - z^{-1}} \right).$$

In the time domain the above equation can be rewritten as

$$cu = K_c ce + (K_c K)e,$$

where e is the error between a predefined set point and the process output, ce is the change in error, and u is the control output signal. In order to generate an equivalent fuzzy controller, the same inputs e , ce and the same output, cu , will be used in its design.

Based on the above, a two-input single-output FLC is derived with the following variables:

- Input variables: $e(t) = r(t) - y(t)$, $ce(t) = e(t) - e(t-1)$
- Output variable: $cu(t) = u(t) - u(t-1)$

where $r(t)$ is the set point at time t (set point moisture), $y(t)$ is the process output at time t (output moisture), $e(t)$, $ce(t)$ are the error and the change of error at time t , respectively, and $cu(t)$ is the change in the control variable at time t .

In a general form the control action cu can be represented as a nonlinear function of the input variables $e(t)$, $ce(t)$:

$$cu = f(e', ce', t) = f(GEe, GCEce, t).$$

For small perturbations δe , δce around equilibrium, the above equation is approximated by the linearized equation

$$cu = \left[\frac{\partial f}{\partial e} \right]_{ce} \delta e + \left[\frac{\partial f}{\partial ce} \right]_e \delta ce.$$

By substituting, one finally obtains the simplified discretized equation ([Mizumoto 1995](#))

$$cu(t) = GEe(t) + GCEce(t)$$

which gives the incremental control output at time t . GE , GCE are the scaling factors for the error and change of error, respectively.

18.4.2.2 PD-Like Fuzzy Controller

In an analogous manner the PD-like fuzzy controller is of the form

$$u(t) = GEe(t) + GCEce(t).$$

Note that the above expressions are derived using the max-product inference technique.

Fig. 18.8 The process under study: control of a plug flow tubular reactor

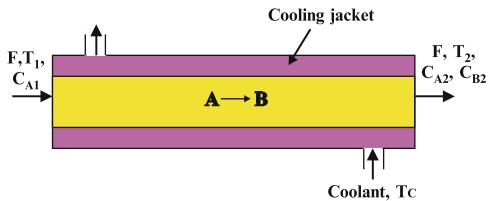


Table 18.4 Nominal values of the tubular reactor parameters

$C_{A1} = 1.6 \text{ kmol/m}^3$	$k_o = 3.34 \cdot 10^8 \text{ min}^{-1}$	$U = 25 \text{ Kcal m}^2 \text{ min}^{-1} \text{ grad}^{-1}$	$A_t = 0.01 \text{ m}^2$
$C_{A2} = 0.11 \text{ kmol/m}^3$	$E/R = 8,600 \text{ K}$	$c_p = 25 \text{ kcal/kmol K}$	$A = 0.002 \text{ m}^2$
$T_1 = 440 \text{ K}$	$DH_R = -44,000 \text{ kcal/kmol}$	$\rho = 47 \text{ kmol/m}^3$	$U = 2 \text{ m/min}$
$T_2 = 423 \text{ K}$		$T_{co} = 293 \text{ K}$	

18.4.3 An Illustrative Example

The design procedure of a FLC is demonstrated through an illustrative example: the system under study is a plug flow tubular reactor, which is a nonlinear distributed parameter with time lag system. The design of the FLC is based on a heuristic approach. The proposed controller is compared with a conventional PI controller, which is tuned with two methods: the process reaction curve tuning method and by using time integral performance criteria such as the integral of absolute error (IAE). Based on dynamic performance criteria, such as IAE, ISE, ITAE (defined later), it is shown that the proposed fuzzy controller exhibits a better performance compared to the PI controller tuned by the process reaction curve tuning method and an equivalent, if not better dynamic behavior, compared to the optimal tuned via the time performance criteria PI controller, for a wide range of disturbances.

18.4.3.1 The Case Study: Fuzzy Control of a Plug Flow Tubular Reactor

The process of concern is shown in Fig. 18.8. It is the problem of the control of a jacketed tubular reactor in which a simple exothermic reaction $A \Rightarrow B$ with first-order kinetics takes place. Assuming plug flow conditions, constant temperature for the coolant, which flows around the tube of the reactor, the governing equations consist of a set of nonlinear time-dependent partial differential equations. The system is a nonlinear distributed parameter with time delay system:

$$\begin{aligned} \frac{\partial C_A}{\partial t} + u \frac{\partial C_A}{\partial z} &= -k C_A \\ c_p \rho A \frac{\partial T}{\partial t} + c_p \rho u A \frac{\partial T}{\partial z} &= h A_t (T_C - T) + (-D H_R) k A C_A \\ k &= k_o \exp \left(-\frac{E}{RT} \right). \end{aligned}$$

The nominal values of the tubular reactor parameters are given in Table 18.4.

The solution of nonlinear, time-dependent, partial differential equations is possible only by means of modern computer-aided methods. The choice here is the combination of Galerkin's method of weighted residuals and finite element basis functions (Zienkiewicz and Morgan 1983).

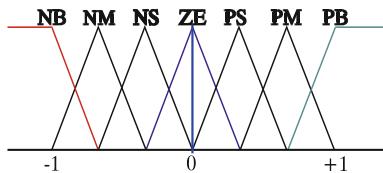
The control objective is to maintain the control variable, which is the composition of the reacting mixture at the output of the reactor, within the desired operational settings and, particularly, to keep the A reactant concentration at the output below its nominal steady state value, eliminating mostly input concentration disturbances. The manipulated variable is taken to be the coolant temperature. The incremental fuzzy controller, a two-input single-output FLC, is derived with the following variables: $e(t) = r(t) - y(t)$, $ce(t) = e(t) - e(t-1)$, $cu(t) = u(t) - u(t-1)$, where $r(t)$ is the set point at time t (set point moisture), $y(t)$ is the process output at time t (output moisture) and $e(t)$, $ce(t)$ are the error and the change of error at time t .

For the fuzzification of the input–output variables, seven fuzzy sets are defined for each variable, $e(t)$, $ce(t)$ and $cu(t)$ with fixed triangular-shaped membership functions normalized in the same universe of discourse, as shown in Fig. 18.9. For the development of the rule base a heuristic approach was employed.

Given the fact that a reduction in the coolant temperature decreases the output concentration, and inversely, the reasoning for the construction of the fuzzy control rules is as follows:

1. Keep the output of the FLC constant if the output has the desired value and the change of error is zero.
2. Change the control action of the FLC according to the values and signs of the error, e , and the change of error, ce :
 - (a) *If* the error is negative (the process output is above the set point) *and* the change of error is negative (at the previous step the controller was driving the system output upwards), *then* the controller should turn its output downwards. Hence, considering negative feedback, the change in control action should be positive, i.e. $cu > 0$, since $u(t) = u(t-1) + cu$.
 - (b) *If* the error is positive (the process output is below the set point) *and* the change of error is positive (at the previous step the controller was driving the system output downwards), *then* the controller should turn its output upwards. Hence, considering negative feedback, the change in control action should be negative, i.e. $cu < 0$, since $u(t) = u(t-1) + cu$.
 - (c) *If* the error is positive (the process output is below the set point) *and* the change of error is negative, implying that at the previous step the controller was driving the system output upwards, trying to correct the control deviation, *then* the controller need not to take any further action.
 - (d) *If* the error is negative (the process output is above the set point) *and* the change of error is positive, implying that at the previous step the controller was driving the system output downwards, *then* the controller need not to take any further action.

Fig. 18.9 Input–output fuzzy sets



		ce						
		nb	nm	ns	ze	ps	pm	pb
e	pb	ze	ze	nm	nb	nb	nb	nb
	pm	ps	ze	ns	nm	nm	nb	nb
	sp	pm	ps	ze	ns	ns	nm	nb
	ze	pb	pm	ps	ze	ns	nm	nb
	ns	pb	pm	ps	ps	ze	ns	nm
	nm	pb	pb	pm	pm	ps	ze	ze
	nb	Pb	Pb	Pb	Pb	pnm	ze	ze

Table 18.5 Fuzzy control rules

Table 18.5 compresses the design of the control rules for the term sets [nb: negative big, nm: negative medium, ns: negative small, ze: zero, ps: positive small, pm: positive medium, pb: positive big] of the fuzzy variables [e , ce , cu].

The input variables are laid out along the axes, and each matrix element represents the output variable. This structure of the rule base provides negative feedback control in order to maintain stability under any condition. For the evaluation of the rules, the fuzzy reasoning unit of the FLC has been developed using the MAX-MIN fuzzy inference method (Lee 1990; Driankov et al. 1993). In the particular FLC, the centroid defuzzification method (Zimmermann 1996; Driankov et al. 1993) is used. Finally, for the projection of the input and output variable values to the normalized universe of discourse, the following values of scaling factors have been chosen: $G_{e(t)} = 5$, $G_{ce(t)} = 45$, $G_{cu(t)} = 2.5$.

18.4.3.2 Performance Analysis: Results and Discussion

To study the performance of the FLC controller, a comparison with a conventional PI controller is made. The parameters of the PI controller are adjusted using two methods of tuning. First it is assumed that the dynamics of the process are poorly known and the tuning of the PI controller is based on the process reaction curve, an empirical tuning method, which provides an experimental model of the process near the operating point. The results of this analysis are: Gain_I = 350, Integral time constant_I = 1.5 min.

In the second approach, the optimal values of the PI controller are determined by minimizing the IAE of the control variable for a predetermined disturbance in input concentration. Here the optimal parameters of the PI controller are adjusted by minimizing the IAE at the +20% step disturbance in input reactant

concentration. The resulting tuning parameters are: $\text{Gain}_{\text{II}} = 155$, Integral time constant $_{\text{II}} = 1.0 \text{ min}$. The relatively large deviation between the parameters obtained by minimizing the IAE and those obtained by the process reaction curve method is rendered to the fact that the process reaction curve method is based on the approximation of the open loop process response by a first-order system plus dead time.

In the case under study, this approximation seems to be rather poor. In order to objectively compare the FLC controller with the conventional PI controller, in addition to the IAE criterion, the integral of time multiplied by the absolute value of error (ITAE) and the integral of the square of the error (ISE) performance criteria are used for both control and manipulated variables.

Simulation results are presented for step change disturbances ranging from 5 % up to 20 % in input reactant concentration. Figure 18.10 depicts in histograms the calculated three dynamic performance criteria IAE, ISE, ITAE for the fuzzy and the PI controller tuned with the two different methods.

The performance criteria are determined for both control and manipulated variables. Based on Fig. 18.10, it is apparent that the overall performance of the FLC seems better compared to the conventional PI controller tuned by the empirical process reaction curve method (controller PI 1) and equivalent, if not better, compared to the optimal PI controller tuned by minimizing the IAE (controller PI 2). The PI 1 controller has the highest values of IAE (Fig. 18.10, top panels), ISE (Fig. 18.10, middle panels) and ITAE (Fig. 18.10, bottom panels) criteria. As is shown, the fuzzy controller exhibits up to 60 % lower IAE (Fig. 18.10, top left), up to 30 % lower ISE (Fig. 18.10, middle right) and up to 200 % lower ITAE (Fig. 18.10, bottom right) in comparison to the PI controller tuned by the process reaction curve method. In comparison to the PI controller, whose parameters are optimally adjusted by minimizing the IAE criterion, the fuzzy controller shows an equivalent, if not better performance, based on IAE, ISE and ITAE criteria for all the range of step disturbances (from 5 % up to 20 %).

However, the approach of optimally adjusting the parameters of the PI controller to some dynamic performance criterion, such as IAE, requires an exact mathematical model of the process, which in real-world processes is very difficult, if not impossible, to derive. In contrast, the design of the fuzzy logic controller is based on a heuristic approach and a mathematical model of the process is not vital.

18.4.4 Fuzzy Adaptive Control Schemes

A major problem encountered in nonlinear and/or time-dependent systems is the degradation of the closed-loop performance as the system shifts away from the initial operational settings. This drawback imposes the need of using adaptive controllers, i.e. controllers which adjust their parameters optimally, according to some objective criteria (Astrom 1983).

Many schemes have been proposed for fuzzy adaptive control, including self-organizing control (Procyk and Mamdani 1979; Siettos et al. 1999b), member-

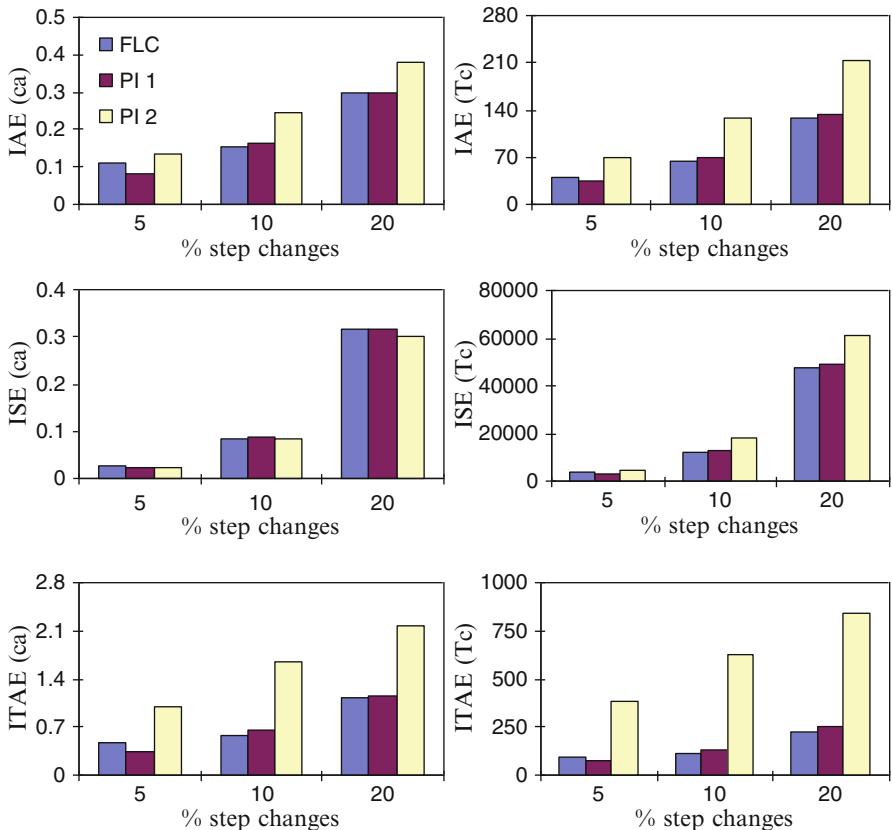


Fig. 18.10 Performance comparison of the fuzzy and the PI controller tuned by the process reaction method (PI 1), by minimizing the IAE criterion (PI 2), top left: IAE of the control variable; top right: IAE of the manipulated variable; middle left: ISE of the control variable; middle right: ISE of the manipulated variable; bottom left: ITAE of the control variable; bottom right: ITAE of the manipulated variable

ship functions adjustment (Batur and Kasparian 1991; Zheng 1992) and scaling factor adjustment (Maeda and Murakami 1992; Daugherty et al. 1992; Palm 1993; Jung et al. 1995; Chou and Lu 1994; Chou 1998; Sagias et al. 2001). Maeda and Murakami and Daugherty et al. proposed adjustment mechanisms for the tuning of scaling factors by evaluating the control result based on system performance indices such as overshoot, rising time, amplitude and settling time. Palm (1993) addressed the method of adjusting optimally the scaling factors by measuring online the linear dependence between each input and output signal of the fuzzy controller. According to the above method the scaling factors are expressed in terms of input-output cross-correlation functions. Jung et al. proposed a real-time tuning of the scaling factors, based on a variable reference tuning index and an instantaneous system fuzzy performance according to system response characteristics. Chou and Lu presented an

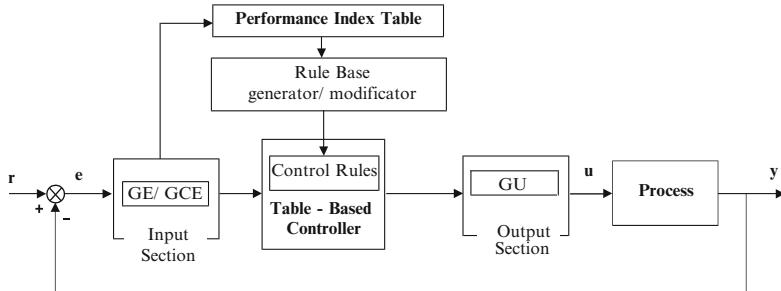


Fig. 18.11 The self-organizing fuzzy logic controller

algorithm for the adjustment of the scaling factors using tuning rules, which are based on heuristics. A model-identification fuzzy adaptive controller for real-time scaling factors adjustment is presented by Sagias et al. (2001).

Among the first attempts to apply a fuzzy adaptive system for the control of dynamic systems was that of Procyk and Mamdani (1979), who introduced the self-organizing controller. The configuration of the proposed controller is shown in Fig. 18.11.

It has a two-level structure, in which the lower level consists of a table-based controller with two inputs and one output. The upper level consists of a performance index table, which relates the state of the process to the deviation from the desired overall response, and defines the corrections required in the table-based controller to bring the system to the desired state. From this point of view, the self-organizing controller performs two tasks simultaneously, namely (a) performance evaluation of the system and (b) system performance improvement by creation and/or modification of the control actions based on experience gained from past system states. Hence, the controller accomplishes its learning through repetition over a sequence of operations. The elements of the table are the control actions as they are calculated from a conventional or a fuzzy controller for a fixed operational range of input variables. Here, the (i, j) element of the table contains the changes in control action inferred for the i th value of error and j th value of change of error.

18.5 Model Identification and Stability of Fuzzy Systems

18.5.1 Fuzzy Systems Modeling

Mathematical models, which can describe efficiently the dynamics of the system under study, play an essential role in process analysis and control. However, most of the real-world processes are complicated and nonlinear in nature, making the derivation of mathematical models and/or subsequent analysis formidable tasks. In practice, such models are not available. For these cases, models need to be developed

based solely on input–output data. Many approaches based on nonlinear time series (Hernandez and Arkun 1993; Ljung 1987), several nonlinear approaches (Henon 1982; Wolf et al. 1985) and normal form theory (Read and Ray 1998a,b,c) have been applied in nonlinear system modeling and analysis. During the last decade, a considerable amount of work has been published on the dynamic modeling of nonlinear systems using neural networks (Narendra and Parthasarathy 1990; Chen and Billings 1992; Shaw et al. 1997; Haykin 1999) and/or fuzzy logic methodologies (Sugeno and Yasukawa 1993; Laukoven and Pasino 1995; Babuška and Verbruggen 1996). Most of them are numerical in nature providing therefore only black-box representations. On the other hand, fuzzy logic methodologies (Laukoven and Pasino 1995; Park et al. 1999; Sugeno and Kang 1988; Sugeno and Yasukawa 1993; Takagi and Sugeno 1985) can incorporate a priori qualitative knowledge of the system dynamics. In Siettos et al. (2001) and Alexandridis et al. (2002) fuzzy logic and Kohonen’s neural networks are combined for the derivation of truncated time series models.

Fuzzy logic can incorporate expertise and a priori qualitative knowledge of the system. In the last 20 years, strikingly results have been obtained by using various fuzzy design methods. In many cases the fuzzy control systems outperform other more traditional approaches. However, the extensive applicability of the former is limited due to the deficiency of formal and systematic design techniques, which can fulfil the two essential requirements of a control system: the requirement for robust stability and that of satisfactory performance. As a consequence, due to the complexity of nonlinear processes, it is difficult to construct a proper fuzzy rule base based only on observation. Moreover, the lack of a mathematical model, which characterizes fuzzy systems, often limits their applicability, since various vital tasks, such as stability analysis, are difficult to accomplish.

18.5.2 Stability of Fuzzy Systems

The problem of designing reliable fuzzy control systems in terms of stability and performance has found a remarkable resonance among engineers and scientists. Various approaches to this problem have been presented. One of the first contributions to this topic was that of Braae and Rutherford (1979), where they utilized the phase plane method for analyzing the stability of a fuzzy system. Kickert and Mamdani (1978) proposed the use of describing functions for the stability analysis of unforced fuzzy control systems. In Kiszka et al. (1985) the notion of the energy of fuzzy relations to investigate the local stability of a free fuzzy dynamic system is introduced. Motivated by the work of Tanaka and Sugeno (1992), many schemes have been proposed for analyzing the stability of fuzzy systems (Feng et al. 1997; Kiriakidis et al. 1998; Leung et al. 1998; Kim et al. 1995; Thathachar and Viswanath 1997; Wang et al. 1996). The main idea behind this approach lies in the decomposition of a global fuzzy model into simpler linear fuzzy models, which locally represent the dynamics of the whole system. In Kiendl and Ruger (1995) and Michels (1997) the

authors proposed numerical methods for the stability analysis of fuzzy controllers in the sense of Lyapunov's direct method. In [Fuh and Tung \(1997\)](#) and [Kandel et al. \(1999\)](#) the stability analysis of fuzzy systems using Popov–Lyapunov techniques is proposed. In recent years, the problem of designing stable, robust and adaptive fuzzy controllers with satisfactory performance based on the sliding-mode approach has attracted much attention ([Chen and Chang 1998](#); [Chen and Chen 1998](#); [Chen and Fukuda 1998](#); [Palm 1992](#); [Tang et al. 1999](#); [Wang 1994](#); [Yi and Chung 1995](#); [Yu et al. 1998](#)). The design of such schemes is based on Lyapunov's direct method. The proposed schemes take advantage of both sliding and fuzzy features. A systematic practical way of deriving analytical expressions for fuzzy systems for use in control, system identification and stability using well-established classical theory methods is presented by [Siettos et al. \(2001\)](#). Finally, in [Siettos and Bafas \(2001\)](#) singular perturbation methods ([Kokotovic et al. 1976](#)) based on a Lyapunov approach are implemented for the derivation of sufficient conditions for the semiglobal stabilization with output tracking of nonlinear systems having internal dynamics, incorporating fuzzy controllers.

18.6 Conclusion and Perspectives

In this chapter an overview of the basics of fuzzy reasoning has been presented. The theory of fuzzy sets has been introduced and definitions concerning the membership function, logical and transformation operators, fuzzy relations, implication and inference rules, and fuzzy similarity measures have been stated. The basic structure of a fuzzy inference system and its elements have been described. Fuzzy control has been introduced and an example of a fuzzy logic controller has been demonstrated, which applies to the control of a plug flow tubular reactor. The issue of fuzzy adaptive control systems has been discussed and the self-organizing scheme has been presented. Subsequently the topics of stability and model identification of fuzzy systems have been outlined and the presentation has concluded with an introduction of fuzzy classification and clustering systems in pattern recognition.

The above are only an elementary attempt to outline a small part of the introductory concepts and areas of interest of fuzzy reasoning, whose theory and applications are fast developing. Indeed, during recent years, the literature on fuzzy logic theory and applications has exploded. Areas of current research include an enormous set of topics, from basic fuzzy set-theoretic concepts and fuzzy mathematics to fuzzy methodology and fuzzy logic in practice ([Dubois and Prade 1980](#)). The statement by H.-J. Zimmermann that “theoretical publications are already so specialized and assume such a background in fuzzy set theory that they are hard to understand” ([Zimmermann 1996](#)) holds much more today than 25 years ago, when it was first made.

In recent years several highly influential papers have appeared that point towards the future directions of fuzzy set theory and applications. Most preferential among these is work reported in the area of fuzzy control ([Sala and Ariño 2007](#); [Tanaka et al. 2007](#); [Fang et al. 2006](#); [Tian and Peng 2006](#); [Guan and Chen 2004](#)). Indeed,

this area has attracted the attention of many researchers since the first years after the introduction of the theory of fuzzy sets by Zadeh in 1965. Apart from fuzzy control, several important papers have appeared that seem to shape the future of this area of research. Among them fuzzy rough sets (Jensen and Shen 2004, 2007; Yeung et al. 2005), genetic fuzzy systems (Herrera 2008), type-2 fuzzy logic systems (Mendel et al. 2006), fuzzy approximation operators (Wu and Zhang 2004), and intuitionistic fuzzy sets (Li 2005) are included. Finally, in the area of applications, the emphasis seems to be on fields like supply chain management (Chan and Kumar 2007; Chen et al. 2006), Internet commerce and online commodity exchanges (Song et al. 2005).

More particularly, in addition to the above, research is continuing on the various basic fuzzy set-theoretic concepts, including possibility theory (Ben Amor et al. 2002; Liu and Liu 2010), fuzzy operators (Pradera et al. 2002; Yager 2002a; Ying 2002; Wang et al. 2003), fuzzy relations (Wang et al. 1995; Naessens et al. 2002; Pedrycz and Vasilakos 2002), measures of information and comparison (Hung 2002; Yager 2002b), genetic fuzzy systems (Cordón et al. 2004), type-2 fuzzy sets (Mendel and John 2002) and fuzzy control systems (Feng 2006).

In the area of fuzzy mathematics, research focuses on various issues of non-classical logics (Biacino and Gerla 2002; Novak 2002), algebra (Di Nola et al. 2002) and topology (Albrecht 2003).

The research on fuzzy methodology is extensive. It encompasses issues related to inference systems (del Amo et al. 2001; Marin-Blazquez and Shen 2002), computational linguistics and knowledge representation (Intan and Mukaidono 2002), production scheduling (Adamopoulos et al. 2000; Karacapilidis et al. 2000), neural networks (Alpaydin et al. 2002; Oh et al. 2002; Wang and Lee 2002), genetic algorithms (Spiegel and Sudkamp 2002; Ishibuchi and Yamamoto 2004), information processing (Liu et al. 2002; Hong et al. 2002; Nikravesh et al. 2002), pattern analysis and classification (Gabrys and Bargiela 2002; de Moraes et al. 2002; Pedrycz and Gacek 2002; Nobuhara et al. 2006), fuzzy systems modeling and control (Mastrokostas and Theocaris 2002; Pomares et al. 2002; Tong et al. 2002; Yi and Heng 2002), decision making (Yager 2002c; Wang 2000; Zimmermann et al. 2000; Wang and Lin 2003), etc.

Finally, extensive research is also reported on various applications of fuzzy logic, including process control (Tamhane et al. 2002), robotics (Lin and Wang 1998; Ruan et al. 2003), scheduling (Muthusamy et al. 2003), transportation (Chou and Teng 2002), nuclear engineering (Kunsch and Fortemps 2002), medicine (Blanco et al. 2002; Kilic et al. 2002; Polat et al. 2006), economics, supply chain management and finance (Kahraman et al. 2002; Afshar and Fathi 2009; Lee 2009). Many other applications of fuzzy logic in various fields are reported in Pappis et al. (2012).

It is this last area and the reported applications of fuzzy reasoning which proves the relevance and vigor of this new approach to understanding, modeling and solving many problems of modern society.

Concluding this chapter, the views of the founder of the theory of fuzzy sets are worthy mentioning. In a paper that appeared in 2008, with the challenging title *Is there a need for fuzzy logic?*, Zadeh notes that the issue is associated with a “long history of spirited discussions and debate” and that there are many misconceptions

about fuzzy logic (Zadeh 2008). According to Zadeh, fuzzy logic is a precise logic, not a fuzzy one, of imprecision and approximate reasoning, with a high power of precisiation, an operation which transforms an object, p , into an object which in some specified sense is defined more precisely than p . It is an attempt to formalize/mechanize two remarkable human capabilities: the capabilities to converse, reason and make rational decisions in an environment of imperfect information, and second, to perform a wide variety of tasks without any measurements and any computations. Among its facets are the logical, fuzzy-set-theoretic, epistemic and relational ones, with the practical applications of fuzzy logic being associated with the latter (relational).

In his paper, Zadeh views fuzzy logic in a nonstandard perspective, where its cornerstones and principal distinguishing features are graduation, granulation, precisiation and the concept of a generalized constraint. After introducing some innovative concepts, Zadeh concludes that, in summary, progression from bivalent logic to fuzzy logic is a significant positive step in the evolution of science as, in large measure, the real-world is a fuzzy world and what is needed in order to deal with fuzzy reality is fuzzy logic. As a consequence, fuzzy logic is likely to grow in visibility, importance and acceptance in coming years.

Tricks of the Trade

Newcomers to the field of fuzzy reasoning often ask themselves (and/or other more experienced fuzzy researchers) questions such as “What is the best way to get started with fuzzy reasoning?”, or “Which papers should I read?”

A very helpful tutorial on fuzzy sets recommended for beginners is available at www.mathworks.com/access/helpdesk/help/toolbox/fuzzy/fuzzytu2.html. This tutorial is a step-by-step introduction to the basic ideas and definitions of fuzzy set theory, with simple and well-designed illustrative examples.

Apart from this tutorial, a variety of sources of information are available, including the first publication on fuzzy reasoning by L. A. Zadeh, the founder of fuzzy logic, which appeared in 1965, as well as his subsequent publications (notably *Outline of a new approach to the analysis of complex systems and decision processes*, IEEE Trans. Syst. Man. Cybern., 3, 28–44, 1973), which inspired so many researchers in this new and fascinating field of research.

Another question often asked is “How should I be acquainted with the world of fuzzy systems and fuzzy reasoning?” This question is best answered by consulting information available on the Web. For example, information useful to practitioners is given at www.cse.dmu.ac.uk/~rij/tools.html about fuzzy logic tools and companies. Information may also be found about books and journals as well as research groups and national and international associations and networks, whose members are researchers and practitioners working on fuzzy sets and systems. For this and other relevant information see the next section.

Sources of Additional Information

A most valuable source of additional information about fuzzy reasoning is the site www.abo.fi/~rfuller/fuzs.html, which includes information on almost everything one might like to know about the world of fuzzy systems and fuzzy reasoning, from L. A. Zadeh, the founder of fuzzy logic, fuzzy national and international associations and networks, personal home pages of fuzzy researchers, and fuzzy-mail archives, to fuzzy logic tools and companies, conferences and workshops on fuzzy systems, fuzzy logic journals and books and research groups. An excellent Internet course on fuzzy logic control and fuzzy clustering from the Technical University of Denmark, Oersted-DTU can be found at www.dtu.dk/service/searchresult.aspx?q=fuzzy.

References

- Adamopoulos GI, Pappis CP, Karacapilidis NI (2000) A methodology for solving a range of sequencing problems with uncertain data. In: Slowinski R, Hapke M (eds) Advances in scheduling and sequencing under fuzziness. Physica, Heidelberg, pp 147–164
- Afshar A, Fathi H (2009) Fuzzy multi-objective optimization of finance-based scheduling for construction projects with uncertainties in cost. Eng Optim 41:1063–1080
- Albrecht RF (2003) Interfaces between fuzzy topological interpretation of fuzzy sets and intervals. Fuzzy Sets Syst 135:11–20
- Alexandridis A, Siettos CI, Sarimveis H, Boudouvis AG, Bafas GV (2002) Modeling of nonlinear process dynamics using kohonen's neural networks. Comput Chem Eng 26:479–486
- Alpaydin G, Dündar G, Balkir S (2002) Evolution-based design of neural fuzzy networks using self-adapting genetic parameters. IEEE Trans Fuzzy Syst 10:211–221
- Assilian S, Mamdani EH (1974) An experiment in linguistic synthesis with a fuzzy logic controller. Int J Man Mach Stud 1:1–13
- Astrom KJ (1983) Theory and applications of adaptive control—a survey. Automatica 19:471–486
- Babuška, R, Verbruggen HB (1996) Neuro-fuzzy methods for nonlinear system identification. Ann Rev Contr 27:73–85 (2003)
- Batur C, Kasparian V (1991) Adaptive expert control. Int J Control 54:867–881
- Ben Amor N, Melloyli K, Benfeshat S, Dubios D, Prade H (2002) A theoretical framework for possibilistic independence in a weakly ordered setting. Int J Uncert Fuzz Knowl Based Syst 10:117–155
- Bezdek JC (1981) Pattern recognition with objective function algorithms. Plenum, London
- Biacino L, Gerla G (2002) Fuzzy logic, continuity and effectiveness. Arch Math Logic 41:643–667

- Blanco A, Pelta DA, Verdegay JL (2002) Applying a fuzzy sets-based heuristic to the protein structure prediction problem. *Int J Intell Syst* 17:629–643
- Braae M, Rutherford DA (1979) Selection of parameters for a fuzzy logic controller. *Fuzzy Sets Syst* 2:185–199
- Chan FTS, Kumar N (2007) Global supplier development considering risk factors using fuzzy extended AHP-based approach. *Omega* 35:417–431
- Chen S, Billings S (1992) Neural networks for nonlinear dynamic system modelling and identification. *Int J Control* 56:319–346
- Chen CL, Chang MH (1998) Optimal design of fuzzy sliding-mode control: a comparative study. *Fuzzy Sets Syst* 93:37–48
- Chen CS, Chen WL (1998) Analysis and design of a stable fuzzy control system. *Fuzzy Sets Syst* 96:21–35
- Chen X, Fukuda T (1998) Robust adaptive quasi-sliding mode controller for discrete-time systems. *Syst Control Lett* 35:165–173
- Chen CT, Lin CT, Huang SF (2006) A fuzzy approach for supplier evaluation and selection in supply chain management. *Int J Prod Econ* 102:289–301
- Cheng JH, Chen SS, Chuang YW (2008) An application of fuzzy delphi and Fuzzy AHP for multi-criteria evaluation model of fourth party logistics. *WSEAS Trans Syst* 7:466–478
- Chou CH (1998) Model reference adaptive fuzzy control: a linguistic approach. *Fuzzy Sets Syst* 96:1–20
- Chou CH, Lu HC (1994) A heuristic self-tuning fuzzy controller. *Fuzzy Sets Syst* 61:249–264
- Chou CH, Teng JC (2002) A fuzzy logic controller for traffic junction signals. *Inform Sci* 143:73–97
- Cordón O, Gomide F, Herrera F, Hoffmann F, Magdalena L (2004) Ten years of genetic fuzzy systems: current framework and new trends. *Fuzzy Sets Syst* 141:5–31
- Cravaris C, Chung C (1987) Nonlinear state feedback synthesis by global input/output linearization. *AIChE J* 33:592–603
- Cross VV, Sudkamp TA (2002) Similarity and compatibility in fuzzy set theory—assessment and applications. *Studies in fuzziness and soft computing*, vol. 93. Springer, Berlin
- Daugherty W, Rathakrishnan B, Yen J (1992) Performance evaluation of a self-tuning fuzzy controller. In: Proceedings of the 1st IEEE international conference on fuzzy systems. San Diego, CA, pp 389–397
- De Moraes RM, Banon GJF, Sandri SA (2002) Fuzzy expert systems architecture for image classification using mathematical morphology operators. *Inform Sci* 142:7–21
- Del Amo A, Comez D, Montero J, Biging G (2001) Relevance and redundancy in fuzzy classification systems. *Mathw Soft Comput* VIII:203–216
- Di Nola A, Esteva F, Garcia P, Godo L, Sessa S (2002) Subvarieties of BL-algebras generated by singlecomponent chains. *Arch Math Logic* 41:673–685
- Driankov D, Hellendoorn H, Reinfrank M (1993) An introduction to fuzzy control. Springer, Berlin

- Dubois D, Prade H (1980) Fuzzy sets and systems: theory and applications. Academic, New York
- Fang CH, Liu YS, Kau SW, Hong L, Lee CH (2006) A new LMI-based approach to relaxed quadratic stabilization of T-S fuzzy control systems. *IEEE Trans Fuzzy Syst* 14:386–397
- Feng G (2006) A survey on analysis and design of model-based fuzzy control systems. *IEEE Trans Fuzzy Syst* 14:676–697
- Feng G, Cao SG, Rees NW, Chak CK (1997) Design of fuzzy control systems with guaranteed stability. *Fuzzy Sets Syst* 85:1–10
- Fuh CC, Tung PC (1997) Robust stability analysis of fuzzy control systems. *Fuzzy Sets Syst* 88:289–298
- Gabrys B, Bargiela A (2002) General fuzzy min-max neural network for clustering and classification. *IEEE Trans Neural Netw* 11:769–783
- Guan XP, Chen CL (2004) Delay-dependent guaranteed cost control for T-S fuzzy systems with time delays. *IEEE Trans Fuzzy Syst* 12:236–249
- Haykin S (1999) Neural networks, 2nd edn. Prentice-Hall, Englewood Cliffs
- Henon M (1982) On the numerical computation of poincaré maps. *Phys D* 5: 412–414
- Henson M, Seborg D (1990) Input–output linearization of general nonlinear processes. *AIChE J* 36:1753–1895
- Hernandez E, Arkun Y (1993) Control of nonlinear systems using polynomial ARMA models. *AIChE J* 39:446–460
- Herrera F (2008) Genetic fuzzy systems: taxonomy, current research trends and prospects. *Evol Intell* 1:27–46
- Hitachi (1984) http://www.hitachi.com/rev/1999/revjun99/r3_109.pdf
- Homblad P, Ostergaard J-J (1982) Control of a cement kiln by fuzzy logic. In: Gupta MM, Sanchez E (eds) Fuzzy information and decision processes. North-Holland, Amsterdam, pp 398–399
- Hong TP, Lin KY, Wang SL (2002) Mining linguistic browsing patterns in the world wide web. *Soft Comput* 6:329–336
- Hung WL (2002) Partial correlation coefficients of intuitionist fuzzy sets. *Int J Uncert Fuzz Knowl Based Syst* 10:105–112
- Intan R, Mukaidono M (2002) On knowledge-based fuzzy sets. *Int J Fuzzy Syst* 4:655–664
- Isermann R (1989) Digital control system II. Springer, Berlin
- Ishibuchi H, Yamamoto T (2004) Fuzzy rule selection by multi-objective genetic local search algorithms and rule evaluation measures in data mining. *Fuzzy Sets Syst* 141:59–88
- Isidori A (1995) Nonlinear control systems, 3rd edn. Springer, Berlin
- Jensen R, Shen Q (2004) Semantics-preserving dimensionality reduction: rough and fuzzy-rough-based approaches. *IEEE Trans Knowl Data Eng* 16:1457–1471
- Jensen R, Shen Q (2007) Fuzzy-rough sets assisted attribute selection. *IEEE Trans Fuzzy Syst* 15:73–89
- Jung CH, Ham CS, Lee KI (1995) A real-time self-tuning controller through scaling factor adjustment for the steam generator of NPP. *Fuzzy Sets Syst* 74:53–60

- Kahraman C, Ruan D, Tolga E (2002) Capital budgeting techniques using discounted fuzzy versus probabilistic cash flows. *Inform Sci* 142:57–56
- Kandel A, Luo Y, Zhang YQ (1999) Stability analysis of fuzzy control systems. *Fuzzy Sets Syst* 105:33–48
- Karacapilidis NI, Pappis CP, Adamopoulos GI (2000) Fuzzy set approaches to lot sizing. In: Slowinski R, Hapke M (eds) *Advances in scheduling and sequencing under fuzziness*. Physica, Heidelberg, pp 291–304
- Karr CL, Gentry EJ (1993) Fuzzy control of pH using genetic algorithms. *IEEE Trans Fuzzy Syst* 1:46–53
- Kickert WM, Mamdani EH (1978) Analysis of a fuzzy logic controller. *Fuzzy Sets Syst* 1:29–44
- Kiendl H, Ruger JJ (1995) Stability analysis of fuzzy control systems using facet functions. *Fuzzy Sets Syst* 70:275–285
- Kilic K, Sproule BA, Türksen IB, Naranjo CA (2002) Fuzzy system modeling in pharmacology: an improved algorithm. *Fuzzy Sets Syst* 130:253–264
- Kim WC, Ahn SC, Kwon WH (1995) Stability analysis and stabilization of fuzzy state space models. *Fuzzy Sets Syst* 71:131–142
- King PJ, Mamdani EH (1977) Analysis of fuzzy control systems to industrial processes. *Automatica* 13:235–242
- Kiriakidis K, Grivas A, Tzes A (1998) Quadratic stability analysis of the Takagi–Sugeno fuzzy model. *Fuzzy Sets Syst* 98:1–14
- Kiszka JB, Gupta MM, Nikiforuk PN (1985) Energetic stability of fuzzy dynamic systems. *IEEE Trans Syst Man Cybern* 15:783–791
- Kokotovic PV, O’Malley RE, Sannuti P (1976) Singular perturbation and order reduction in control theory—an overview. *Automatica* 12:123–132
- Kosko B (1992) Neural networks and fuzzy systems: a dynamical system approach. Prentice-Hall, Englewood Cliffs
- Kunsch PL, Fortemps P (2002) A Fuzzy decision support system for the economic calculus in radioactive waste management. *Inform Sci* 142:103–116
- Laukoven EG, Pasino KM (1995) Training fuzzy systems to perform estimation and identification. *Eng Applic Artif Intell* 8:499–514
- Lee CC (1990) Fuzzy logic in control systems: fuzzy logic controllers-parts I, II. *IEEE Trans Syst Man Cybern* 20:404–435
- Lee AHI (2009) A fuzzy supplier selection model with the consideration of benefits, opportunities, costs and risks. *Expert Syst Appl* 36:2879–2893
- Leung FHF, Lam HK, Tam PKS (1998) Design of fuzzy controllers for uncertain nonlinear systems using stability and robustness analyses. *Syst Control Lett* 35:237–243
- Li DF (2005) Multiattribute decision making models and methods using intuitionistic fuzzy sets. *J Comput Syst Sci* 70:73–85
- Lin CK, Wang SD (1998) A self-organizing fuzzy control approach for bank-to-turn missiles. *Fuzzy Sets Syst* 96:281–306
- Liu ZQ, Liu YK (2010) Type-2 fuzzy variables and their arithmetic. *Soft Comput* 14:729–747

- Liu M, Wan C, Wang L (2002) Content-based audio classification and retrieval using a fuzzy logic system: towards multimedia search engines. *Soft Comput* 6: 357–364
- Ljung L (1987) System identification: theory for the user. Prentice-Hall, Englewood Cliffs
- Maeda M, Murakami S (1992) A self-tuning fuzzy controller. *Fuzzy Sets Syst* 51:29–40
- Marin-Blazquez JG, Qiang Shen (2002) From approximative to descriptive fuzzy classifiers. *IEEE Trans Fuzzy Syst* 10:484–497
- Mastrokostas PA, Theocharis JB (2002) A recurrent fuzzy- neural model for dynamic system identification. *IEEE Trans Syst Man Cybern B* 32:176–190
- Mendel JM, John RIB (2002) Type-2 fuzzy sets made simple. *IEEE Trans Fuzzy Syst* 10:117–127
- Mendel JM, John RIB, Liu F (2006) Interval type-2 fuzzy logic systems made simple. *IEEE Trans Fuzzy Syst* 14:808–821
- Michels K (1997) Numerical stability analysis for a fuzzy or neural network controller. *Fuzzy Sets Syst* 89:335–350
- Mizumoto M (1995) Realization of PID controls by fuzzy control methods. *Fuzzy Sets Syst* 70:171–182
- Muthusamy K, Sung SC, Vlach M, Ishii, H. (2003) Scheduling with fuzzy delays and fuzzy precedences. *Fuzzy Sets Syst* 134:387–395
- Naessens H, De Meyer H, De Baets B (2002) Algorithms for the computation of T-transitive closures. *IEEE Trans Fuzzy Syst* 10:541–551
- Narendra KS, Parthasarathy K (1990) Identification and control of dynamical systems using neural networks. *IEEE Trans Neural Netw* 1:4–27
- Nikravesh M, Loia V, Azvine B (2002) Fuzzy logic and the internet (FLINT): internet, world wide web and search engines. *Soft Comput* 6:287–299
- Nobuhara H, Bede B et al (2006) On various eigen fuzzy sets and their application to image reconstruction. *Inform Sci* 176:2988–3010
- Novak V (2002) Joint consistency of fuzzy theories. *Math Log Q* 48:563–573
- Oh SK, Kim DW, Pedrycz W (2002) Hybrid fuzzy polynomial neural networks. *Int J Uncert Fuzz Knowl Based Syst* 10:257–280
- Østergaard JJ (1990) Fuzzy II: the new generation of high level kiln control. *Zement Kalk Gips* 43(11): 539–541
- Østergaard JJ (1977) Fuzzy logic control of a heat exchange process. In: Gupta MM, Gains BR, Saridis GN (eds) *Fuzzy automata and decision processes*. Elsevier, New York
- Palm R (1992) Sliding mode fuzzy control. In: Proceedings of the 1st IEEE international conference on fuzzy systems. San Diego, CA, pp 519–526
- Palm R (1993) Tuning of scaling factors in fuzzy controllers using correlation functions. In: Proceedings of the 2nd IEEE international conference on fuzzy systems. San Diego, CA, pp 691–696
- Pappis CP (1976) On a fuzzy set theoretic approach to aspects of decision making in ill-defined systems. PhD thesis, University of London

- Pappis CP (1991) Value approximation of fuzzy systems variables. *Fuzzy Sets Syst* 39:111–115
- Pappis CP, Karacapilidis NI (1993) A comparative assessment of measures of similarity of fuzzy values. *Fuzzy Sets Syst* 56:171–174
- Pappis CP, Karacapilidis NI (1995) Application of a similarity measure of fuzzy sets to fuzzy relational equations. *Fuzzy Sets Syst* 75:35–142
- Pappis CP, Mamdani EH (1977) A fuzzy logic controller for a traffic junction. *IEEE Syst Man Cybern SMC-7* 10:707–717
- Pappis CP, Sugeno M (1985) Fuzzy relational equations and the inverse problem. *Fuzzy Sets Syst* 15:79–90
- Pappis CP, Siettos I, Dasaklis TK (2012) Fuzzy sets, systems, and applications. In: Gass S, Fu M (eds) *Encyclopedia of operations research and management science (E/ORMS)* (3/e). Springer, Berlin
- Park M, Ji S, Kim E, Park M (1999) A new approach to the identification of a fuzzy model. *Fuzzy Sets Syst* 104:169–181
- Pedrycz W, Gacek A (2002) Temporal granulation and its application to signal analysis. *Inform Sci* 143:47–71
- Pedrycz W, Vasilakos AV (2002) Modularization of fuzzy relational equations. *Soft Comput* 6:33–37
- Polat K, Şahan S et al. (2006) A new method to medical diagnosis: artificial immune recognition system (AIRS) with fuzzy weighted pre-processing and application to ECG arrhythmia. *Expert Syst Appl* 31:264–269
- Pomares H, Rojas I, Gonzalez J, Prieto A (2002) Structure identification in complete rule-based fuzzy systems. *IEEE Trans Fuzzy Syst* 10:349–359
- Pradera A, Trillas E, Calvo T (2002) A general class of triangular norm-based aggregation operators: quasilinear T-S operators. *Int J Approx Reason* 30:57–72
- Procyk TJ, Mamdani EH (1979) A linguistic self-organizing process controller. *Automatica* 15:15–30
- Read NK, Ray WH (1998a) Application of nonlinear dynamic analysis in the identification and control of nonlinear systems I. Simple dynamics. *J Process Control* 8:1–15
- Read NK, Ray WH (1998b) Application of nonlinear dynamic analysis in the identification and control of nonlinear systems II more complex dynamics. *J Process Control* 8:17–34
- Read NK, Ray WH (1998c) Application of nonlinear dynamic analysis in the identification and control of nonlinear systems III n -dimensional systems. *J Process Control* 8:35–46
- Ross TJ (1995) Fuzzy logic with engineering applications. McGraw-Hill, New York
- Ruan D, Zhou C, Gupta MM (2003) Fuzzy set techniques for intelligent robotic systems. *Fuzzy Sets Syst* 134:1–4
- Sagias DI, Sarafis EN, Siettos CI, Bafas GV (2001) Design of a model identification fuzzy adaptive controller and stability analysis of nonlinear processes. *Fuzzy Sets Syst* 121:169–179

- Sala A, Ariño C (2007) Asymptotically necessary and sufficient conditions for stability and performance in fuzzy control: applications of Polya's theorem. *Fuzzy Sets Syst* 158:2671–2686
- Sanchez E (1976) Resolution of composite fuzzy relational equations. *Inform Control* 30:38–48
- Shaw AM, Doyle III FJ, Schwaber JS (1997) A dynamic neural network approach to nonlinear process modeling. *Comput Chem Eng* 21:371–385
- Siettos CI, Bafas GV (2001) Semiglobal stabilization of nonlinear systems using fuzzy control and singular perturbation methods. *Fuzzy Sets Syst* 129:275–294
- Siettos CI, Boudouvis AG, Bafas GV (1999a) Implementation and performance of a fuzzy adaptive controller for a tubular reactor with limit points. *Syst Anal Model Simul* 38:725–739
- Siettos CI, Kiranoudis CT, Bafas GV (1999b) Advanced control strategies for fluidized bed dryers. *Dry Technol* 17:2271–2292
- Siettos CI, Boudouvis AG, Bafas GV (2001) Approximation of fuzzy control systems using truncated Chebyshev series. *Fuzzy Sets Syst* 126:89–104
- Song S, Hwang K, Zhou R, Kwok Y-K (2005) Trusted P2P transactions with fuzzy reputation aggregation. *IEEE Internet Comput* 9:24–34
- Spiegel D, Sudkamp T (2002) Employing locality in the evolutionary generation of fuzzy rule bases. *IEEE Trans Syst Man Cybern B* 32:296–305
- Stephanopoulos G (1984) Chemical process control: an introduction to theory and practice. Prentice-Hall, Englewood Cliffs
- Sugeno M, Kang GT (1988) Structure identification of fuzzy model. *Fuzzy Sets Syst* 28:15–23
- Sugeno M, Yasukawa T (1993) A fuzzy-logic-based approach to qualitative modelling. *IEEE Trans Fuzzy Syst* 1:7–31
- Takagi T, Sugeno M (1985) Fuzzy identification of systems and its application to modelling and control. *IEEE Trans Syst Man Cybern* 15:116–132
- Tamhane D, Wong PM, Aminzadeh F (2002) Integrating linguistic descriptions and digital signals in petroleum reservoirs. *Int J Fuzzy Syst* 4:586–591
- Tanaka K, Sugeno M (1992) Stability analysis and design of fuzzy control systems. *Fuzzy Sets Syst* 45:135–156
- Tanaka K, Ohtake H, Wang HO (2007) A descriptor system approach to fuzzy control system design via fuzzy Lyapunov functions. *IEEE Trans Fuzzy Syst* 15: 333–341
- Tang Y, Zhang N, Li Y (1999) Stable fuzzy adaptive control for a class of nonlinear systems. *Fuzzy Sets Syst* 104:279–288
- Tapravitis AV, Siettos CI, Bafas GV (1997) Fuzzy control of a fluidized bed dryer. *Dry Technol* 15:511–537
- Thathachar MA, Viswanath P (1997) On the stability of fuzzy systems. *IEEE Trans Fuzzy Syst* 5:145–151
- Tian E, Peng C (2006) Delay-dependent stability analysis and synthesis of uncertain T-S fuzzy systems with time-varying delay. *Fuzzy Sets Syst* 157:544–559
- Togai M, Watanabe H (1986) Expert systems on a chip: an engine for real-time approximate reasoning. *IEEE Expert Mag* 1:55–62

- Tong S, Wang T, Li HX (2002) Fuzzy robust tracking control for uncertain nonlinear systems. *Int J Approx Reason* 30:73–90
- Wang LX (1992) Fuzzy systems are universal approximators. In: Proceedings of 1st IEEE international conference on fuzzy systems. San Diego, CA, pp 1163–1170
- Wang LX (1994) Adaptive fuzzy systems and control: design and stability analysis. Prentice-Hall, Englewood Cliffs
- Wang W (1997) New similarity measures on fuzzy sets and on elements. *Fuzzy Sets Syst* 85:305–309
- Wang HF (2000) Fuzzy multicriteria decision making—an overview. *J Intell Fuzzy Syst* 9:61–84
- Wang JS, Lee CSG (2002) Self-adaptive neuro-fuzzy inference systems for classification applications. *IEEE Trans Fuzzy Syst* 10:790–802
- Wang J, Lin YI (2003) A fuzzy multicriteria group decision making approach to select configuration items for software development. *Fuzzy Sets Syst* 134: 343–363
- Wang W, De Baets B, Kerre E (1995) A comparative study of similarity measures. *Fuzzy Sets Syst* 73:259–268
- Wang HO, Tanaka K, Griffin MF (1996) An approach to fuzzy control of nonlinear systems: stability and design issues. *IEEE Trans Fuzzy Syst* 4:14–23
- Wang SM, Wang BS, Wang GJ (2003) A triangular-norm-based propositional fuzzy logic. *Fuzzy Sets Syst* 136:55–70
- Wolf A, Swift JB, Swinney HL, Vastano JA (1985) Determining Lyapunov exponents from a time series. *Phys D* 16:285–317
- Wu WZ, Zhang WX (2004) Constructive and axiomatic approaches of fuzzy approximation operators. *Inform Sci* 159:233–254
- Yager RR (2002a) On the cardinality index and attitudinal character of fuzzy measures. *Int J Gen Syst* 31:303–329
- Yager RR (2002b) The power average operator. *IEEE Trans Syst Man Cybern A Syst Hum* 31:724–730
- Yager RR (2002c) On the valuation of alternatives for decision-making under uncertainty. *Int J Intell Syst* 17:687–707
- Yan J, Ryan M, Power J (1994) Using fuzzy logic. Prentice-Hall, Englewood Cliffs
- Yeung DS, Chen DG, Tsang ECC, Lee JWT, Wang XZ (2005) On the generalization of fuzzy rough sets. *IEEE Trans Fuzzy Syst* 13:343–361
- Yi SY, Chung MJ (1995) Systematic design and stability analysis of a fuzzy logic controller. *Fuzzy Sets Syst* 72:271–298
- Yi Z, Heng PA (2002) Stability of fuzzy control systems with bounded uncertain delays. *IEEE Trans Fuzzy Syst* 10:92–97
- Ying M (2002) Implication operators in fuzzy logic. *IEEE Trans Fuzzy Syst* 10: 88–91
- Yu X, Man Z, Wu B (1998) Design of fuzzy sliding-mode control systems. *Fuzzy Sets Syst* 95:295–306
- Zadeh LA (1965) Fuzzy sets. *Inform Control* 8:338–353
- Zadeh LA (1973) Outline of a new approach to the analysis complex systems and decision processes. *IEEE Trans Syst Man Cybern* 3:28–44

- Zadeh LA (2008) Is there a need for fuzzy logic? *Inform Sci* 178:2751–2779
- Zheng L (1992) A practical guide to tune of proportional and integral (PI) like fuzzy controllers. In: Proceedings of 1st IEEE international conference on fuzzy systems. San Diego, CA, pp 633–640
- Zienkiewicz OC, Morgan K (1983) Finite elements and approximation. Wiley, New York
- Zimmermann HJ (1996) Fuzzy set theory and its applications, 3rd edn. Kluwer, Dordrecht
- Zimmermann HJ, Ruan D, Huang C (eds) (2000) Fuzzy sets and operations research for decision support: key selected papers. Normal University Press, Beijing

Chapter 19

Rough-Set-Based Decision Support

Roman Słowiński, Salvatore Greco, and Benedetto Matarazzo

19.1 Introduction

In this chapter,¹ we are concerned with the discovery of knowledge from data describing a decision situation. A decision situation is characterized by a set of states or examples, which relate the input with the output of the situation. The aim is to find concise knowledge patterns that summarize a decision situation, and that are useful for explanation of this situation, as well as for the prediction of future similar situations. They are particularly useful in such decision problems as technical or medical diagnostics, performance evaluation and risk assessment. A decision situation is described by a set of *attributes*, which we might also call properties, features, characteristics, etc. Such attributes may be concerned with either the *input* or *output* of a situation or, in other words, with either *conditions* or *decisions*. Within this chapter, we will refer to states or examples of a decision situation as *objects*. The goal of the chapter is to present a knowledge discovery paradigm for multi-attribute and multicriteria decision making, which is based upon the concept of *rough sets*. Rough set theory was introduced by Pawlak (1982, 1991). Since then, it has often proved to be an excellent mathematical tool for the analysis of a *vague* description

R. Słowiński (✉)

Institute of Computing Science, Poznań University of Technology, Poznań,
and Polish Academy of Sciences, Systems Research Institute, Warsaw, Poland
e-mail: roman.slowinski@cs.put.poznan.pl

S. Greco

Department of Economics and Business, University of Catania, Catania, Italy

Portsmouth Business School, Operations & Systems Management University of Portsmouth,
Portsmouth PO1 3DE, United Kingdom

B. Matarazzo

Department of Economics and Business, University of Catania, Catania, Italy

¹ The new material added to the first edition of this chapter is taken from our survey published in Słowiński et al. (2012), with kind permission of the Brazilian Society of Operations Research (SOBRAPO).

of objects. The adjective vague (referring to the quality of information) is concerned with inconsistency or ambiguity. The rough set philosophy is based on the assumption that with every object of the universe U there is associated a certain amount of information (data, knowledge). This information can be expressed by means of a number of attributes. The attributes describe the object. Objects which have the same description are said to be indiscernible (similar) with respect to the available information. The *indiscernibility relation* thus generated constitutes the mathematical basis of rough set theory. It induces a partition of the universe into blocks of indiscernible objects, called elementary sets, which can then be used to build knowledge about a real or abstract world. The use of the indiscernibility relation results in information *granulation*.

Any subset X of the universe may be expressed in terms of these blocks either precisely (as a union of elementary sets) or approximately. In the latter case, the subset X may be characterized by two ordinary sets, called the *lower* and *upper approximations*. A rough set is defined by means of these two approximations, which coincide in the case of an ordinary set. The lower approximation of X is composed of all the elementary sets included in X (whose elements, therefore, certainly belong to X), while the upper approximation of X consists of all the elementary sets which have a non-empty intersection with X (whose elements, therefore, may belong to X). The difference between the upper and lower approximation constitutes the *boundary* region of the rough set, whose elements cannot be characterized with certainty as belonging or not to X (by using the available information). The information about objects from the boundary region is, therefore, inconsistent or ambiguous. The cardinality of the boundary region states, moreover, the extent to which it is possible to express X in exact terms, on the basis of the available information. For this reason, this cardinality may be used as a measure of vagueness of the information about X .

Some important characteristics of the rough set approach makes it a particularly interesting tool in a variety of problems and concrete applications. For example, it is possible to deal with both quantitative and qualitative input data and inconsistencies need not to be removed prior to the analysis. In terms of the output information, it is possible to acquire a posteriori information regarding the relevance of particular attributes and their subsets to the quality of approximation considered within the problem at hand. Moreover, the lower and upper approximations of a partition of U into decision classes, prepare the ground for inducing *certain* and *possible* knowledge patterns in the form of “if . . . , then . . . ” decision rules.

Several attempts have been made to employ rough set theory for decision support (Pawlak and Słowiński 1994; Słowiński 1993). The Indiscernibility-Based Rough Set Approach is not able, however, to deal with preference ordered attribute domains and preference ordered decision classes. In decision analysis, an attribute with a preference ordered domain (scale or value set) is called a *criterion*.

In the late 1990s, adapting the Indiscernibility-Based Rough Set Approach to knowledge discovery from preference ordered data became a particularly challenging problem within the field of multicriteria decision analysis. Why might it be so important? The answer is connected with the nature of the input preference information available in multicriteria decision analysis, and of the output of that analysis.

As to the input, the rough set approach requires a set of decision examples which is also convenient for the acquisition of preference information from decision makers. Very often in multicriteria decision analysis, this information has to be given in terms of preference model parameters, such as importance weights, substitution ratios and various thresholds. Presenting such information requires significant effort on the part of the decision maker. It is generally acknowledged that people often prefer to make exemplary decisions and cannot always explain them in terms of specific parameters. For this reason, the idea of inferring preference models from exemplary decisions provided by the decision maker is very attractive. Furthermore, the exemplary decisions may be inconsistent because of limited clear discrimination between criteria and because of hesitation on the part of the decision maker (see, for example, [Roy 1996](#)). These *inconsistencies* cannot be considered as a simple error or as noise. They can convey important information that should be taken into account in the construction of the decision makers preference model. The rough set approach is intended to deal with inconsistency and this is a major argument to support its application to multicriteria decision analysis. Note also that the output of the analysis, i.e. the model of preferences in terms of decision rules, is very convenient for decision support because it is intelligible and *speaks the same language* as the decision maker.

An extension of the Indiscernibility-Based Rough Set Approach which enables the analysis of preference ordered data was proposed by [Greco et al. \(1998a, 1999a,b\)](#). This extension, called the Dominance-Based Rough Set Approach (DRSA) is mainly based on the substitution of the indiscernibility relation by a dominance relation in the rough approximation of decision classes. An important consequence of this fact is the possibility of inferring (from exemplary decisions) the preference model in terms of decision rules which are logical statements of the type “if..., then...”. The separation of *certain* and *uncertain* knowledge about the decision maker’s preferences is carried out by the distinction of different kinds of decision rules, depending upon whether they are induced from lower approximations of decision classes or from the boundary region, i.e. the difference between upper and lower approximations (composed of inconsistent examples). Such a preference model is more general than the classical functional models considered within multi-attribute utility theory or the relational models considered, for example, in outranking methods.

In the next section, we present the basic version of the Indiscernibility-Based Rough Set Approach by way of an example.

19.2 Rough Set Fundamentals

19.2.1 Explanation by an Example

Let us assume that we want to describe the classification of basic traffic signs to a novice. We start by saying that there are three main classes of traffic signs corresponding to

Table 19.1 Examples of traffic signs described by S and PC

Traffic sign	Shape (S)	Primary Color (PC)	Class
a)	triangle	yellow	W
b)	circle	white	I
c)	circle	blue	I
d)	circle	blue	O

- Warning (W),
- Interdiction (I),
- Order (O).

Then, we say that these classes may be distinguished by such attributes as the shape (S) and the principal color (PC) of the sign. Finally, we give a few examples of traffic signs, like those shown in Table 19.1. These are:

1. Sharp right turn,
2. Speed limit of 50 km/h,
3. No parking,
4. Go ahead.

The sets of signs indiscernible by “Class” are

$$W = \{a\}_{\text{Class}}, I = \{b, c\}_{\text{Class}}, O = \{d\}_{\text{Class}}$$

and the sets of signs indiscernible by S and PC are as follows:

$$\{a\}_{S,PC}, \{b\}_{S,PC}, \{c, d\}_{S,PC}.$$

The above sets are *granules of knowledge* generated by, on the one hand, the classification of traffic signs by “Class” and, on the other hand, their description by S and PC. The sets of signs indiscernible by “Class” are denoted by $\{\bullet\}_{\text{Class}}$ and those by S and PC are denoted by $\{\bullet\}_{S,PC}$. We can see that granule $W=\{a\}_{\text{Class}}$ is characterized precisely by granule $\{a\}_{S,PC}$. In order to characterize granules $I=\{b, c\}_{\text{Class}}$ and $O=\{d\}_{\text{Class}}$, one needs granules $\{b\}_{S,PC}$ and $\{b, c\}_{\text{Class}}$, however, only granule $\{b\}_{S,PC}$ is included in $I=\{b, c\}_{\text{Class}}$ while $\{c, d\}_{S,PC}$ has a non-empty intersection with both $I=\{b, c\}_{\text{Class}}$ and $O=\{d\}_{\text{Class}}$. It follows, from this characterization, that by using attributes S and PC, one can characterize class W precisely, while classes I and O can only be characterized approximately:

- Class W includes sign a certainly, and possibly no other sign,
- Class I includes sign b certainly, and possibly signs b, c and d,

Table 19.2 Examples of traffic signs described by S, PC and SC

Traffic sign	Shape (S)	Primary Color (PC)	Secondary color (SC)	Class
a)	triangle	yellow	red	W
b)	circle	white	red	I
c)	circle	blue	red	I
d)	circle	blue	white	O

- Class O includes no sign certainly, and possibly signs c and d.

The terms “certainly” and “possibly” refer to the absence or presence of ambiguity between the description of signs by S and PC from the one side, and by “Class”, from the other side. In other words, using knowledge about the description of signs by S and PC, one can say that all signs from granules $\{\bullet\}_{S,PC}$ included in granules $\{\bullet\}_{\text{Class}}$ belong certainly to the corresponding class, while all signs from granules $\{\bullet\}_{S,PC}$ having a non-empty intersection with granules $\{\bullet\}_{\text{Class}}$ belong to the corresponding class only possibly. The two sets of certain and possible signs are, respectively, the *lower* and *upper approximations* of the corresponding class by attributes S and PC:

$$\begin{aligned} \text{lower_approx}_{S,PC}(W) &= \{a\}, & \text{upper_approx}_{S,PC}(W) &= \{a\}, \\ \text{lower_approx}_{S,PC}(I) &= \{b\}, & \text{upper_approx}_{S,PC}(I) &= \{b,c,d\}, \\ \text{lower_approx}_{S,PC}(O) &= \emptyset, & \text{upper_approx}_{S,PC}(O) &= \{c,d\}. \end{aligned}$$

The *quality of approximation* of the classification by attributes S and PC is equal to the number of all the signs in the lower approximations divided by the number of all the signs in the table, i.e. 1/2.

One way to increase the quality of the approximation is to add a new attribute to better describe the objects and, consequently, to decrease the ambiguity. Let us introduce the secondary color (SC) as a new attribute. The new situation is now shown in Table 19.2.

As one can see, the sets of signs indiscernible by S, PC and SC, i.e. the granules $\{\bullet\}_{S,PC,SC}$, are now

$$\{a\}_{S,PC,SC}, \{b\}_{S,PC,SC}, \{c\}_{S,PC,SC}, \{d\}_{S,PC,SC}.$$

It is worth noting that the granularity is finer than before and it enables the ambiguity to be eliminated. Consequently, the quality of approximation of the classification by attributes S, PC and SC is now equal to 1.

A natural question occurring here is to ask if, indeed, all three attributes are necessary to characterize precisely the classes W, I and O. When we eliminate attribute

S or attribute PC from the description of the signs, we obtain the granules $\{\bullet\}_{PC,SC}$ or $\{\bullet\}_{S,SC}$, respectively, as follows:

$$\{a\}_{PC,SC}, \{b\}_{PC,SC}, \{c\}_{PC,SC}, \{d\}_{PC,SC}, \{a\}_{S,SC}, \{b,c\}_{S,SC}, \{d\}_{S,SC}.$$

Using any one of the above sets of granules, it is possible to characterize (approximate) classes W, I and O with the same quality (equal to 1) as it is when using the granules $\{\bullet\}_{S,PC,SC}$ (i.e. those generated by the complete set of three attributes). Thus, the answer to the above question is that the three attributes are not all necessary to characterize precisely the classes W, I and O. It is, in fact, sufficient to use either PC and SC or S and SC. The subsets of attributes {PC, SC} and {S, SC} are called *reducts* of {S, PC, SC} because they have this property. Note that the identification of reducts enables us to *reduce* knowledge about the signs from the table to that which is *relevant*.

Other useful information can be generated from the identification of reducts by taking their intersection. This is called the *core*. In our example, the core contains attribute SC. This tells us that it is clearly an *indispensable* attribute, i.e. it cannot be eliminated from the description of the signs without decreasing the quality of the approximation. Note that other attributes from the reducts (i.e. S and PC) are *exchangeable*. If there happened to be some other attributes which were not included in any reduct, then they would be *superfluous*, i.e. they would not be useful at all in the characterization of the classes W, I and O.

If, however, we eliminate column S or PC from Table 19.2 then we still do not have a minimal representation of knowledge about the classification of the four traffic signs. Note that, in order to characterize class W in Table 19.2, it is sufficient to use the descriptor “S = triangle”. Moreover, class I is characterized by two descriptors (“S = circle” and “SC = red”) and class O is characterized by the descriptor “SC = white”. Thus, the minimal representation of this knowledge requires only four descriptors (rather than the eight descriptors that are presented in Table 19.2 with either column S or PC eliminated). This representation corresponds to the following set of *decision rules* which may be seen as knowledge (classification) patterns discovered in the dataset contained in Table 19.2 (in the braces there are symbols of signs covered by the corresponding rule):

- Rule #1: if S=triangle, then Class=W {a},
- Rule #2: if S=circle and SC=red, then Class=I {b,c},
- Rule #3: if SC=white, then Class=O {d}.

This is not the only representation, because an alternative set of rules is

- Rule #1': if PC=yellow, then Class=W {a},
- Rule #2': if PC=white, then Class=I {b},
- Rule #3': if PC=blue and SC=red, then Class=I {c},
- Rule #4': if SC=white, then Class=O {d}.

It is interesting to return to Table 19.1 and to ask what decision rules represent the knowledge contained in this dataset. As the description of the four signs by S

and PC is not sufficient to characterize precisely all the classes, it is not surprising that not all the rules will have a non-ambiguous decision. Indeed, we have

Rule #1": if S=triangle, then Class=W {a}

Rule #2": if PC=white, then Class=I {b}

Rule #3": if PC=blue, then Class=I or O {c,d}.

Note that these rules can be induced from the lower and upper approximations of classes W, I, O defined above. Indeed, for rule #1", the supporting example is in $\text{lower}_{\text{appx}, \text{S,PC}}(\text{W}) = \{a\}$, for rule #2" it is in $\text{lower}_{\text{appx}, \text{S,PC}}(\text{I}) = \{b\}$ and the supporting examples for rule #3" are in the set called the *boundary* of both I and O:

$$\text{Boundary}_{\text{S,PC}}(\text{I}) = \text{upper}_{\text{appx}, \text{S,PC}}(\text{I}) - \text{lower}_{\text{appx}, \text{S,PC}}(\text{I}) = \{c, d\},$$

$$\text{Boundary}_{\text{S,PC}}(\text{O}) = \text{upper}_{\text{appx}, \text{S,PC}}(\text{O}) - \text{lower}_{\text{appx}, \text{S,PC}}(\text{O}) = \{c, d\}.$$

As a result of the approximate characterization of classes W, I and O by S and PC, we can obtain an approximate representation in terms of decision rules. Since the quality of the approximation is 1/2, *certain rules* (#1" and #2") cover one-half of the examples and the other half is covered by the *approximate rule* (#3"). Now, the quality of approximation by S and SC or by PC and SC was equal to 1, so all examples were covered by certain rules (#1 to #3 or #1' to #4', respectively).

We can see from this simple example that the rough set analysis of a dataset provides some useful information. In particular, we can determine:

- A characterization of decision classes in terms of chosen attributes through lower and upper approximation.
- A measure of the quality of approximation which indicates how good the chosen set of attributes is for approximation of the classification.
- A reduction of the knowledge contained in the table to a description by relevant attributes, i.e. those belonging to reducts; at the same time, exchangeable and superfluous attributes are also identified.
- The core which indicates indispensable attributes.
- A set of decision rules which is induced from the lower and upper approximations of the decision classes; this shows classification patterns which exist in the dataset.

Other important information can also be induced but it cannot be illustrated by such a simple example. In the next section, we will present a more formal treatment. For more details, the reader is referred to Pawlak (1991), Polkowski (2002), Slowinski (1992b) and many others (see the *Sources of Additional Information* section at the end of the chapter).

19.2.2 A Formal Description of the Indiscernibility-Based Rough Set Approach

For algorithmic reasons, we supply the information regarding the objects in the form of a data table, whose separate rows refer to distinct *objects* and whose columns refer

to the different *attributes* considered. Each cell of this table indicates an *evaluation* (quantitative or qualitative) of the object placed in that row by means of the attribute in the corresponding column.

Formally, a data table is the 4-tuple $\mathbf{S} = \langle U, Q, V, f \rangle$, where U is a finite set of *objects* (universe), $Q = \{q_1, q_2, \dots, q_m\}$ is a finite set of *attributes*, V_q is the domain (value set) of the attribute q , $V = \bigcup_{q \in Q} V_q$ and $f : U \times Q \rightarrow V$ is a total function such that $f(x, q) \in V_q$ for each $q \in Q$, $x \in U$, called the *information function*.

Each object x of U is described by a vector (string)

$$\text{Des}_Q(x) = [f(x, q_1), f(x, q_2), \dots, f(x, q_m)]$$

called the *description* of x in terms of the evaluations of the attributes from Q . It represents the available information about x .

To every (non-empty) subset of attributes P we associate an *indiscernibility relation* on U , denoted by I_P and defined as follows:

$$I_P = \{(x, y) \in U \times U : f(x, q) = f(y, q) \text{ for each } q \in P\}.$$

If $(x, y) \in I_P$, we say that the objects x and y are P -indiscernible. Clearly, the indiscernibility relation thus defined is an equivalence relation (reflexive, symmetric and transitive). The family of all the equivalence classes of the relation I_P is denoted by UI_P and the equivalence class containing an element $x \in U$ is denoted by $I_P(x)$. The equivalence classes of the relation I_P are called the *P-elementary sets* or *granules of knowledge* encoded by P .

Let \mathbf{S} be a data table, X be a non-empty subset of U and $\emptyset \neq P \subseteq Q$.

The set X may be characterized by two ordinary sets, called the *P-lower approximation* of X (denoted by $\underline{P}(X)$) and the *P-upper approximation* of X (denoted by $\overline{P}(X)$) in \mathbf{S} . They can be defined, respectively, as

$$\underline{P}(X) = \{x \in U : I_P(x) \subseteq X\}, \quad \overline{P}(X) = \{x \in U : I_P(x) \cap X \neq \emptyset\} = \bigcup_{x \in X} I_P(x).$$

The family of all the sets $X \subseteq U$ having the same P -lower and P -upper approximations is called a *P-rough set*. The elements of $\underline{P}(X)$ are all and only those objects $x \in U$ which belong to the equivalence classes generated by the indiscernibility relation I_P contained in X . The elements of $\overline{P}(X)$ are all and only those objects $x \in U$ which belong to the equivalence classes generated by the indiscernibility relation I_P containing at least one object x belonging to X . In other words, $\underline{P}(X)$ is the largest union of the P -elementary sets included in X , while $\overline{P}(X)$ is the smallest union of the P -elementary sets containing X .

The *P-boundary* of X in \mathbf{S} , denoted by $Bn_P(X)$, is defined as

$$Bn_P(X) = \overline{P}(X) - \underline{P}(X).$$

The term *rough approximation* is a general term used to express the operation of the P -lower and P -upper approximation of a set or of a union of sets. The rough approximations obey the following basic laws (see Pawlak 1991):

- The *inclusion property*: $\underline{P}(X) \subseteq X \subseteq \bar{P}(X)$,
- The *complementarity property*: $\underline{P}(X) = U - \bar{P}(U - X)$.

Therefore, if an object x belongs to $\underline{P}(X)$, it is also *certainly* contained in X , while if x belongs to $\bar{P}(X)$, it is only *possibly* contained in X . $Bn_P(X)$ constitutes the *doubtful region* of X : using the knowledge encoded by P nothing can be said with certainty about the inclusion of its elements in set X .

If the P -boundary of X is empty (i.e. $Bn_P(X) = \emptyset$) then the set X is an ordinary set, called the P -exact set. By this, we mean that it may be expressed as the union of some P -elementary sets. Otherwise, if $Bn_P(X) \neq \emptyset$, then the set X is a P -rough set and may be characterized by means of $\underline{P}(X)$ and $\bar{P}(X)$.

The following ratio defines an *accuracy* measure of the approximation of X ($X \neq \emptyset$) by means of the attributes from P :

$$\alpha_P(X) = \frac{|\underline{P}(X)|}{|\bar{P}(X)|},$$

where $|Y|$ denotes the cardinality of a (finite) set Y . Obviously, $0 \leq \alpha_P(X) \leq 1$. If $\alpha_P(X) = 1$, then X is a P -exact set. If $\alpha_P(X) < 1$, then X is a P -rough set.

Another ratio defines a *quality* measure of the approximation of X by means of the attributes from P :

$$\gamma_P(X) = \frac{|\underline{P}(X)|}{|X|}.$$

The quality $\gamma_P(X)$ represents the relative frequency of the objects correctly assigned by means of the attributes from P . Moreover, $0 \leq \alpha_P(X) \leq \gamma_P(X) \leq 1$, and $\gamma_P(X) = 0$ iff $\alpha_P(X) = 0$, while $\gamma_P(X) = 1$ iff $\alpha_P(X) = 1$.

The definition of approximations of a subset $X \subseteq U$ can be extended to a classification, i.e. a partition $Y = \{Y_1, \dots, Y_n\}$ of U . The subsets Y_i , $i = 1, \dots, n$, are disjunctive classes of Y . By the P -lower and P -upper approximations of Y in S we mean the sets

$$\underline{P}(Y) = \{\underline{P}(Y_1), \dots, \underline{P}(Y_n)\} \quad \text{and} \quad \bar{P}(Y) = \{\bar{P}(Y_1), \dots, \bar{P}(Y_n)\}$$

respectively. The coefficient

$$\gamma_P(Y) = \frac{\sum_{i=1}^n |\underline{P}(Y_i)|}{|U|}$$

is called the *quality of approximation of classification* Y by the set of attributes P , or in short, the *quality of classification*. It expresses the ratio of all P -correctly classified objects to all objects in the data table.

The main issue in rough set theory is the approximation of subsets or partitions of U , representing *knowledge* about U , with other sets or partitions that have been built up using available information about U . From the perspective of a particular object $x \in U$, it may be interesting, however, to use the available information

to assess the degree of its membership to a subset X of U . The subset X can be identified with the knowledge to be approximated. Using the rough set approach one can calculate the membership function $\mu_X^P(x)$ (*rough membership function*) as

$$\mu_X^P(x) = \frac{|X \cap I_P(x)|}{|I_P(x)|}.$$

The value of $\mu_X^P(x)$ may be interpreted analogously as conditional probability and may be understood as the *degree of certainty* (credibility) to which x belongs to X . Observe that the value of the membership function is calculated from the available data, and not subjectively assumed, as it is in the case of membership functions of fuzzy sets.

Between the rough membership function and the rough approximations of X the following relationships hold:

$$\begin{aligned} P(X) &= \{x \in U : \mu_X^P(x) = 1\}, \\ \overline{P}(X) &= \{x \in U : \mu_X^P(x) > 0\}, \\ BnP(X) &= \{x \in U : 0 < \mu_X^P(x) < 1\}, \\ P(U - X) &= \{x \in U : \mu_X^P(x) = 0\}. \end{aligned}$$

In rough set theory there is, therefore, a close link between the granularity connected with the rough approximation of sets and the uncertainty connected with the rough membership of objects to sets.

A very important concept for concrete applications is that of the dependence of attributes. Intuitively, a set of attributes $T \subseteq Q$ *totally depends* upon a set of attributes $P \subseteq Q$ if all the values of the attributes from T are uniquely determined by the values of the attributes from P . In other words, this is the case if a functional dependence exists between evaluations by the attributes from P and by the attributes from T . This means that the partition (granularity) generated by the attributes from P is at least as fine as that generated by the attributes from T , so that it is sufficient to use the attributes from P to build the partition U/I_T . Formally, T totally depends on P iff $I_P \subseteq I_T$.

Therefore, T is totally (partially) dependent on P if all (some) objects of the universe U may be univocally assigned to granules of the partition U/I_T , using only the attributes from P .

Another issue of great practical importance is that of *knowledge reduction*. This concerns the elimination of superfluous data from the data table, without deteriorating the information contained in the original table.

Let $P \subseteq Q$ and $p \in P$. It is said that attribute p is *superfluous* in P if $I_P = I_{P-\{p\}}$; otherwise, p is *indispensable* in P .

The set P is *independent* if all its attributes are indispensable. The subset P' of P is a *reduct* of P (denoted by $RED(P)$) if P' is independent and $I_{P'} = I_P$.

A reduct of P may also be defined with respect to an approximation of the classification Y of objects from U . It is then called a *Y -reduct* of P (denoted by $RED_Y(P)$) and it specifies a minimal (with respect to inclusion) subset P' of P which keeps

the quality of the classification unchanged, i.e. $\gamma_{P'}(Y) = \gamma_P(Y)$. In other words, the attributes that do not belong to a Y -reduct of P are superfluous with respect to the classification Y of objects from U .

More than one Y -reduct (or reduct) of P may exist in a data table. The set containing all the indispensable attributes of P is known as the Y -core (denoted by $CORE_Y(P)$). In formal terms, $CORE_Y(P) = \bigcap RED_Y(P)$. Obviously, since the Y -core is the intersection of all the Y -reducts of P , it is included in every Y -reduct of P . It is the most important subset of attributes of Q , because none of its elements can be removed without deteriorating the quality of the classification.

19.2.3 Decision Rules Induced from Rough Approximations

In a data table the attributes of the set Q are often divided into *condition* attributes (set $C \neq \emptyset$) and *decision* attributes (set $D \neq \emptyset$). Note that $C \cup D = Q$ and $C \cap D = \emptyset$. Such a table is called a *decision table*. The decision attributes induce a partition of U deduced from the indiscernibility relation I_D in a way that is independent of the condition attributes. D -elementary sets are called *decision classes*. There is a tendency to reduce the set C while keeping unchanged all important relationships between C and D , in order to make decisions on the basis of a smaller amount of information. When the set of condition attributes is replaced by one of its reducts, the quality of approximation of the classification induced by the decision attributes does not deteriorate.

Since the tendency is to underline the functional dependencies between condition and decision attributes, a decision table may also be seen as a set of *decision rules*. These are logical statements of the type “if..., then...”, where the antecedent (condition part) specifies values assumed by one or more condition attributes (describing C -elementary sets) and the consequence (decision part) specifies an assignment to one or more decision classes (describing D -elementary sets). Therefore, the syntax of a rule can be outlined as follows:

if $f(x, q_1)$ is equal to r_{q1} *and* $f(x, q_2)$ is equal to r_{q2} *and* ... $f(x, q_p)$ is equal to r_{qp}
then x belongs to Y_{j1} *or* Y_{j2} *or* ... Y_{jk}

where

$$\{q_1, q_2, \dots, q_p\} \subseteq C, (r_{q1}, r_{q2}, \dots, r_{qp}) \in V_{q1} \times V_{q2} \times \dots \times V_{qp} \text{ and } Y_{j1}, Y_{j2}, \dots, Y_{jk}$$

are some decision classes of the considered classification (D -elementary sets). If there is only one possible consequence, i.e. $k = 1$, then the rule is said to be *certain*, otherwise it is said to be *approximate* or *ambiguous*.

An object $x \in U$ supports decision rule r if its description is matching both the condition part and the decision part of the rule. We also say that decision rule r covers object x if it matches at least the condition part of the rule. Each decision

rule is characterized by its *strength*, defined as the number of objects supporting the rule. In the case of approximate rules, the strength is calculated for each possible decision class separately.

Let us observe that certain rules are supported only by objects from the lower approximation of the corresponding decision class. Approximate rules are supported, in turn, only by objects from the boundaries of the corresponding decision classes.

Procedures for the generation of decision rules from a decision table use an *inductive learning* principle. The objects are considered as examples of decisions. In order to induce decision rules with a unique consequent assignment to a D -elementary set, the examples belonging to the D -elementary set are called *positive* and all the others *negative*. A decision rule is *discriminant* if it is consistent (i.e. if it distinguishes positive examples from negative ones) and *minimal* (i.e. if removing any attribute from a condition part gives a rule covering negative objects). It may be also interesting to look for *partly discriminant* rules. These are rules that, besides positive examples, could cover a limited number of negative ones. They are characterized by a coefficient, called the *level of confidence*, which is the ratio of the number of positive examples (supporting the rule) to the number of all examples covered by the rule.

The generation of decision rules from decision tables is a complex task and a number of procedures have been proposed to solve it (see, for example, Grzymala-Busse 1992, 1997; Skowron 1993; Ziarko and Shan 1994; Skowron and Polkowski 1997; Stefanowski 1998; Słowiński et al. 2000). The existing induction algorithms use one of the following strategies:

1. The generation of a minimal set of rules covering all objects from a decision table.
2. The generation of an exhaustive set of rules consisting of all possible rules for a decision table.
3. The generation of a set of *strong* decision rules, even partly discriminant, covering relatively many objects from the decision table (but not necessarily all of them).

Internet links to freely available software implementations of these algorithms can be found in the last section of this chapter.

19.2.4 From Indiscernibility to Similarity

As mentioned above, the classical definitions of lower and upper approximations are based on the use of the binary indiscernibility relation which is an equivalence relation. The indiscernibility implies the impossibility of distinguishing between two objects of U having the *same* description in terms of the attributes from Q . This relation induces equivalence classes on U , which constitute the basic granules of knowledge. In reality, due to the imprecision of data describing the objects, small differences are often not considered significant for the purpose of discrimination.

This situation may be formally modeled by considering similarity or tolerance relations (see e.g. Nieminen 1988; Marcus 1994; Slowinski 1992a; Polkowski et al. 1995; Skowron and Stepaniuk 1995; Slowinski and Vanderpooten 1997; Slowinski and Vanderpooten 2000; Stepaniuk 2000; Yao and Wong 1995).

Replacing the indiscernibility relation by a weaker binary *similarity* relation has considerably extended the capacity of the rough set approach. This is because, in the least demanding case, the similarity relation requires reflexivity only, relaxing the assumptions of symmetry and transitivity of the indiscernibility relation.

In general, a similarity relation R does not generate a partition but a cover of U . The information regarding similarity may be represented using *similarity classes* for each object $x \in U$. More precisely, the similarity class of x , denoted by $R(x)$, consists of the set of objects which are similar to x :

$$R(x) = \{y \in U : yRx\}.$$

It is obvious that an object y may be similar to both x and z , while z is not similar to x , i.e. $y \in R(x)$ and $y \in R(z)$, but $z \notin R(x)$, $x, y, z \in U$. The similarity relation is of course reflexive (each object is similar to itself). Slowinski and Vanderpooten (1995, 2000) have proposed a *similarity* relation which is only *reflexive*. The abandonment of the transitivity requirement is easily justifiable. For example, see Luce's paradox of the cups of tea (Luce 1956). As for the symmetry, one should notice that yRx , which means "y is similar to x ", is directional. There is a *subject* y and a *referent* x , and in general this is not equivalent to the proposition " x is similar to y ", as maintained by Tversky (1977). This is quite immediate when the similarity relation is defined in terms of a percentage difference between evaluations of the objects compared on a numerical attribute in hand, calculated with respect to evaluation of the referent object. Therefore, the symmetry of the similarity relation should not be imposed. It then makes sense to consider the inverse relation of R , denoted by R^{-1} , where $xR^{-1}y$ means again "y is similar to x ". $R^{-1}(x)$, $x \in U$, is the class of referent objects to which x is similar:

$$R^{-1}(x) = \{y \in U : xRy\}.$$

Given a subset $X \subseteq U$ and a similarity relation R on U , an object $x \in U$ is said to be *non-ambiguous* in each of the two following cases:

- x belongs to X without ambiguity, that is $x \in X$ and $R^{-1}(x) \subseteq X$; such objects are also called *positive*;
- x does not belong to X without ambiguity (x clearly does not belong to X), that is $x \in U - X$ and $R^{-1}(x) \subseteq U - X$ (or $R^{-1}(x) \cap X = \emptyset$); such objects are also called *negative*.

The objects which are neither positive nor negative are said to be *ambiguous*. A more general definition of lower and upper approximation may thus be offered (see Slowinski and Vanderpooten 2000). Let $X \subseteq U$ and let R be a reflexive binary relation defined on U . The lower approximation of X , denoted by $\underline{R}(X)$, and the upper approximation of X , denoted by $\overline{R}(X)$, are defined, respectively, as

$$\underline{R}(X) = \{x \in U : R^{-1}(x) \subseteq X\}, \quad \overline{R}(X) = \bigcup_{x \in X} R(x).$$

It may be demonstrated that the key properties—*inclusion* and *complementarity*—still hold and that

$$\overline{R}(X) = \{x \in U : R^{-1}(x) \cap X \neq \emptyset\}.$$

Moreover, the above definition of rough approximation is the *only one* that correctly characterizes the set of positive objects (lower approximation) and the set of positive or ambiguous objects (upper approximation) when a similarity relation is reflexive, but not necessarily symmetric nor transitive.

Using a similarity relation, we are able to induce decision rules from a decision table. The syntax of a rule is represented as follows:

$$\begin{aligned} &\text{if } f(x, q_1) \text{ is similar to } r_{q1} \text{ and } f(x, q_2) \text{ is similar to } r_{q2} \text{ and } \dots \\ &\quad f(x, q_p) \text{ is similar to } r_{qp}, \text{ then } x \text{ belongs to } Y_{j1} \text{ or } Y_{j2} \text{ or } \dots Y_{jk}, \end{aligned}$$

where

$$\begin{aligned} &\{q_1, q_2, \dots, q_p\} \subseteq C, \\ &(r_{q1}, r_{q2}, \dots, r_{qp}) \in V_{q1} \times V_{q2} \times \dots \times V_{qp} \end{aligned}$$

and $Y_{j1}, Y_{j2}, \dots, Y_{jk}$ are some classes of the considered classification (D -elementary sets). As mentioned above, if $k = 1$ then the rule is *certain*, otherwise it is *approximate* or *ambiguous*. Procedures for generation of decision rules follow the induction principle described in Sect. 19.2.3. One such procedure has been proposed by Krawiec et al. (1998)—it involves a similarity relation that is learned from data. We would also like to point out that Greco et al. (1998b, 2000b) proposed a fuzzy extension of the similarity, that is, rough approximation of fuzzy sets (decision classes) by means of fuzzy similarity relations (reflexive only).

19.3 The Knowledge Discovery Paradigm and Prior Knowledge

The dataset in which classification patterns are searched for is called the *learning sample*. The learning of patterns from this sample should take into account available *prior knowledge* that may include the following items (see Słowiński et al. 2000a):

- (i) Domains of attributes, i.e. sets of values that an attribute may take while being meaningful to the user.
- (ii) A division of attributes into condition and decision attributes which restricts the range of patterns to functional relations between condition and decision attributes.
- (iii) A preference order in the domains of some attributes and a semantic correlation between pairs of these attributes, requiring the patterns to observe the dominance principle.

In fact, item (i) is usually taken into account in knowledge discovery. With this prior knowledge only, one can discover patterns called *association rules* (Agrawal et al. 1996) which show strong relationships between values of some attributes, without fixing which attributes will be on the condition and which ones on the decision side in all rules.

If item (i) is combined with item (ii) in the prior knowledge, then one can consider a partition of the learning sample into decision classes defined by decision attributes. The patterns to be discovered have then the form of *decision trees* or *decision rules*, representing functional relations between condition and decision attributes. These patterns are typically discovered by machine learning and data mining methods (Michalski et al. 1998). Since there is a direct correspondence between a decision tree and rules, we will concentrate our attention on decision rules only.

As item (iii) is crucial for decision support, let us explain it in more detail. Consider an example of a data set concerning pupils' achievements in a high school. Suppose that among the attributes describing the pupils there are results in mathematics (*Math*) and physics (*Ph*). There is also a general achievement (*GA*) result. The domains of these attributes are composed of three values: *bad*, *medium* and *good*. This information constitutes item (i) of prior knowledge. Item (ii) is also available because, clearly, *Math* and *Ph* are condition attributes while *GA* is a decision attribute. The preference order of the attribute values is obvious: *good* is better than *medium* and *bad*, and *medium* is better than *bad*. It is known, moreover, that both *Math* and *Ph* are semantically correlated with *GA*. This is, precisely, item (iii) of the prior knowledge.

Attributes with preference ordered domains are called *criteria* because they involve an evaluation. We will use the name of *regular attributes* for those attributes whose domains are not preference-ordered. *Semantic correlation between two criteria* (condition and decision) means that an improvement on one criterion should not worsen the evaluation on the second criterion, while other attributes and criteria are unchanged (*monotonicity constraint*). In our example, an improvement of a pupil's score in *Math* or *Ph*, with other attribute values unchanged, should not worsen the pupil's general achievement (*GA*), but rather improve it. In general, semantic correlation between condition criteria and decision criteria requires that an object *x* dominating object *y* on all condition criteria (i.e. *x* having evaluations at least as good as *y* on all condition criteria) should also dominate *y* on all decision criteria (i.e. *x* should have evaluations at least as good as *y* on all decision criteria). This principle is called the *dominance principle* (or Pareto principle) and it is the only objective principle that is widely agreed upon in the multicriteria comparisons of objects. An alternative name of the classification problem with semantic correlation between condition and decision criteria is *ordinal classification with monotonicity constraints*.

Let us consider two questions:

- What classification patterns can be drawn from the pupils' data set?
- How does item (iii) influence the classification patterns?

The answer to the first question is: monotonic “if . . . , then . . . ” decision rules. Each decision rule is characterized by a *condition profile* and a *decision profile*, corresponding to vectors of threshold values of regular attributes and criteria in the condition and decision parts of the rule, respectively. The answer to the second question is that condition and decision profiles of a decision rule should observe the dominance principle (monotonicity constraint) if the rule has at least one pair of semantically correlated criteria spanned over the condition and decision part. We say that one profile *dominates* another if they both involve the same values of regular attributes and the values of criteria of the first profile are not worse than the values of criteria of the second profile.

Let us explain the dominance principle with respect to decision rules on the pupils’ example. Suppose that two rules induced from the pupils’ data set relate *Math* and *Ph* on the condition side, with *GA* on the decision side:

- Rule #1: if *Math* = *medium* and *Ph* = *medium*, then *GA* = *good*,
- Rule #2: if *Math* = *good* and *Ph* = *medium*, then *GA* = *medium*.

The two rules do not observe the dominance principle because the condition profile of rule #2 dominates the condition profile of rule #1, while the decision profile of rule #2 is dominated by the decision profile of rule #1. Thus, in the sense of the dominance principle, the two rules are inconsistent, i.e. they are wrong.

One could say that the above rules are true because they are supported by examples of pupils from the learning sample, but this would mean that the examples are also inconsistent. The *inconsistency* may come from many sources. Examples include:

- Missing attributes (regular ones or criteria) in the description of objects. Maybe the data set does not include such attributes as the *opinion of the pupil’s tutor* expressed only verbally during an assessment of the pupil’s *GA* by a school assessment committee.
- Unstable preferences of decision makers. Maybe the members of the school assessment committee changed their view on the influence of *Math* on *GA* during the assessment.

Handling these inconsistencies is of crucial importance for knowledge discovery. They cannot be simply considered as noise or error to be eliminated from data, or amalgamated with consistent data by some averaging operators. They should be identified and presented as uncertain patterns.

If item (iii) was ignored in prior knowledge, then the handling of the above-mentioned inconsistencies would be impossible. Indeed, there would be nothing wrong with rules #1 and #2. They would be supported by different examples discerned by considered attributes.

It has been acknowledged by many authors that rough set theory provides an excellent framework for dealing with inconsistencies in knowledge discovery (see, for example, Grzymala-Busse 1992; Pawlak 1991; Pawlak et al. 1995; Polkowski 2002; Polkowski and Skowron 1999; Słowiński 1992b; Słowiński and Zopounidis 1995; Ziarko 1998). As we have shown in Sect. 19.2, the paradigm of rough set theory is

that of *granular computing*, because the main concept of the theory (rough approximation of a set) is built up of blocks of objects which are indiscernible by a given set of attributes, called *granules of knowledge*. In the space of regular attributes, the indiscernibility granules are bounded sets. Decision rules induced from rough approximation of a classification are also built up of such granules. While taking into account prior knowledge of type (i) and (ii), the rough approximation and the inherent rule induction ignore, however, prior knowledge of type (iii). In consequence, the resulting decision rules may be inconsistent with the dominance principle.

The authors have proposed an extension of the granular computing paradigm that enables us to take into account prior knowledge of type (iii), in addition to either (i) only (Greco et al. 2002a), or (i) and (ii) together (Greco et al. 1998a, 1999b, 2000d, 2001a, 2002b,c; Slowinski et al. 2002a, 2009). The combination of the new granules with the idea of rough approximation is called the Dominance-Based Rough Set Approach.

In the following, we present the concept of granules which permit us to handle prior knowledge of type (iii) when inducing decision rules.

Let U be a finite set of objects (universe) and let Q be a finite set of attributes divided into a set C of *condition attributes* and a set D of *decision attributes* where $C \cap D = \emptyset$. Also, let

$$X_C = \prod_{q=1}^{|C|} X_q \quad \text{and} \quad X_D = \prod_{q=1}^{|D|} X_q$$

be attribute spaces corresponding to sets of condition and decision attributes, respectively. The elements of X_C and X_D can be interpreted as possible evaluations of objects on attributes from set $C = \{1, \dots, |C|\}$ and from set $D = \{1, \dots, |D|\}$, respectively. Therefore, X_q is the set of possible evaluations of considered objects with respect to attribute q . The value of object x on attribute $q \in Q$ is denoted by x_q . Objects x and y are *indiscernible* by $P \subseteq C$ if $x_q = y_q$ for all $q \in P$ and, analogously, objects x and y are indiscernible by $R \subseteq D$ if $x_q = y_q$ for all $q \in R$. The sets of indiscernible objects are equivalence classes of the corresponding *indiscernibility relation* I_P or I_R . Moreover, $I_P(x)$ and $I_R(x)$ denote equivalence classes including object x . I_D generates a partition of U into a finite number of decision classes $Cl = \{Cl_t, t = 1, \dots, n\}$. Each $x \in U$ belongs to one and only one class $Cl_t \in Cl$.

The above definitions take into account prior knowledge of type (i) and (ii) only. In this case, the granules of knowledge are bounded sets in X_P and X_R ($P \subseteq C$ and $R \subseteq D$), defined by partitions of U induced by the indiscernibility relations I_P and I_R , respectively. Then, classification patterns to be discovered are functions representing granules $I_R(x)$ by granules $I_P(x)$ in the condition attribute space X_P , for any $P \subseteq C$ and for any $x \in U$.

If prior knowledge includes item (iii) in addition to (i) and (ii), then the indiscernibility relation is unable to produce granules in X_C and X_D that would take into account the preference order. To do so, the indiscernibility relation has to be substituted by a dominance relation in X_P and X_R ($P \subseteq C$ and $R \subseteq D$). Suppose, for simplicity, that all condition attributes in C and all decision attributes in D are criteria, and that C and D are semantically correlated.

Let \succeq_q be a *weak preference relation* on U (often called *outranking*) representing a preference on the set of objects with respect to criterion $q \in \{C \cup D\}$. Now, $x_q \succeq_q y_q$ means “ x_q is at least as good as y_q with respect to criterion q ”. On the one hand, we say that x *dominates* y with respect to $P \subseteq C$ (in brief, $x P\text{-dominates } y$) in the condition attribute space X_P (denoted by xD_{Py}) if $x_q \succeq_q y_q$ for all $q \in P$. Assuming, without loss of generality, that the domains of the criteria are numerical (i.e. $X_q \subseteq \mathbb{R}$ for any $q \in C$) and that they are ordered so that the preference increases with the value, we can say that xD_{Py} is equivalent to $x_q \geq y_q$ for all $q \in P, P \subseteq C$. Observe that for each $x \in X_P$, xD_{Px} , i.e. P -dominance is reflexive. On the other hand, the analogous definition holds in the decision attribute space X_R (denoted by xD_{Ry}), where $R \subseteq D$.

The dominance relations xD_{Py} and xD_{Ry} ($P \subseteq C$ and $R \subseteq D$) are directional statements, where x is a subject and y is a referent.

If $x \in X_P$ is the referent, then one can define a set of objects $y \in X_P$ dominating x , called the *P-dominating set* (denoted by $D_P^+(x)$) and defined as

$$D_P^+(x) = \{y \in U : yD_{Px}\}.$$

If $x \in X_P$ is the subject, then one can define a set of objects $y \in X_P$ dominated by x , called the *P-dominated set* (denoted by $D_P^-(x)$) and defined as

$$D_P^-(x) = \{y \in U : xD_{Py}\}.$$

P-dominating sets $D_P^+(x)$ and *P*-dominated sets $D_P^-(x)$ correspond to *positive* and *negative dominance cones* in X_P , with the origin x .

With respect to the decision attribute space X_R (where $R \subseteq D$), the *R*-dominance relation enables us to define the following sets:

$$Cl_R^{\geq x} = \{y \in U : yD_{Rx}\}, \quad Cl_R^{\leq x} = \{y \in U : xD_{Ry}\}.$$

We denote by $Cl_{t_q} = \{x \in X_D : x_q = t = q\}$ the decision class corresponding to $q \in D$. $Cl_R^{\geq x}$ is called the *upward union* of classes and $Cl_R^{\leq x}$ is the *downward union* of classes. If $x \in Cl_R^{\geq x}$, then x belongs to class $Cl_{t_q}, x_q = t_q$, or better, on each decision attribute $q \in R$. On the other hand, if $x \in Cl_R^{\leq x}$, then x belongs to class $Cl_{t_q}, x_q = t_q$, or worse, on each decision attribute $q \in R$. The downward and upward unions of classes correspond to the *positive* and *negative dominance cones* in X_R , respectively.

In this case, the granules of knowledge are open sets in X_P and X_R defined by dominance cones $D_P^+(x), D_P^-(x)$ ($P \subseteq C$) and $Cl_R^{\geq x}, Cl_R^{\leq x}$ ($R \subseteq D$), respectively. Then, classification patterns to be discovered are functions representing granules $Cl_R^{\geq x}, Cl_R^{\leq x}$ by granules $D_P^+(x), D_P^-(x)$, respectively, in the condition attribute space X_P , for any $P \subseteq C$ and $R \subseteq D$ and for any $x \in X_P$.

In both cases above, the functions are sets of decision rules.

19.4 The Dominance-Based Rough Set Approach

19.4.1 Granular Computing with Dominance Cones

When discovering classification patterns, a set D of decision attributes is, usually, a singleton, $D = \{d\}$. Let us take this assumption for further presentation, although it is not necessary for the DRSA. The decision attribute d makes a partition of U into a finite number of classes, $\mathbf{Cl} = \{Cl_t, t = 1, \dots, n\}$. Each $x \in U$ belongs to one and only one class, $Cl_t \in \mathbf{Cl}$. The upward and downward unions of classes boil down, respectively, to

$$Cl_t^{\geq} = \bigcup_{s \geq t} Cl_s$$

$$Cl_t^{\leq} = \bigcup_{s \leq t} Cl_s$$

where $t = 1, \dots, n$. Notice that for $t = 2, \dots, n$ we have $Cl_n^{\leq} = U - Cl_{t-1}^{\leq}$, i.e. all the objects not belonging to class Cl_t or better, belong to class Cl_{t-1} or worse.

Let us explain how the rough set concept has been generalized to the DRSA in order to enable granular computing with dominance cones (for more details, see [Greco et al. 1998a, 1999b, 2000d, 2001a, 2002b; Slowinski et al. 2009, 2000](#)).

Given a set of criteria, $P \subseteq C$, the inclusion of an object $x \in U$ to the upward union of classes Cl_t^{\geq} , $t = 2, \dots, n$, is *inconsistent with the dominance principle* if one of the following conditions holds:

- x belongs to class Cl_t or better but it is P -dominated by an object y belonging to a class worse than Cl_t , i.e. $x \in Cl_t^{\geq}$ but $D_P^+(x) \cap Cl_{t-1}^{\leq} \neq \emptyset$,
- x belongs to a worse class than Cl_t but it P -dominates an object y belonging to class Cl_t or better, i.e. $x \notin Cl_t^{\geq}$ but $D_P^+(x) \cap Cl_t^{\geq} \neq \emptyset$.

If, given a set of criteria $P \subseteq C$, the inclusion of $x \in U$ to Cl_t^{\geq} , where $t = 2, \dots, n$, is inconsistent with the dominance principle, we say that x belongs to Cl_t^{\geq} with some ambiguity. Thus, x belongs to Cl_t^{\geq} without any ambiguity with respect to $P \subseteq C$, if $x \in Cl_t^{\geq}$ and there is no inconsistency with the dominance principle. This means that all objects P -dominating x belong to Cl_t^{\geq} , i.e. $D_P^+(x) \subseteq Cl_t^{\geq}$. Geometrically, this corresponds to the inclusion of the complete set of objects contained in the positive dominance cone originating in x , in the positive dominance cone Cl_t^{\geq} originating in Cl_t .

Furthermore, x possibly belongs to Cl_t^{\geq} with respect to $P \subseteq C$ if one of the following conditions holds:

- According to decision attribute d , x belongs to Cl_t^{\geq} ,
- According to decision attribute d , x does not belong to Cl_t^{\geq} , but it is inconsistent in the sense of the dominance principle with an object y belonging to Cl_t^{\geq} .

In terms of ambiguity, x possibly belongs to Cl_t^{\geq} with respect to $P \subseteq C$, if x belongs to Cl_t^{\geq} with or without any ambiguity. Due to the reflexivity of the P -dominance relation D_P , the above conditions can be summarized as follows:

x possibly belongs to class Cl_t or better, with respect to $P \subseteq C$, if among the objects P -dominated by x there is an object y belonging to class Cl_t or better, i.e. $D_P^+(x) \cap Cl_t^{\geq} \neq \emptyset$. Geometrically, this corresponds to the non-empty intersection of the set of objects contained in the negative dominance cone originating in x , with the positive dominance cone Cl_t^{\geq} originating in Cl_t .

For $P \subseteq C$, the set of all objects belonging to Cl_t^{\geq} without any ambiguity constitutes the P -lower approximation of Cl_t^{\geq} , denoted by $\underline{P}Cl_t^{\geq}$, and the set of all objects that possibly belong to Cl_t^{\geq} constitutes the P -upper approximation of Cl_t^{\geq} , denoted by $\overline{P}(Cl_t^{\geq})$. More formally, we can say that

$$\underline{P}(Cl_t^{\geq}) = \{x \in U : D_P^+(x) \subseteq Cl_t^{\geq}\},$$

$$\overline{P}(Cl_t^{\geq}) = \{x \in U : D_P^-(x) \cap Cl_t^{\geq} \neq \emptyset\},$$

where $t = 1, \dots, n$. Analogously, one can define the P -lower approximation and the P -upper approximation of Cl_t^{\leq} as follows:

$$\underline{P}(Cl_t^{\leq}) = \{x \in U : D_P^-(x) \subseteq Cl_t^{\leq}\},$$

$$\overline{P}(Cl_t^{\leq}) = \{x \in U : D_{P(x)}^+ \cap Cl_t^{\leq} \neq \emptyset\},$$

where $t = 1, \dots, n$. The P -lower and P -upper approximations so defined satisfy the following *inclusion properties* for each $t \in \{1, \dots, n\}$ and for all $P \subseteq C$:

$$\underline{P}(Cl_t^{\geq}) \subseteq Cl_t^{\geq} \subseteq \overline{P}(Cl_t^{\geq}),$$

$$\underline{P}(Cl_t^{\leq}) \subseteq Cl_t^{\leq} \subseteq \overline{P}(Cl_t^{\leq}).$$

All the objects belonging to Cl_t^{\geq} and Cl_t^{\leq} with some ambiguity constitute the P -boundary of Cl_t^{\geq} and Cl_t^{\leq} , denoted by $Bn_P(Cl_t^{\geq})$ and $Bn_P(Cl_t^{\leq})$, respectively. They can be represented, in terms of upper and lower approximations, as follows:

$$Bn_P(Cl_t^{\geq}) = \overline{P}(Cl_t^{\geq}) - \underline{P}(Cl_t^{\geq}),$$

$$Bn_P(Cl_t^{\leq}) = \overline{P}(Cl_t^{\leq}) - \underline{P}(Cl_t^{\leq}),$$

where $t = 1, \dots, n$. The P -lower and P -upper approximations of the unions of classes Cl_t^{\geq} and Cl_t^{\leq} have an important *complementarity property*. It says that if object x belongs without any ambiguity to class Cl_t or better, then it is impossible that it could belong to class Cl_{t-1} or worse, i.e. $\underline{P}(Cl_t^{\geq}) = U - \overline{P}(Cl_{t-1}^{\leq}), t = 2, \dots, n$.

Due to the complementarity property, $Bn_P(Cl_t^{\geq}) = Bn_P(Cl_{t-1}^{\leq})$, for $t = 2, \dots, n$, which means that if x belongs with ambiguity to class Cl_t or better, then it also belongs with ambiguity to class Cl_{t-1} or worse.

From the knowledge discovery point of view, P -lower approximations of unions of classes represent *certain knowledge* provided by criteria from $P \subseteq C$, while P -upper approximations represent *possible knowledge* and the P -boundaries contain *doubtful knowledge* provided by the criteria from $P \subseteq C$.

The above definitions of rough approximations are based on a strict application of the dominance principle. However, when defining non-ambiguous objects, it is reasonable to accept a limited proportion of negative examples, particularly for large data tables. This extended version of the DRSA is called the Variable Consistency DRSA (VC-DRSA) model (Greco et al. 2001f).

For any $P \subseteq C$, we say that $x \in U$ belongs to Cl_t^{\geq} with no ambiguity at consistency level $l \in (0, 1]$, if $x \in Cl_t^{\geq}$ and at least $l * 100\%$ of all objects $y \in U$ dominating x with respect to P also belong to Cl_t^{\geq} , i.e.

$$\frac{|D_P^+(x) \cap Cl_t^{\geq}|}{|D_P^+(x)|} \geq l.$$

The term $|D_P^+(x) \cap Cl_t^{\geq}| / |D_P^+(x)|$ is called *rough membership* and can be interpreted as conditional probability $\Pr(y \in Cl_t^{\geq} | y \in D_P^+(x))$. The level l is called the *consistency level* because it controls the degree of consistency between objects qualified as belonging to Cl_t^{\geq} without any ambiguity. In other words, if $l < 1$, then at most $(1 - l) * 100\%$ of all objects $y \in U$ dominating x with respect to P do not belong to Cl_t^{\geq} and thus contradict the inclusion of x in Cl_t^{\geq} .

Analogously, for any $P \subseteq C$ we say that $x \in U$ belongs to Cl_t^{\leq} with no ambiguity at consistency level $l \in (0, 1]$, if $x \in Cl_t^{\leq}$ and at least $l * 100\%$ of all the objects $y \in U$ dominated by x with respect to P also belong to Cl_t^{\leq} , i.e.

$$\frac{|D_P^-(x) \cap Cl_t^{\leq}|}{|D_P^-(x)|} \geq l.$$

The rough membership $|D_P^-(x) \cap Cl_t^{\leq}| / |D_P^-(x)|$ can be interpreted as conditional probability $\Pr(y \in Cl_t^{\leq} | y \in D_P^-(x))$. Thus, for any $P \subseteq C$, each object $x \in U$ is either ambiguous or non-ambiguous at consistency level l with respect to the upward union Cl_t^{\geq} ($t = 2, \dots, n$) or with respect to the downward union Cl_t^{\leq} ($t = 1, \dots, n - 1$).

The concept of non-ambiguous objects at some consistency level l leads naturally to the definition of P -lower approximations of the unions of classes Cl_t^{\geq} and Cl_t^{\leq} which can be formally presented as follows:

$$\begin{aligned} P^l(Cl_t^{\geq}) &= \left\{ x \in Cl_t^{\geq} : \frac{|D_P^+(x) \cap Cl_t^{\geq}|}{|D_P^+(x)|} \geq l \right\}, \\ P^l(Cl_t^{\leq}) &= \left\{ x \in Cl_t^{\leq} : \frac{|D_P^-(x) \cap Cl_t^{\leq}|}{|D_P^-(x)|} \geq l \right\}. \end{aligned}$$

Given $P \subseteq C$ and consistency level l , we can define the P -upper approximations of Cl_t^{\geq} and Cl_t^{\leq} , denoted by $\bar{P}^l(Cl_t^{\geq})$ and $\bar{P}^l(Cl_t^{\leq})$, respectively, by complementation of $\underline{P}^l(Cl_{t-1}^{\leq})$ and $\underline{P}^l(Cl_{t+1}^{\geq})$ with respect to U as follows:

$$\bar{P}^l(Cl_t^{\geq}) = U - \underline{P}^l(Cl_{t-1}^{\leq}),$$

$$\bar{P}^l(Cl_t^{\leq}) = U - \underline{P}^l(Cl_{t+1}^{\geq}).$$

$\bar{P}^l(Cl_t^{\geq})$ can be interpreted as the set of all the objects belonging to Cl_t^{\geq} , which are *possibly ambiguous* at consistency level l . Analogously, $\bar{P}^l(Cl_t^{\leq})$ can be interpreted as the set of all the objects belonging to Cl_t^{\leq} , which are *possibly ambiguous* at consistency level l . The P -boundaries (P -doubtful regions) of Cl_t^{\geq} and Cl_t^{\leq} are defined as

$$Bn_P(Cl_t^{\geq}) = \bar{P}^l(Cl_t^{\geq}) - \underline{P}^l(Cl_t^{\geq}),$$

$$Bn_P(Cl_t^{\leq}) = \bar{P}^l(Cl_t^{\leq}) - \underline{P}^l(Cl_t^{\leq}),$$

where $t = 1, \dots, n$. The VC-DRSA model provides some degree of flexibility in assigning objects to lower and upper approximations of the unions of decision classes. It can easily be demonstrated that for $0 < l' < l \leq 1$ and $t = 2, \dots, n$,

$$\underline{P}^l(Cl_t^{\geq}) \subseteq \underline{P}^{l'}(Cl_t^{\geq}) \quad \text{and} \quad \bar{P}^{l'}(Cl_t^{\geq}) \subseteq \bar{P}^l(Cl_t^{\geq}).$$

The VC-DRSA model is inspired by Ziarko's model of the *variable-precision* rough set approach (Ziarko 1993, 1998). However, there is a significant difference in the definition of rough approximations because $\underline{P}^l(Cl_t^{\geq})$ and $\bar{P}^l(Cl_t^{\geq})$ are composed of non-ambiguous and ambiguous *objects* at the consistency level l , respectively, while Ziarko's $\underline{P}^l(Cl_t)$ and $\bar{P}^l(Cl_t)$ are composed of P -indiscernibility sets such that at least $l*100\%$ of these sets are included in Cl_t or have an non-empty intersection with Cl_t , respectively. If one would like to use Ziarko's definition of variable-precision rough approximations in the context of multiple-criteria classification, then the P -indiscernibility sets should be substituted by P -dominating sets $D_P^+(x)$. However, then the notion of ambiguity that naturally leads to the general definition of rough approximations (see Słowiński and Vanderpooten 2000) loses its meaning. Moreover, a bad side effect of the direct use of Ziarko's definition is that a lower approximation $\underline{P}^l(Cl_t^{\geq})$ may include objects y assigned to Cl_h , where h is much less than t , if y belongs to $D_P^+(x)$, which was included in $\underline{P}^l(Cl_t^{\geq})$. When the decision classes are preference ordered, it is reasonable to expect that objects assigned to far worse classes than the considered union are not counted to the lower approximation of this union.

The VC-DRSA model presented above has been generalized by Greco et al. (2008b) and Blaszcynski et al. (2009). The generalized model applies two types of consistency generalizing the concept of rough membership in the definition of lower approximations:

- Gain-type consistency measures $f_{\geq t}^P(x), f_{\leq t}^P(x)$:

$$\underline{P}_{\alpha \geq t}^{\alpha} (Cl_t^{\geq}) = \left\{ x \in Cl_t^{\geq} : f_{\geq t}^P(x) \geq \alpha_{\geq t} \right\},$$

$$\underline{P}_{\alpha \leq t}^{\alpha} (Cl_t^{\leq}) = \left\{ x \in Cl_t^{\leq} : f_{\leq t}^P(x) \geq \alpha_{\leq t} \right\}$$

- Cost-type consistency measures $g_{\geq t}^P(x), g_{\leq t}^P(x)$:

$$\underline{P}_{\beta \geq t}^{\beta} (Cl_t^{\geq}) = \left\{ x \in Cl_t^{\geq} : g_{\geq t}^P(x) \leq \beta_{\geq t} \right\},$$

$$\underline{P}_{\beta \leq t}^{\beta} (Cl_t^{\leq}) = \left\{ x \in Cl_t^{\leq} : g_{\leq t}^P(x) \leq \beta_{\leq t} \right\}$$

where $\alpha_{\geq t}, \alpha_{\leq t}, \beta_{\geq t}, \beta_{\leq t}$ are threshold values on the consistency measures which are conditioning the inclusion of object x in the P -lower approximation of Cl_t^{\geq} , or Cl_t^{\leq} . To be concordant with the rough set philosophy, consistency measures should enjoy some monotonicity properties (see Table 19.3). A consistency measure is monotonic if it does not decrease (or does not increase) when

- (m1) the set of attributes is growing,
- (m2) the set of objects is growing,
- (m3) the union of ordered classes is growing,
- (m4) x improves its evaluation, so that it dominates more objects.

The ε -consistency measures which enjoy at least three from among four monotonicity properties are defined as follows:

$$\varepsilon_{\geq t}^P(x) = \frac{|D_P^+ \cap \neg Cl_t^{\geq}|}{|\neg Cl_t^{\geq}|}, \quad \varepsilon_{\leq t}^P(x) = \frac{|D_P^- \cap \neg Cl_t^{\leq}|}{|\neg Cl_t^{\leq}|}.$$

They can be interpreted as estimates of conditional probability, respectively:

$$\Pr(y \in D_P^+(x) | y \in \neg Cl_t^{\geq}), \quad \Pr(y \in D_P^-(x) | y \in \neg Cl_t^{\leq}).$$

They say how far the implications $y \in D_P^+(x) \Rightarrow y \in Cl_t^{\geq}, y \in D_P^-(x) \Rightarrow y \in Cl_t^{\leq}$ are *not* supported by the data.

For every $P \subseteq C$, the objects are consistent in the sense of the dominance principle with all upward and downward unions of classes being *P-correctly classified*. For every $P \subseteq C$, the *quality of approximation of classification CI* by the set of criteria P is defined as the ratio between the number of P -correctly classified objects and the number of all the objects in the data sample set. Since the objects which are

Table 19.3 Monotonicity properties of consistency measures (Blaszczyński et al. 2009)

Consistency measure	(m1)	(m2)	(m3)	(m4)
μ (rough membership)	No	Yes	Yes	No
μ'	No	Yes	Yes	Yes
B (Bayesian)	No	No	No	No
β	No	Yes	Yes	Yes
ϵ	Yes	Yes	No	Yes
ϵ^*	Yes	Yes	Yes	Yes
ϵ'	Yes	Yes	Yes	Yes
$\bar{\mu}$	Yes	Yes	Yes	Yes

P -correctly classified are those that do not belong to any P -boundary of unions $Cl_t^>$ and $Cl_t^<$, $t = 1, \dots, n$, the quality of approximation of classification Cl by set of criteria P can be written as

$$\gamma_P(Cl) = \frac{|(U - (\bigcup_{t \in \{1, \dots, n\}} Bn_P(Cl_t^<)) \cup (\bigcup_{t \in \{1, \dots, n\}} Bn_P(Cl_t^>)))|}{|U|} \\ = \frac{|(U - (\bigcup_{t \in \{1, \dots, n\}} Bn_P(Cl_t^>)))|}{|U|}.$$

$\gamma_P(Cl)$ can be seen as a measure of the quality of knowledge that can be extracted from the data table, where P is the set of criteria and Cl is the considered classification.

Each minimal subset $P \subseteq C$ such that $\gamma_P(Cl) = \gamma_C(Cl)$ is called a *reduct* of Cl and is denoted by RED_{Cl} . Note that a decision table can have more than one reduct. The intersection of all reducts is called the *core* and is denoted by $CORE_{Cl}$. Criteria from $CORE_{Cl}$ cannot be removed from the data sample set without deteriorating the knowledge to be discovered. This means that in set C there are three categories of criteria:

- *Indispensable* criteria included in the core,
- *Exchangeable* criteria included in some reducts but not in the core,
- *Redundant* criteria being neither indispensable nor exchangeable, thus not included in any reduct.

Note that reducts are minimal subsets (with respect to inclusion) of attributes and criteria conveying the relevant knowledge contained in the learning sample. This knowledge is relevant for the explanation of patterns in a given decision table, but not necessarily for prediction.

It has been shown in Greco et al. (2001d) that the quality of classification satisfies properties of set functions which are called *fuzzy measures*. For this reason, we can use the quality of classification for the calculation of indices which measure the relevance of particular attributes and/or criteria, in addition to the strength of interactions between them. The useful indices are: the value index and interaction indices of Shapley and Banzhaf; the interaction indices of Murofushi-Sonedda and

Roubens; and the Möbius representation. All these indices can help to assess the interaction between the considered attributes and criteria, and can help to choose the best reduct.

19.4.2 Stochastic DRSA

From a probabilistic point of view, the assignment of object x_i to “at least” class t can be made with probability $\Pr(y_i \geq t|x_i)$, where y_i is classification decision for x_i , $t = 1, \dots, n$. This probability is supposed to satisfy the usual axioms of probability: $\Pr(y_i \geq t|x_i) = 1$, $\Pr(y_i \leq t|x_i) = 1 - \Pr(y_i \geq t+1|x_i)$, and $\Pr(y_i > t|x_i) < \Pr(y_i > t'|x_i)$ for $t > t'$. These probabilities are unknown but can be estimated from data.

For each class $t = 2, \dots, n$, we have a binary problem of estimating the conditional probabilities $\Pr(y_i \geq t|x_i)$, $\Pr(y_i < t|x_i)$. It can be solved by *isotonic regression* (Kotlowski et al. 2008). Let $y_{it} = 1$ if $y_i \geq t$, otherwise $y_{it} = 0$. Let also p_{it} be the estimate of the probability $\Pr(y_i \geq t|x_i)$. Then, choose estimates p_{it}^* which minimize the squared distance to the class assignment y_{it} , subject to the monotonicity constraints

$$\begin{aligned} \text{Minimize : } & \sum_{i=1}^{|U|} (y_{it} - p_{it})^2 \\ \text{subject to } & x_i \succeq x_j \rightarrow p_{it} \geq p_{jt} \quad \text{for all } x_i, x_j \in U, \end{aligned}$$

where $x_i \succeq x_j$ means that x_i dominates x_j . Then, stochastic α -lower approximations for classes “at least t ” and “at most $t-1$ ” can be defined as

$$\begin{aligned} \underline{P}^\alpha(Cl_t^{\geq}) &= \{x_i \in U : \Pr(y_i \geq t|x_i) \geq \alpha\} \\ \underline{P}^\alpha(Cl_{t-1}^{\leq}) &= \{x_i \in U : \Pr(y_i < t|x_i) \geq \alpha\}. \end{aligned}$$

Replacing the unknown probabilities $\Pr(y_i \geq t|x_i)$, $\Pr(y_i < t|x_i)$ by their estimates p_{it}^* obtained from isotonic regression, we get

$$\begin{aligned} \underline{P}^\alpha(Cl_t^{\geq}) &= \{x_i \in U : p_{it}^* \geq \alpha\} \\ \underline{P}^\alpha(Cl_{t-1}^{\leq}) &= \{x_i \in U : p_{it}^* \leq 1 - \alpha\} \end{aligned}$$

where parameter $\alpha \in [0.5, 1]$ controls the allowed amount of inconsistency.

Solving isotonic regression requires $O(|U|^4)$ time, but a good heuristic needs only $O(|U|^2)$.

In fact, as shown by Kotlowski et al. (2008), we don't really need to know the probability estimates to obtain stochastic lower approximations. We only need to know for which object x_i , $p_{it}^* \geq \alpha$ and for which x_i , $p_{it}^* \leq 1 - \alpha$. This can be found by solving a linear programming (reassignment) problem.

As before, $y_{it} = 1$ if $y_i \geq t$, otherwise $y_{it} = 0$. Let d_{it} be the decision variable which determines a new class assignment for object x_i . Then, reassign objects from union of classes indicated by y_{it} to union of classes indicated by d_{it}^* , such that the new class assignments are consistent with the dominance principle, where d_{it}^* results from solving the following linear programming problem:

$$\begin{aligned} \text{Minimize: } & \sum_{i=1}^{|U|} w_{y_{it}} |y_{it} - d_{it}| \\ \text{subject to: } & x_i \succeq x_j \rightarrow d_{it} \geq d_{jt} \quad \text{for all } x_i, x_j \in U. \end{aligned}$$

Due to unimodularity of the constraint matrix, the optimal solution of this linear programming problem is always integer, i.e. $d_{it}^* \in \{0, 1\}$. For all objects consistent with the dominance principle, $d_{it}^* = y_{it}$. If we set $w_0 = \alpha$ and $w_1 = \alpha - 1$, then the optimal solution d_{it}^* satisfies: $d_{it}^* = 1 \Leftrightarrow p_{it}^* \geq \alpha$. If we set $w_0 = 1 - \alpha$ and $w_1 = \alpha$, then the optimal solution d_{it}^* satisfies: $d_{it}^* = 0 \Leftrightarrow p_{it}^* \leq 1 - \alpha$.

For each $t = 2, \dots, n$, solving the reassignment problem twice, we can obtain the lower approximations $\underline{P}^\alpha(Cl_t^{\geq})$, $\underline{P}^\alpha(Cl_{t-1}^{\leq})$ without knowing the probability estimates!

19.4.3 Induction of Decision Rules

The dominance-based rough approximations of upward and downward unions of classes can serve to induce a generalized description of the objects contained in the decision table in terms of “if ..., then ...” decision rules. For a given upward or downward union of classes, Cl_t^{\geq} or Cl_s^{\leq} , the decision rules induced under a hypothesis that objects belonging to $\underline{P}(Cl_t^{\geq})$ or $\underline{P}(Cl_s^{\leq})$ are positive and all the others are negative suggest an assignment to “class Cl_t or better”, or to “class Cl_s or worse”, respectively. On the other hand, the decision rules induced under a hypothesis that objects belonging to the intersection $\overline{P}(Cl_s^{\leq}) \cap \overline{P}(Cl_t^{\geq})$ are positive and all the others are negative suggest an assignment to some classes between Cl_s and Cl_t ($s < t$).

In the case of preference ordered data it is meaningful to consider the following five types of decision rules:

1. *Certain D \geq -decision rules.* These provide lower profile descriptions for objects belonging to Cl_t^{\geq} without ambiguity: if $x_{q1} \succeq_{q1} r_{q1}$ and $x_{q2} \succeq_{q2} r_{q2}$ and ... $x_{qp} \succeq_{qp} r_{qp}$, then $x \in Cl_t^{\geq}$, where for each $w_q, z_q \in X_q$, “ $w_q \succeq_q z_q$ ” means “ w_q is at least as good as z_q ”.
2. *Possible D \geq -decision rules.* Such rules provide lower profile descriptions for objects belonging to Cl_t^{\geq} with or without any ambiguity: if $x_{q1} \succeq_{q1} r_{q1}$ and $x_{q2} \succeq_{q2} r_{q2}$ and ... $x_{qp} \succeq_{qp} r_{qp}$, then x possibly belongs to Cl_t^{\geq} .
3. *Certain D \leq -decision rules.* These give upper profile descriptions for objects belonging to Cl_t^{\leq} without ambiguity: if $x_{q1} \preceq_{q1} r_{q1}$ and $x_{q2} \preceq_{q2} r_{q2}$ and ... $x_{qp} \preceq_{qp} r_{qp}$, then $x \in Cl_t^{\leq}$, where for each $w_q, z_q \in X_q$, “ $w_q \preceq_q z_q$ ” means “ w_q is at most as good as z_q ”.

4. *Possible D_≤-decision rules.* These provide upper profile descriptions for objects belonging to $Cl_t^≤$ with or without any ambiguity: if $x_{q1} \preceq_{q1} r_{q1}$ and $x_{q2} \preceq_{q2} r_{q2}$ and $x_{qp} \preceq_{qp} r_{qp}$, then x possibly belongs to $Cl_t^≤$.
5. *Approximate D_{≥≤}-decision rules.* These represent simultaneously lower and upper profile descriptions for objects belonging to $Cl_s \cup Cl_{s+1} \cup \dots \cup Cl_t$ without the possibility of discerning the actual class: if $x_{q1} \succeq_{q1} r_{q1}$ and ... $x_{qk} \succeq_{qk} r_{qk}$ and $x_{qk+1} \preceq_{qk+1} r_{qk+1}$ and ... $x_{qp} \preceq_{qp} r_{qp}$, then $x \in Cl_s \cup Cl_{s+1} \cup \dots \cup Cl_t$.

In the left-hand side of a $D_{≥≤}$ -decision rule we can have $x_q \succeq_q r_q$ and $x_q \preceq_q r'_q$, where $r_q \leq r'_q$, for the same $q \in C$. Moreover, if $r_q = r'_q$, the two conditions boil down to $x_q \sim_q r_q$, where for each $w_q, z_q \in X_q$, then $w_q \sim_q z_q$ means “ w_q is indifferent to z_q ”.

A *minimal* rule is an implication where we understand that there is no other implication with a left hand side which has at least the same weakness (which means that it uses a subset of elementary conditions and/or weaker elementary conditions) and which has a right hand side that has at least the same strength (which means, a $D_≥$ - or a $D_≤$ -decision rule assigning objects to the same union or sub-union of classes, or a $D_{≥≤}$ -decision rule assigning objects to the same or larger set of classes).

The rules of type (1) and (3) represent certain knowledge extracted from the data table, while the rules of type (2) and (4) represent possible knowledge. Rules of type (5) represent doubtful knowledge.

The rules of type (1) and (3) are *exact* if they do not cover negative examples; they are *probabilistic*, otherwise. In the latter case, each rule is characterized by a *confidence* ratio, representing the probability that an object matching the left-hand side of the rule also matches its right-hand side. Probabilistic rules concord to the VC-DRSA model mentioned above.

We will now comment upon the application of decision rules to some objects described by criteria from C . When applying $D_≥$ -decision rules to an object x , it is possible that x either matches the left hand side of at least one decision rule or it does not. In the case of at least one such match, it is reasonable to conclude that x belongs to the lowest class resulting from intersection of all the right-hand sides of the rules covering x . More precisely, if x matches the left-hand side of rules p_1, p_2, \dots, p_m , having right-hand sides $x \in Cl_{t1}^≥, x \in Cl_{t2}^≥, \dots, x \in Cl_{tm}^≥$, then x is assigned to class Cl_t , where $t = \max\{t1, t2, \dots, tm\}$. In the case of no matching, we can conclude that x belongs to Cl_1 , i.e. to the worst class, since no rule with a right-hand side suggesting a better classification of x is covering this object.

Analogously, when applying $D_≤$ -decision rules to the object x , we can conclude that x belongs either to the highest class resulting from the intersection of all the right-hand sides of the rules covering x or to class Cl_n , i.e. to the best class, when x is not covered by any rule. More precisely, if x matches the left-hand side of rules p_1, p_2, \dots, p_m , having right-hand sides $x \in Cl_{t1}^≤, x \in Cl_{t2}^≤, \dots, x \in Cl_{tm}^≤$, then x is assigned to class Cl_t , where $t = \min\{t1, t2, \dots, tm\}$. In the case of no matching, it is concluded that x belongs to the best class Cl_n because no rule with a right hand side suggesting a worse classification of x is covering this object.

Finally, when applying $D_{\geq\leq}$ -decision rules to x , it is possible to conclude that x belongs to the union of all the classes suggested in the right-hand side of the rules covering x .

A new classification scheme has been proposed by [Blaszczyński et al. \(2007\)](#). Let $\phi_1 \rightarrow \psi_1, \dots, \phi_k \rightarrow \psi_k$, be the rules matching object x . Then, $R_t(x) = \{j : Cl_t \in \psi_j, j = 1, \dots, k\}$ denotes the set of rules matching x , which are recommending assignment of object x to a union including class Cl_t , and $R_{\neg t}(x) = \{j : Cl_t \notin \psi_j, j = 1, \dots, k\}$ denotes the set of rules matching x , which are not recommending assignment of object x to a union including class Cl_t . $\|\phi_j\|, \|\psi_j\|$ are sets of objects with property ϕ_j and ψ_j , respectively, $j = 1, \dots, k$. For a classified object x , one has to calculate the score for each candidate class:

$$score(Cl_t, x) = score^+(Cl_t, x) - score^-(Cl_t, x)$$

where

$$score^+(Cl_t, x) = \frac{\left| \bigcup_{j \in R_t(x)} (\|\phi_j\| \cap Cl_t) \right|^2}{\left| \bigcup_{j \in R_t(x)} \|\phi_j\| \right| \times |Cl_t|}$$

and

$$score^-(Cl_t, x) = \frac{\left| \bigcup_{j \in R_{\neg t}(x)} (\|\phi_j\| \cap \|\psi_j\|) \right|^2}{\left| \bigcup_{j \in R_{\neg t}(x)} \|\phi_j\| \right| \times \left| \bigcup_{j \in R_{\neg t}(x)} \|\psi_j\| \right|}.$$

$score^+(Cl_t, x)$ and $score^-(Cl_t, x)$ can be interpreted in terms of conditional probability as a product of confidence and coverage of the matching rules:

$$score^+(Cl_t, x) = \Pr(\{\phi_j : j \in R_t(x)\} | Cl_t) \times \Pr(Cl_t | \{\phi_j : j \in R_t(x)\})$$

$$score^-(Cl_t, x) = \Pr(\{\phi_j : j \in R_{\neg t}(x)\} | \neg Cl_t) \times \Pr(\neg Cl_t | \{\phi_j : j \in R_{\neg t}(x)\}).$$

The recommendation of the univocal classification $x \rightarrow Cl_t$ is such that

$$Cl_t = \arg \max_{t \in \{1, \dots, n\}} [score(Cl_t, x)].$$

A set of decision rules is *complete* if it is able to cover all objects from the decision table in such a way that consistent objects are re-classified to their original classes and inconsistent objects are classified to clusters of classes which refer to this inconsistency. Each set of decision rules that is complete and non-redundant is called *minimal*. Note that an exclusion of any rule from this set makes it non-complete.

In the case of the VC-DRSA, the decision rules are induced from the P -lower approximations whose composition is controlled by the user-specified consistency level l . Consequently, the value of confidence α for the rule should be constrained from the bottom. It is reasonable to require that the smallest accepted confidence level of the rule should not be lower than the currently used consistency level l . Indeed, in the worst case, some objects from the P -lower approximation may create a rule using all the criteria from P thus giving a confidence $\alpha \geq l$.

Observe that the syntax of decision rules induced from dominance-based rough approximations uses the concept of dominance cones: each condition profile is a dominance cone in X_C , and each decision profile is a dominance cone in X_D . In both cases the cone is positive for D_{\geq} -rules and negative for D_{\leq} -rules.

Also note that dominance cones which correspond to condition profiles can originate in any point of X_C , without the risk of being too specific. Thus, in contrast to granular computing based on an indiscernibility (or similarity) relation, in the case of granular computing based on dominance, the condition attribute space X_C need not be discretized (Greco et al. 2007, 2008a, 2009).

Some procedures for induction of rules from dominance-based rough approximations have been proposed by Greco et al. (2001g) and Blaszczyński et al. (2010b).

The utility of decision rules is threefold: they *explain* (summarize) decisions made on objects from the dataset; they can be used to *make decisions* with respect to new (unseen) objects which are matching conditions of some rules; and they make it possible to *build up a strategy of intervention* (Greco et al. 2005b). The attractiveness of particular decision rules can be measured in many different ways. However, the most convincing measures are Bayesian confirmation measures enjoying a special monotonicity property, as reported in Greco et al. (2004b).

In Giove et al. (2002), a new methodology for the induction of monotonic decision trees from dominance-based rough approximations of preference-ordered decision classes has been proposed.

It is finally worth noting that several algebraic models have been proposed for the DRSA (Greco et al. 2010a)—the algebraic structures are based on a bipolar disjoint representation (positive and negative) of the interior and exterior of a concept. These algebra models give elegant representations of basic properties of Dominance-Based Rough Sets. Moreover, a topology for the DRSA in a bitopological space has been proposed by Greco et al. (2010b).

19.4.4 An Illustrative Example

To illustrate the application of the DRSA to multicriteria classification, we will use a part of some data provided by a Greek industrial bank ETEVA which finances industrial and commercial firms in Greece (Slowinski and Zopounidis 1995). A sample composed of 39 firms has been chosen for the study in co-operation with the ETEVA's financial manager. The manager has classified the selected firms into three classes of bankruptcy risk. The sorting decision is represented by decision attribute d making a trichotomic partition of the 39 firms:

- $d = A$ means “acceptable”
- $d = U$ means “uncertain”
- $d = NA$ means “non-acceptable”.

The partition is denoted by $Cl = \{Cl_A, Cl_U, Cl_{NA}\}$ and, obviously, class Cl_A is better than Cl_U which is better than Cl_{NA} .

The firms were evaluated using the following 12 criteria (\uparrow means *preference increasing with value* and \downarrow means *preference decreasing with value*):

- A_1 = earnings before interests and taxes/total assets, \uparrow
- A_2 = net income/net worth, \uparrow
- A_3 = total liabilities/total assets, \downarrow
- A_4 = total liabilities/cash flow, \downarrow
- A_5 = interest expenses/sales, \downarrow
- A_6 = general and administrative expense/sales, \downarrow
- A_7 = managers' work experience, \uparrow (very low = 1, low = 2, medium = 3, high = 4, very high = 5)
- A_8 = firm's market niche/position, \uparrow (bad = 1, rather bad = 2, medium = 3, good = 4, very good = 5)
- A_9 = technical structure/facilities, \uparrow (bad = 1, rather bad = 2, medium = 3, good = 4, very good = 5)
- A_{10} = organization/personnel, \uparrow (bad = 1, rather bad = 2, medium = 3, good = 4, very good = 5)
- A_{11} = special competitive advantage of firms, \uparrow (low = 1, medium = 2, high = 3, very high = 4)
- A_{12} = market flexibility, \uparrow (very low = 1, low = 2, medium = 3, high = 4, very high = 5).

The first six criteria are cardinal (financial ratios) and the last six are ordinal. The data table is presented in Table 19.4.

The main questions to be answered by the knowledge discovery process were the following:

- Is the information contained in Table 19.4 consistent?
- What are the reducts of criteria ensuring the same quality of approximation of the multicriteria classification as the whole set of criteria?
- What decision rules can be extracted from Table 19.4?
- What are the minimal sets of decision rules?

We will answer these questions using the DRSA. The *first result* from this approach is a discovery that the financial data matrix is *consistent* for the complete set of criteria C . Therefore, the C -lower and C -upper approximations of $Cl_{NA}^{\leq}, Cl_U^{\leq}$ and Cl_A^{\geq}, Cl_U^{\geq} are the same. In other words, the quality of approximation of all upward and downward unions of classes, as well as the quality of classification, is equal to 1.

The *second discovery* is a set of 18 *reducts* of criteria ensuring the same quality of classification as the whole set of 12 criteria:

Table 19.4 Financial data table

Firm	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	d
F1	16.4	14.5	59.82	2.5	7.5	5.2	5	3	5	4	2	4	A
F2	35.8	67.0	64.92	1.7	2.1	4.5	5	4	5	5	4	5	A
F3	20.6	61.75	75.71	3.6	3.6	8.0	5	3	5	5	3	5	A
F4	11.5	17.1	57.1	3.8	4.2	3.7	5	2	5	4	3	4	A
F5	22.4	25.1	49.8	2.1	5.0	7.9	5	3	5	5	3	5	A
F6	23.9	34.5	48.9	1.7	2.5	8.0	5	3	4	4	3	4	A
F7	29.9	44.0	57.8	1.8	1.7	2.5	5	4	4	5	3	5	A
F8	8.7	5.4	27.4	3.3	4.5	4.5	5	2	4	4	1	4	A
F9	25.7	29.7	46.8	1.7	4.6	3.7	4	2	4	3	1	3	A
F10	21.2	24.6	64.8	3.7	3.6	8.0	4	2	4	4	1	4	A
F11	18.32	31.6	69.3	4.4	2.8	3.0	4	3	4	4	3	4	A
F12	20.7	19.3	19.7	0.7	2.2	4.0	4	2	4	4	1	3	A
F13	9.9	3.5	53.1	4.5	8.5	5.3	4	2	4	4	1	4	A
F14	10.4	9.3	80.9	9.4	1.4	4.1	4	2	4	4	3	3	A
F15	17.7	19.8	52.8	3.2	7.9	6.1	4	4	4	4	2	5	A
F16	14.8	15.9	27.94	1.3	5.4	1.8	4	2	4	3	2	3	A
F17	16.0	14.7	53.5	3.9	6.8	3.8	4	4	4	4	2	4	A
F18	11.7	10.01	42.1	3.9	12.2	4.3	5	2	4	2	1	3	A
F19	11.0	4.2	60.8	5.8	6.2	4.8	4	2	4	4	2	4	A
F20	15.5	8.5	56.2	6.5	5.5	1.8	4	2	4	4	2	4	A
F21	13.2	9.1	74.1	11.21	6.4	5.0	2	2	4	4	2	3	U
F22	9.1	4.1	44.8	4.2	3.3	10.4	3	4	4	4	3	4	U
F23	12.9	1.9	65.02	6.9	14.01	7.5	4	3	3	2	1	2	U
F24	5.9	-27.7	77.4	-32.2	16.6	12.7	3	2	4	4	2	3	U
F25	16.9	12.4	60.1	5.2	5.6	5.6	3	2	4	4	2	3	U
F26	16.7	13.1	73.5	7.1	11.9	4.1	2	2	4	4	2	3	U
F27	14.6	9.7	59.5	5.8	6.7	5.6	2	2	4	4	2	4	U
F28	5.1	4.9	28.9	4.3	2.5	46.0	2	2	3	3	1	2	U
F29	24.4	22.3	32.8	1.4	3.3	5.0	2	3	4	4	2	3	U
F30	29.7	8.6	41.8	1.6	5.2	6.4	2	3	4	4	2	3	U
F31	7.3	-64.5	67.5	-2.2	30.1	8.7	3	3	4	4	2	3	NA
F32	23.7	31.9	63.6	3.5	12.1	10.2	3	2	3	4	1	3	NA
F33	18.9	13.5	74.5	10.0	12.0	8.4	3	3	3	4	3	4	NA
F34	13.9	3.3	78.7	25.5	14.7	10.1	2	2	3	4	3	4	NA
F35	-13.3	-31.1	63.0	-10.0	21.2	23.1	2	1	4	3	1	2	NA
F36	6.2	-3.2	46.1	5.1	4.8	10.5	2	1	3	3	2	3	NA
F37	4.8	-3.3	71.9	34.6	8.6	11.6	2	2	4	4	2	3	NA
F38	0.1	-9.6	42.5	-20.0	12.9	12.4	1	1	4	3	1	3	NA
F39	13.6	9.1	76.0	11.4	17.1	10.3	1	1	2	1	1	2	NA

$$\begin{aligned}
RED_{CI}^1 &= \{A_1, A_4, A_5, A_7\}, RED_{CI}^2 = \{A_2, A_4, A_5, A_7\}, \\
RED_{CI}^3 &= \{A_3, A_4, A_6, A_7\}, RED_{CI}^4 = \{A_4, A_5, A_6, A_7\}, \\
RED_{CI}^5 &= \{A_4, A_5, A_7, A_8\}, RED_{CI}^6 = \{A_2, A_3, A_7, A_9\}, \\
RED_{CI}^7 &= \{A_1, A_3, A_4, A_7, A_9\}, RED_{CI}^8 = \{A_1, A_5, A_7, A_9\}, \\
RED_{CI}^9 &= \{A_2, A_5, A_7, A_9\}, RED_{CI}^{10} = \{A_4, A_5, A_7, A_9\}, \\
RED_{CI}^{11} &= \{A_5, A_6, A_7, A_9\}, RED_{CI}^{12} = \{A_4, A_5, A_7, A_{10}\}, \\
RED_{CI}^{13} &= \{A_1, A_3, A_4, A_7, A_{11}\}, RED_{CI}^{14} = \{A_2, A_3, A_4, A_7, A_{11}\}, \\
RED_{CI}^{15} &= \{A_4, A_5, A_6, A_{12}\}, RED_{CI}^{16} = \{A_1, A_3, A_5, A_6, A_9, A_{12}\}, \\
RED_{CI}^{17} &= \{A_3, A_4, A_6, A_{11}, A_{12}\}, RED_{CI}^{18} = \{A_1, A_2, A_3, A_6, A_9, A_{11}, A_{12}\}.
\end{aligned}$$

All the 18 subsets of criteria are equally good and sufficient for the perfect approximation of the classification performed by ETEVA's financial manager on the 39 firms. The core of CI is empty ($CORE_{CI} = \emptyset$) which means that no criterion is indispensable for the approximation. Moreover, all the criteria are exchangeable and no criterion is redundant.

The *third discovery* is the set of *all* decision rules. We obtained 74 rules describing Cl_{NA}^{\leq} , 51 rules describing Cl_U^{\leq} , 75 rules describing Cl_U^{\geq} and 79 rules describing Cl_A^{\geq} .

The *fourth discovery* is the finding of *minimal sets* of decision rules. Several minimal sets were found. One of them is shown below. The number in parenthesis indicates the number of objects which support the corresponding rule, i.e. the rule strength:

1. ***if*** $f(x, A_3) \geq 67.5$ ***and*** $f(x, A_4) \geq -2.2$ ***and*** $f(x, A_6) \geq 8.7$, ***then*** $x \in Cl_{NA}^{\leq}$, (4),
2. ***if*** $f(x, A_2) \leq 3.3$ ***and*** $f(x, A_7) \leq 2$, ***then*** $x \in Cl_{NA}^{\leq}$, (5),
3. ***if*** $f(x, A_3) \geq 63.6$ ***and*** $f(x, A_7) \leq 3$ ***and*** $f(x, A_9) \leq 3$, ***then*** $x \in Cl_{NA}^{\leq}$, (4),
4. ***if*** $f(x, A_2) \leq 12.4$ ***and*** $f(x, A_6) \geq 5.6$, ***then*** $x \in Cl_U^{\leq}$, (14),
5. ***if*** $f(x, A_7) \leq 3$, ***then*** $x \in Cl_U^{\leq}$, (18),
6. ***if*** $f(x, A_2) \geq 3.5$ ***and*** $f(x, A_5) \leq 8.5$, ***then*** $x \in Cl_U^{\geq}$, (26),
7. ***if*** $f(x, A_7) \geq 4$, ***then*** $x \in Cl_U^{\geq}$, (21),
8. ***if*** $f(x, A_1) \geq 8.7$ ***and*** $f(x, A_9) \geq 4$, ***then*** $x \in Cl_U^{\geq}$, (27),
9. ***if*** $f(x, A_2) \geq 3.5$ ***and*** $f(x, A_7) \geq 4$, ***then*** $x \in Cl_A^{\geq}$, (20).

As the minimal set of rules is complete and composed of D_{\geq} -decision rules and D_{\leq} -decision rules only, application of these rules to the 39 firms will result in their exact re-classification to classes of risk.

Minimal sets of decision rules represent the most concise and non-redundant knowledge representations. The above minimal set of 9 decision rules uses 8 criteria and 18 elementary conditions, i.e. 3.85 % of descriptors from the data matrix.

The well-known machine discovery methods cannot deal with multicriteria classification because they do not consider preference orders in the domains of attributes and among the classes. There are multicriteria decision analysis methods for such

classification. However, they are not discovering classification patterns from data. They simply apply a preference model, like the utility function in scoring methods (see, for example, Thomas et al. 1992), to a set of objects to be classified. In this sense, they are not knowledge discovery methods at all.

Comparing the DRSA to the Indiscernibility-Based Rough Set Approach, we can notice the following differences between the two approaches. The Indiscernibility-Based Rough Set Approach extracts knowledge about a partition of U into classes which are not preference-ordered. The granules used for knowledge representation are sets of objects which are indiscernible by a set of condition attributes.

In the case of the DRSA and multicriteria classification, the condition attributes are criteria and the classes are preference-ordered. The extracted knowledge concerns a collection of upward and downward unions of classes and the granules used for knowledge representation are sets of objects defined using the dominance relation. This is the main difference between the Indiscernibility-Based Rough Set Approach and the DRSA.

There are three notable advantages of the DRSA over the Indiscernibility-Based Rough Set Approach. The first one is the ability to handle criteria, preference-ordered classes and inconsistencies in the set of decision examples that the Indiscernibility-Based Rough Set Approach is simply not able to discover. Consequently, the rough approximations separate the certain information from the doubtful, which is taken into account in rule induction. The second advantage is the ability to analyze a data matrix without any preprocessing of data. The third advantage lies in the richer syntax of decision rules that are induced from rough approximations. The elementary conditions of decision rules resulting from the DRSA use relations from $\{\leq, =, \geq\}$, while those resulting from the Indiscernibility-Based Rough Set Approach only use $=$. The DRSA syntax is more understandable to practitioners. The minimal sets of decision rules are smaller than the minimal sets which result from the Indiscernibility-Based Rough Set Approach.

19.5 The DRSA to Multicriteria Choice and Ranking

One of the very first extensions of the DRSA concerned preference-ordered data representing pairwise comparisons (i.e. binary relations) between objects on both condition and decision attributes (Greco et al. 1999a,b, 2000d, 2001c). Note that while classification is based on the absolute evaluation of objects, choice and ranking refer to pairwise comparisons of objects. In this case, the patterns (i.e. decision rules) to be discovered from the data characterize a comprehensive binary relation on the set of objects. If this relation is a preference relation and if, from among the condition attributes, there are some criteria which are semantically correlated with the comprehensive preference relation, then the data set (serving as the learning sample) can be considered to be preference information for a decision maker in a multicriteria choice or ranking problem. In consequence, the comprehensive preference relation characterized by the decision rules discovered from this data set

can be considered as a *preference model* for the decision maker. It may be used to explain the decision policy of the decision maker and to recommend a good choice or preference ranking with respect to new objects.

Let us consider a finite set A of objects evaluated by a finite set of criteria C . The best choice (or the preference ranking) in set A is semantically correlated with the criteria from set C . The preference information concerning the multicriteria choice or ranking problem is a data set in the form of a pairwise comparison table which includes pairs of some *reference objects* from a subset $B \subseteq A \times A$. This is described by preference relations on particular criteria and a comprehensive preference relation. One such example is a weak preference relation called the *outranking relation*. By using the DRSA for the analysis of the pairwise comparison table, we can obtain a rough approximation of the outranking relation by a dominance relation. The decision rules induced from the rough approximation are then applied to the complete set A of the objects associated with the choice or ranking. As a result, one obtains a four-valued outranking relation on this set. In order to obtain a recommendation, it is advisable to use an exploitation procedure based on the net flow score of the objects. We present this methodology in more detail below.

19.5.1 The Pairwise Comparison Table as Preference Information and as a Learning Sample

A set of reference objects represents a decision problem and a decision maker can express the preferences by pairwise comparisons. In the following, $xS^c y$ will denote the presence, while $xS^c y$ denotes the absence of the outranking relation for a pair of objects $(x, y) \in A \times A$.

For each pair of reference objects $(x, y) \in B \subseteq A \times A$, the decision maker can select one of the three following possibilities:

1. Object x is at least as good as y , i.e. $xS^c y$
2. Object x is worse than y , i.e. $xS^c y$
3. The two objects are incomparable at the present stage.

A pairwise comparison table, denoted by S_{PCT} , is then created on the basis of this information. The first m columns correspond to the criteria from set C . The last, i.e. the $(m+1)$ th, column represents the comprehensive binary preference relation S or S^c . The rows correspond to the pairs of objects from B . For each pair in S_{PCT} , a difference between criterion values is put in the corresponding column. If the decision maker judges that two objects are incomparable, then the corresponding pair does not appear in S_{PCT} .

We will define S_{PCT} more formally. For any criterion $g_i \in C$, let T_i be a finite set of binary relations defined on A on the basis of the evaluations of objects from A with respect to the considered criterion g_i , such that for every $(x, y) \in A \times A$ exactly one binary relation $t \in T_i$ is verified. More precisely, given the domain V_i of $g_i \in C$, if $v'_i v''_i \in V_i$ are the respective evaluations of $x, y \in A$ by means of g_i and $(x, y) \in t$,

with $t \in T_i$, then for each $w, z \in A$ having the same evaluations v'_i, v''_i by means of $g_i, (w, z) \in t$. Furthermore, let T_d be a set of binary relations defined on set A (comprehensive pairwise comparisons) such that at most one binary relation $t \in T_d$ is verified for every $(x, y) \in A \times A$.

The *pairwise comparison table* is defined as data table $SPCT = \langle B, C \cup \{d\}, T_G \cup T_d, f \rangle$, where $B \subseteq A \times A$ is a non-empty set of exemplary pairwise comparisons of reference objects, $T_G = \bigcup_{q \in C} T_q$, d is a decision corresponding to

the comprehensive pairwise comparison (comprehensive preference relation), and $f : B \times (C \cup \{d\}) \rightarrow T_G \cup T_d$ is a total function such that $f[(x, y), q] \in T_i$ for every $(x, y) \in A \times A$ and for each $g_i \in C$, and $f[(x, y), q] \in T_d$ for every $(x, y) \in B$. It follows that for any pair of reference objects $(x, y) \in B$ there is verified one and only one binary relation $t \in T_d$. Thus, T_d induces a partition of B . In fact, the data table $SPCT$ can be seen as a decision table, since the set of considered criteria C and the decision d are distinguished.

We assume that the exemplary pairwise comparisons made by the decision maker can be represented in terms of *graded preference relations* (for example “very large preference”, “large preference”, “strict preference”, “strong preference” and “very strong preference”), denoted by P_q^h . For each $q \in C$ and for every $(x, y) \in A \times A$, $T_i = \{ \bigcup_{q \in C} T_q, h \in H_i \}$, where H_i is a particular subset of the relative integers and

- $xP_i^h y, h > 0$, means that object x is preferred to object y by degree h with respect to criterion g_i ,
- $xP_i^h y, h < 0$, means that object x is not preferred to object y by degree h with respect to criterion g_i ,
- $xP_i^0 y$ means that object x is similar (asymmetrically indifferent) to object y with respect to criterion g_i .

Within the preference context, the similarity relation P_i^0 , even if not symmetric, resembles the indifference relation. Thus, in this case, we call this similarity relation *asymmetric indifference*. Of course, for each $g_i \in C$ and for every $(x, y) \in A \times A$,

$$[xP_i^h y, h > 0] \Rightarrow [yP_i^k x, k \leq 0], [xP_i^h y, h < 0] \Rightarrow [yP_i^k x, k \geq 0].$$

The set of binary relations T_d may be defined in a similar way, but $xP_d^h y$ means that object x is comprehensively preferred to object y by degree h . We are considering a pairwise comparison table where the set T_d is composed of two binary relations defined on A :

- x outranks y (denoted by xSy or $(x, y) \in S$), where $(x, y) \in B$,
- x does not outrank y (denoted by $xS^c y$ or $(x, y) \in S^c$), where $(x, y) \in B$, and $S \cup S^c = B$.

Observe that the binary relation S is reflexive, but not necessarily transitive or complete.

19.5.2 Rough Approximation of the Outranking and Non-outranking Relations Specified in the Pairwise Comparison Table

In the following we will distinguish between two types of evaluation scales of criteria: *cardinal* and *ordinal*. Let C^N be the set of criteria expressing preferences on a cardinal scale, and let C^O , be the set of criteria expressing preferences on an ordinal scale, such that $C^N \cup C^O = C$ and $C^N \cap C^O = \emptyset$. Moreover, for each $P \subseteq C$, we denote by P^O the subset of P composed of criteria expressing preferences on an ordinal scale, i.e. $P^O = P \cap C^O$, and by P^N we denote the subset of P composed of criteria expressing preferences on a cardinal scale, i.e. $P^N = P \cap C^N$. Of course, for each $P \subseteq C$, we have $P = P^N \cup P^O$ and $P^N \cap P^O = \emptyset$.

The meaning of the two scales is such that in the case of the cardinal scale we can specify the intensity of preference for a given difference of evaluations, while in the case of the ordinal scale, this is not possible and we can only establish an order of evaluations.

19.5.2.1 Multigraded Dominance

Let $P = P^N$ and $P^O = \emptyset$. Given $P \subseteq C$ ($P \neq \emptyset$), $(x, y), (w, z) \in A \times A$, the pair of objects (x, y) is said to dominate (w, z) with respect to criteria from P (denoted by $(x, y)D_P(w, z)$), if x is preferred to y at least as strongly as w is preferred to z with respect to each $g_i \in P$. More precisely, “at least as strongly as” means “by at least the same degree”, i.e. $h_i \geq k_i$, where $h_i, k_i \in H_i$, $xP^{h_i}y$ and $wP_i^{k_i}z$ for each $g_i \in P$.

Let $D_{\{i\}}$ be the dominance relation confined to the single criterion $g_i \in P$. The binary relation $D_{\{i\}}$ is reflexive $((x, y)D_{\{i\}}(x, y)$, for every $(x, y) \in A \times A$), transitive $((x, y)D_{\{i\}}(w, z)$ and $(w, z)D_{\{i\}}(u, v)$ imply $(x, y)D_{\{i\}}(u, v)$, for every $(x, y), (w, z), (u, v) \in A \times A$), and complete $((x, y)D_{\{i\}}(w, z)$ and/or $(w, z)D_{\{i\}}(x, y)$, for all $(x, y), (w, z) \in A \times A$). Therefore, $D_{\{i\}}$ is a complete preorder on $A \times A$. Since the intersection of complete preorders is a partial preorder and

$$D_P = \bigcap_{g_i \in P} D_{\{i\}}, \quad P \subseteq C$$

then the dominance relation D_P is a partial preorder on $A \times A$.

Let $R \subseteq P \subseteq C$ and $(x, y), (u, v) \in A \times A$; then the following implication holds:

$$(x, y)D_P(u, v) \Rightarrow (x, y)D_R(u, v).$$

Given $P \subseteq C$ and $(x, y) \in A \times A$, we define the following:

- A set of pairs of objects dominating (x, y) , called the *P-dominating set*, denoted by $D_P^+(x, y)$ and defined to be $\{(w, z) \in A \times A : (w, z)D_P(x, y)\}$.
- A set of pairs of objects dominated by (x, y) , called the *P-dominated set*, denoted by $D_P^-(x, y)$ and defined as $\{(w, z) \in A \times A : (x, y)D_P(w, z)\}$.

The P -dominating sets and the P -dominated sets defined on B for all pairs of reference objects from B are *granules of knowledge* that can be used to express P -lower and P -upper approximations of the comprehensive outranking relations S and S^c , respectively:

$$\begin{aligned}\underline{P}(S) &= \{(x, y) \in B : D_P^+(x, y) \subseteq S\}, \\ \overline{P}(S) &= \bigcup_{(x, y) \in S} D_P^+(x, y), \\ \underline{P}(S^c) &= \{(x, y) \in B : D_P^-(x, y) \subseteq S^c\}, \\ \overline{P}(S^c) &= \bigcup_{(x, y) \in S^c} D_P^-(x, y)\end{aligned}$$

It has been proved by [Greco et al. \(1999a\)](#) that

$$\underline{P}(S) \subseteq S \subseteq \overline{P}(S), \quad \underline{P}(S^c) \subseteq S^c \subseteq \overline{P}(S^c).$$

Furthermore, the following complementarity properties hold:

$$\begin{aligned}\underline{P}(S) &= B - \overline{P}(S^c), \quad \overline{P}(S) = B - \underline{P}(S^c) \\ \underline{P}(S^c) &= B - \overline{P}(S), \quad \overline{P}(S^c) = B - \underline{P}(S).\end{aligned}$$

The P -boundaries (P -doubtful regions) of S and S^c are defined as

$$Bn_P(S) = \overline{P}(S) - \underline{P}(S), \quad Bn_P(S^c) = \overline{P}(S^c) - \underline{P}(S^c).$$

From the above, it follows that $Bn_P(S) = Bn_P(S^c)$.

The concepts of the quality of approximation, reducts and core can be extended also to the approximation of the outranking relation by multigraded dominance relations.

In particular, the coefficient

$$\gamma_P = \frac{|\underline{P}(S) \cup \underline{P}(S^c)|}{|B|}$$

defines the *quality of approximation* of S and S^c by $P \subseteq C$. It expresses the ratio of all pairs of reference objects $(x, y) \in B$ correctly assigned to S and S^c by the set P of criteria to all the pairs of objects contained in B . Each minimal subset $P \subseteq C$, such that $\gamma_P = \gamma_C$, is called a *reduct* of C (denoted by RED_{SPCT}). Note that $SPCT$ can have more than one reduct. The intersection of all B -reducts is called the *core* (denoted by $CORE_{SPCT}$).

It is also possible to use the Variable Consistency Model on $SPCT$ ([Slowinski et al. 2002b](#)) but being aware that some of the pairs in the positive or negative dominance sets belong to the opposite relation but at least $l * 100\%$ of pairs belong to the correct one. Then the definition of the lower approximations of S and S^c boils down to

$$\underline{P}(S) = \left\{ (x, y) \in B : \frac{|D_P^+(x, y) \cap S|}{|D_P^+(x, y)|} \geq l \right\}$$

$$\underline{P}(S^c) = \left\{ (x, y) \in B : \frac{|D_P^-(x, y) \cap S^c|}{|D_P^-(x, y)|} \geq l \right\}.$$

19.5.2.2 Dominance Without Degrees of Preference

The degree of graded preference considered above is defined on a cardinal scale of the strength of preference. However, in many real-world problems, the existence of such a quantitative scale is questionable. This is the case with ordinal scales of criteria. In this case, the dominance relation is defined directly on evaluations $g_i(x)$ for all objects $x \in A$. Let us explain this latter case in more detail.

Let $P = P^O$ and $P^N = \emptyset$, then, given $(x, y), (w, z) \in A \times A$, the pair (x, y) is said to dominate the pair (w, z) with respect to criteria from P (denoted by $(x, y)D_P(w, z)$), if for each $g_i \in P$, $g_i(x) \geq g_i(w)$ and $g_i(z) \geq g_i(y)$.

Let $D_{\{i\}}$ be the dominance relation confined to the single criterion $g_i \in P^O$. The binary relation $D_{\{i\}}$ is reflexive, transitive, but non-complete (it is possible that *not* $(x, y)D_{\{i\}}(w, z)$ and *not* $(w, z)D_{\{i\}}(x, y)$ for some $(x, y), (w, z) \in A \times A$). Therefore, $D_{\{i\}}$ is a partial preorder. Since the intersection of partial preorders is also a partial preorder and

$$D_P = \bigcap_{g_i \in P} D_{\{i\}}, \quad P = P^O$$

then the dominance relation D_P is a partial preorder.

If some criteria from $P \subseteq C$ express preferences on a quantitative or a numerical non-quantitative scale and others on an ordinal scale, i.e. if $P^N \neq \emptyset$ and $P^O \neq \emptyset$, then, given $(x, y), (w, z) \in A \times A$, the pair (x, y) is said to dominate the pair (w, z) with respect to criteria from P , if (x, y) dominates (w, z) with respect to both P^N and P^O . Since the dominance relation with respect to P^N is a partial preorder on $A \times A$ (because it is a multigraded dominance) and the dominance with respect to P^O is also a partial preorder on $A \times A$ (as explained above), then the dominance D_P , being the intersection of these two dominance relations, is a partial preorder. In consequence, all the concepts introduced in the previous section can be restored using this specific definition of dominance.

19.5.3 Induction of Decision Rules from Rough Approximations of Outranking and Non-outranking Relations

Using the rough approximations of S and S^c defined in the previous section, it is possible to induce a generalized description of the preference information contained in a given S_{PCT} in terms of suitable decision rules. The syntax of these rules is based on the concept of *upward cumulated preferences* (denoted by $P_i^{\geq h}$) and *downward cumulated preferences* (denoted by $P_i^{\leq h}$), having the following interpretation:

- $xP_i^{\geq h}y$ means “ x is preferred to y with respect to g_i by at least degree h ”,
- $xP_i^{\leq h}y$ means “ x is preferred to y with respect to g_i by at most degree h ”.

Exact definition of the cumulated preferences, for each $(x, y) \in A \times A$, $g_i \in C$ and $h \in H_i$, can be represented as follows:

- $xP_i^{\geq h}y$ if $xP_i^k y$, where $k \in H_i$ and $k \geq h$,
- $xP_i^{< h}y$ if $xP_i^k y$, where $k \in H_i$ and $k \leq h$.

Let also $G_i = \{g_i(x), x \in A\}$, $g_i \in C^O$. The decision rules then have the following syntax:

1. D_{\geq} -decision rules:

If $xP_{i1}^{\geq h(i1)}y$ and $\dots xP_{ie}^{\geq h(ie)}y$ and $g_{ie+1}(x) \geq r_{ie+1}$
 and $g_{ie+1}(y) \leq s_{ie+1}$ and $\dots g_{ip}(x) \geq r_{ip}$ and $g_{ip}(y) \leq s_{ip}$,
 then xSy ,

where

$$P = \{g_{i1}, \dots, g_{ip}\} \subseteq C, P^N = \{g_{i1}, \dots, g_{ie}\}, P^O = \{g_{ie+1}, \dots, g_{ip}\}, \\ (h(i1), \dots, h(ie)) \in H_{i1} \times \dots \times H_{ie} \\ \text{and } (r_{ie+1}, \dots, r_{ip}), (s_{ie+1}, \dots, s_{ip}) \in G_{ie+1} \times \dots \times G_{ip}.$$

These rules are supported by pairs of objects from the P -lower approximation of S only.

2. D_{\leq} -decision rules:

If $xP_{i1}^{\leq h(i1)}y$ and $\dots xP_{ie}^{\leq h(ie)}y$ and $g_{ie+1}(x) \leq r_{ie+1}$
 and $g_{ie+1}(y) \geq s_{ie+1}$ and $\dots g_{ip}(x) \leq r_{ip}$ and $g_{ip}(y) \geq s_{ip}$,
 then $xS^c y$,

where

$$P = \{g_{i1}, \dots, g_{ip}\} \subseteq C, P^N = \{g_{i1}, \dots, g_{ie}\}, P^O = \{g_{ie+1}, \dots, g_{ip}\}, \\ (h(i1), \dots, h(ie)) \in H_{i1} \times \dots \times H_{ie} \\ \text{and } (r_{ie+1}, \dots, r_{ip}), (s_{ie+1}, \dots, s_{ip}) \in G_{ie+1} \times \dots \times G_{ip}.$$

These rules are supported by pairs of objects from the P -lower approximation of S^c only.

3. $D_{\geq\leq}$ -decision rules:

If $xP_{i1}^{\geq h(i1)}y$ and $\dots xP_{ie}^{\geq h(ie)}y$ and $xP_{ie+1}^{\leq h(ie+1)}y \dots xP_{if}^{\leq h(if)}y$
 and $g_{if+1}(x) \geq r_{if+1}$ and $g_{if+1}(y) \leq s_{if+1}$ and $\dots g_{ig}(x) \geq r_{ig}$ and $g_{ig}(y) \leq s_{ig}$
 and $g_{ig+1}(x) \leq r_{ig+1}$ and $g_{ig+1}(y) \geq s_{ig+1}$ and $\dots g_{ip}(x) \leq r_{ip}$ and $g_{ip}(y) \geq s_{ip}$,
 then xSy or $xS^c y$,

where

$$O' = \{g_{i1}, \dots, g_{ie}\} \subseteq C, O'' = \{g_{ie+1}, \dots, g_{if}\} \subseteq C, \\ P^N = O' \cup O'', O' \text{ and } O'' \text{ are not necessarily disjoint}, \\ P^O = \{g_{if+1}, \dots, g_{ip}\}, \\ (h(i1), \dots, h(if)) \in H_{i1} \times \dots \times H_{if}, \\ (r_{if+1}, \dots, r_{ip}), (s_{if+1}, \dots, s_{ip}) \in G_{if+1} \times \dots \times G_{ip}.$$

These rules are supported by pairs of objects from the P -boundary of S and S^c only.

19.5.4 Use of Decision Rules for Decision Support

The decision rules induced from a given S_{PCT} describe the comprehensive preference relations S and S^c either exactly (D_{\geq} - and D_{\leq} -decision rules) or approximately ($D_{\geq\leq}$ -decision rules). A set of these rules covering all pairs of S_{PCT} represent a preference model from the decision maker who gave the pairwise comparison of reference objects. The application of these decision rules on a new subset $M \subseteq A$ of objects induces a specific preference structure on M .

In fact, any pair of objects $(u, v) \in M \times M$ can match the decision rules in one of four ways:

- at least one D_{\geq} -decision rule and neither D_{\leq} - nor $D_{\geq\leq}$ -decision rules,
- at least one D_{\leq} -decision rule and neither D_{\geq} - nor $D_{\geq\leq}$ -decision rules,
- at least one D_{\geq} -decision rule and at least one D_{\leq} -decision rule, or at least one $D_{\geq\leq}$ -decision rule, or at least one $D_{\geq\leq}$ -decision rule and at least one D_{\geq} - and/or at least one D_{\leq} -decision rule,
- no decision rule.

These four ways correspond to the following four situations of outranking, respectively:

- uSv and *not* uS^cv , i.e. *true* outranking (denoted by $uS^T v$),
- uS^cv and *not* uSv , i.e. *false* outranking (denoted by $uS^F v$),
- uSv and uS^cv , i.e. *contradictory* outranking (denoted by $uS^K v$),
- *not* uSv and *not* uS^cv , i.e. *unknown* outranking (denoted by $uS^U v$).

The four above situations, which together constitute the so-called *four-valued outranking* (Tsoukias and Vincke 1995; Greco et al. 1998c), have been introduced to underline the presence and absence of *positive* and *negative* reasons for the outranking. Moreover, they make it possible to distinguish contradictory situations from unknown ones.

A final *recommendation* (choice or ranking) can be obtained upon a suitable exploitation of this structure, i.e. of the presence and the absence of outranking S and S^c on M . A possible exploitation procedure consists of calculating a specific score, called the Net Flow Score, for each object $x \in M$:

$$S_{nf}(x) = S^{++}(x) - S^{+-}(x) + S^{-+}(x) - S^{--}(x)$$

where

$$\begin{aligned} S^{++}(x) &= \text{card}(\{y \in M: \text{there is at least one decision rule which affirms } xSy\}), \\ S^{+-}(x) &= \text{card}(\{y \in M: \text{there is at least one decision rule which affirms } ySx\}), \\ S^{-+}(x) &= \text{card}(\{y \in M: \text{there is at least one decision rule which affirms } yS^cx\}), \\ S^{--}(x) &= \text{card}(\{y \in M: \text{there is at least one decision rule which affirms } xS^cy\}). \end{aligned}$$

The recommendation in ranking problems consists of the total preorder determined by $S_{nf}(x)$ on M . In choice problems, it consists of the object(s) $x^* \in M$ such that $S_{nf}(x^*) = \max_{x \in M} \{S_{nf}(x)\}$.

Table 19.5 Decision table with reference objects

Warehouse	A_1	A_2	A_3	d (ROE %)
1	Good	Medium	Good	10.35
2	Good	Sufficient	Good	4.58
3	Medium	Medium	Good	5.15
4	Sufficient	Medium	Medium	-5
5	Sufficient	Medium	Medium	2.42
6	Sufficient	Sufficient	Good	2.98
7	Good	Medium	Good	15
8	Good	Sufficient	Good	-1.55

The above procedure has been characterized with reference to a number of desirable properties by Greco et al. (1998c).

Recently, Fortemps et al. (2008) extended the DRSA to multicriteria choice and ranking on multi-graded preference relations, instead of simple S and S^c .

19.5.5 An Illustrative Example

Let us suppose that a company managing a chain of warehouses wants to buy some new warehouses. To choose the best proposals or to rank them all, the managers of the company decide to analyze first the characteristics of eight warehouses already owned by the company (reference objects). This analysis should give some indications for the choice and ranking of the new proposals. Eight warehouses belonging to the company have been evaluated by the following three criteria: capacity of the sales staff (A_1), perceived quality of goods (A_2) and high traffic location (A_3). The domains (scales) of these attributes are presently composed of three preference-ordered echelons: $V_1 = V_2 = V_3 = \{\text{sufficient}, \text{medium}, \text{good}\}$. The decision attribute (d) indicates the profitability of warehouses, expressed by the return on equity (ROE) ratio (in %). Table 19.5 presents a decision table which represents this situation.

With respect to the set of criteria $C = C^N = \{A_1, A_2, A_3\}$, the following multi-graded preference relations P_i^h , $i = 1, 2, 3$, are defined:

- xP_i^0y (and yP_i^0x), meaning that x is *indifferent* to y with respect to A_i , if $f(x, A_i) = f(y, A_i)$.
- xP_i^1y (and $yP_i^{-1}x$), meaning that x is *preferred* to y with respect to A_i , if $f(x, A_i) = \text{good}$ and $f(y, A_i) = \text{medium}$, or if $f(x, A_i) = \text{medium}$ and $f(y, A_i) = \text{sufficient}$,
- xP_i^2y (and $yP_i^{-2}x$), meaning that x is *strongly preferred* to y with respect to A_i , if $f(x, A_i) = \text{good}$ and $f(y, A_i) = \text{sufficient}$.

Using the decision attribute, the comprehensive outranking relation was built as follows: warehouse x is at least as good as warehouse y with respect to profitability (xSy) if

$$ROE(x) \geq ROE(y) - 2\%.$$

Otherwise, i.e. if $ROE(x) < ROE(y) - 2\%$, warehouse x is *not* at least as good as warehouse y with respect to profitability ($xS^c y$).

The pairwise comparisons of the reference objects result in S_{PCT} . The rough set analysis of the S_{PCT} leads to the conclusion that the set of decision examples on the reference objects is inconsistent. The quality of approximation of S and S^c by all criteria from set C is equal to 0.44. Moreover, $RED_{S_{PCT}} = CORE_{S_{PCT}} = \{A_1, A_2, A_3\}$. This means that no criterion is superfluous.

The C -lower approximations of S and S^c , obtained by means of multigraded dominance relations, are

$$\underline{C}(S) = \{(1,2), (1,4), (1,5), (1,6), (1,8), (3,2), (3,4), (3,5), \\ (3,6), (3,8), (7,2), (7,4), (7,5), (7,6), (7,8)\}$$

$$\underline{C}(S^c) = \{(2,1), (2,7), (4,1), (4,3), (4,7), (5,1), (5,3), (5,7), \\ (6,1), (6,3), (6,7), (8,1), (8,7)\}.$$

All the remaining 36 pairs of reference objects belong to the C -boundaries of S and S^c , i.e. $Bn_C(S) = Bn_C(S^c)$.

The following minimal D_{\geq} -decision rules and D_{\leq} -decision rules can be induced from lower approximations of S and S^c , respectively (the figures within parentheses represent the pairs of objects supporting the corresponding rules):

- If $xP_1^{\geq 1}y$ and $xP_2^{\geq 1}y$, then xSy ; ((1,6),(3,6),(7,6))
- If $xP_2^{\geq 1}y$ and $xP_3^{\geq 0}y$, then xSy ; ((1,2),(1,6),(1,8),(3,2),(3,6),(3,8),(7,2),(7,6),(7,8))
- If $xP_2^{\geq 0}y$ and $xP_3^{\geq 1}y$, then xSy ; ((1,4),(1,5),(3,4),(3,5),(7,4),(7,5))
- If $xP_1^{\leq -1}y$ and $xP_2^{\leq -1}y$, then xS^cy ; ((6,1),(6,3),(6,7))
- If $xP_2^{\leq 0}y$ and $xP_3^{\leq -1}y$, then xS^cy ; ((4,1),(4,3),(4,7),(5,1),(5,3),(5,7))
- If $xP_1^{\leq 0}y$ and $xP_2^{\leq -1}y$ and $xP_3^{\leq 0}y$, then xS^cy ; ((2,1),(2,7),(6,1),(6,3),(6,7),(8,1),(8,7)).

Moreover, it is possible to induce five minimal $D_{\leq\leq}$ -decision rules from the boundary of approximation of S and S^c :

- If $xP_2^{\leq 0}y$ and $xP_2^{\geq 0}y$ and $xP_3^{\leq 0}y$ and $xP_3^{\geq 0}y$, then xSy or xS^cy ;

$$((1,1), (1,3), (1,7), (2,2), (2,6), (2,8), (3,1), (3,3), (3,7), (4,4), (4,5), (5,4), \\ (5,5), (6,2), (6,6), (6,8), (7,1), (7,3), (7,7), (8,2), (8,6), (8,8)).$$

- If $xP_2^{\leq -1}y$ and $xP_3^{\geq 1}y$, then xSy or xS^cy ; ((2,4),(2,5),(6,4),(6,5),(8,4),(8,5))
- If $xP_2^{\geq 1}y$ and $xP_3^{\leq -1}y$, then xSy or xS^cy ; ((4,2),(4,6),(4,8),(5,2),(5,6),(5,8))
- If $xP_1^{\geq 1}y$ and $xP_2^{\leq 0}y$ and $xP_3^{\leq 0}y$, then xSy or xS^cy ; ((1,3),(2,3),(2,6),(7,3),(8,3),(8,6))
- If $xP_1^{\geq 1}y$ and $xP_2^{\leq -1}y$, then xSy or xS^cy ; ((2,3),(2,4),(2,5),(8,3),(8,4),(8,5)).

Using all the above decision rules and the Net Flow Score exploitation procedure on ten other warehouses proposed for purchase, the managers can obtain the result presented in Table 19.6. The DRSA gives a clear recommendation:

- For the *choice problem* it suggests the *selection of warehouse 2' and 6'*, having maximum score (11).

Table 19.6 Ranking of warehouses for sale by decision rules and the net flow score procedure

Warehouse for sale	A_1	A_2	A_3	Net flow score	Ranking
1'	Good	Sufficient	Medium	1	5
2'	Sufficient	Good	Good	11	1
3'	Sufficient	Medium	Sufficient	-8	8
4'	Sufficient	Good	Sufficient	0	6
5'	Sufficient	Sufficient	Medium	-4	7
6'	Sufficient	Good	Good	11	1
7'	Medium	Sufficient	Sufficient	-11	9
8'	Medium	Medium	Medium	7	3
9'	Medium	Good	Sufficient	4	4
10'	Medium	Sufficient	Sufficient	-11	9

- For the *ranking problem* it suggests the *ranking* presented in the last column of Table 19.5, as follows:

$$(2', 6') \rightarrow (8') \rightarrow (9') \rightarrow (1') \rightarrow (4') \rightarrow (5') \rightarrow (3') \rightarrow (7', 10').$$

19.5.6 Summary

We have briefly presented the contribution of the DRSA to multicriteria choice and ranking problems. Let us point out the main features of the described methodology:

- The decision maker is asked for the preference information necessary to deal with a multicriteria decision problem in terms of exemplary decisions.
- The rough set analysis of preference information supplies some useful elements of knowledge about the decision situation. These are: the relevance of particular attributes and/or criteria, information about their interaction, minimal subsets of attributes or criteria (reducts) conveying important knowledge contained in the exemplary decisions and the set of the non-reducible attributes or criteria (core).
- The preference model induced from the preference information is expressed in a natural and comprehensible language of “if..., then...” decision rules. The decision rules concern pairs of objects and from them we can determine either the presence or the absence of a comprehensive preference relation. The conditions for the presence are expressed in “at least” terms, and for the absence in “at most” terms, on particular criteria.
- The decision rules do not convert ordinal information into numeric, but keep the ordinal character of input data due to the syntax proposed.
- Heterogeneous information (qualitative and quantitative, ordered and non-ordered) and scales of preference (ordinal, cardinal) can be processed within the DRSA, while classical methods consider only quantitative ordered evaluations (with rare exceptions).
- No prior discretization of the quantitative domains of criteria is necessary.

19.6 Conclusions and Promising Areas of Future Work

We have introduced a knowledge discovery paradigm for multi-attribute and multicriteria decision support, based on the concept of rough sets. Rough set theory provides mathematical tools for dealing with granularity of information and possible inconsistencies in the description of objects. Considering this description as an input data about a decision problem, the knowledge discovery paradigm consists of searching for patterns in the data that facilitate an understanding of the decision maker's preferences and that enable us to recommend a decision which is in line with these preferences. An original component of this paradigm is that it takes into account prior knowledge about preference semantics in the patterns to be discovered.

Knowledge discovery from preference-ordered data differs from usual knowledge discovery, since the former involves preference orders in domains of attributes and in the set of decision classes. This requires that a knowledge discovery method applied to preference-ordered data respects the dominance principle. As this is not the case for the well-known methods of data mining and knowledge discovery, they are not able to discover all relevant knowledge contained in the analyzed data sample and, even worse, they may yield unreasonable discoveries, because of inconsistency with the dominance principle. These deficiencies are addressed in the DRSA. Moreover, this approach enables us to apply a rough set approach to multicriteria decision making. We showed how the approach could be used for multicriteria classification, choice and ranking. In more advanced papers, we have presented many extensions of the approach that make it a useful tool for other practical applications. These extensions are:

- DRSA to decision under risk and uncertainty ([Greco et al. 2001e](#))
- DRSA to decision under uncertainty and time preference ([Greco et al. 2010c](#))
- DRSA handling missing data ([Greco et al. 1999c, 2000a](#))
- DRSA for imprecise object evaluations and assignments ([Dembczynski et al. 2009](#))
- Dominance-based approach to induction of association rules ([Greco et al. 2002a](#))
- Fuzzy-rough hybridization of DRSA ([Greco et al. 1999b, 2000b,c; Greco et al. 2002e; Greco et al. 2004c](#))
- DRSA as a way of operator-free fuzzy-rough hybridization ([Greco et al. 2004c, 2005a, 2007](#))
- DRSA to granular computing ([Greco et al. 2008a, 2009](#))
- DRSA to case-based reasoning ([Greco et al. 2008d](#))
- DRSA for hierarchical structure of evaluation criteria ([Dembczynski et al. 2002](#))
- DRSA to decision involving multiple decision makers ([Greco et al. 2006](#))
- DRSA to interactive multiobjective optimization ([Greco et al. 2008c](#))
- DRSA to interactive evolutionary multiobjective optimization under risk and uncertainty ([Greco et al. 2010d](#)).

The DRSA leads to a preference model of a decision maker in terms of decision rules. The decision rules have a special syntax which involves partial evaluation pro-

files and dominance relations on these profiles. The clarity of the rule representation of preferences enables us to see the limits of other traditional aggregation functions: the utility function and the outranking relation. In several studies ([Greco et al. 2001b](#); [Greco et al. 2002d](#); [Greco et al. 2004a](#); [Slowinski et al. 2002c](#)) we have proposed an axiomatic characterization of these aggregation functions in terms of conjoint measurement theory and in terms of a set of decision rules. In comparison to other studies on the characterization of aggregation functions, our axioms do not require any preliminary assumptions about the scales of criteria. A side-result of these investigations is that the decision rule aggregation (preference model) is the most general among the known aggregation functions. The decision rule preference model fulfills, moreover, the postulate of transparency and interpretability of preference models in decision support.

Dealing with ordered data and monotonicity constraints also makes sense in general classification problems, where the notion of preference has no meaning. Even when the ordering seems irrelevant, the presence or the absence of a property has an ordinal interpretation. If two properties are related, one of the two: the presence or the absence of one property should make more (or less) probable the presence of the other property. A formal proof showing that the Indiscernibility-Based Rough Set Approach is a particular case of the DRSA has been given in [Greco et al. \(2007\)](#). Having this in mind, DRSA can be seen as a general framework for analysis of classification data. Although it has been designed for ordinal classification problems with monotonicity constraints, DRSA can be used to solve a general classification problem where no additional information about ordering is taken into account.

The idea which stands behind this claim is the following ([Blaszcynski et al. 2012](#)). We assume, without loss of generality, that the value sets of all regular attributes are number-coded. While this is natural for numerical attributes, categorical attributes must get numerical codes for categories. In this way, the value sets of all regular attributes get ordered (as all sets of numbers are ordered). Now, to analyze a non-ordinal classification problem using DRSA, we transform the decision table such that each regular attribute is cloned (doubled). It is assumed that the value set of each original attribute is ordered with respect to increasing preference (gain type), and the value set of its clone is ordered with respect to decreasing preference (cost type). Using DRSA, for each $t \in \{1, \dots, n\}$, we approximate two sets of objects from the decision table: class Cl_t and its complement $\neg Cl_t$. Obviously, we can calculate dominance-based rough approximations of the two sets. Moreover, they can serve to induce “if... then...” decision rules recommending assignment to class Cl_t or to its complement $\neg Cl_t$. In this way, we reformulated the original non-ordinal classification problem to an ordinal classification problem with monotonicity constraints. Due to cloning of attributes with opposite preference orders, we can have rules that cover a subspace in the condition space, which is bounded from the top and from the bottom. This leads (without discretization) to more synthetic rules than those resulting from the Indiscernibility-Based Rough Set Approach.

Tricks of the Trade

Below we give some hints about how to start a typical session of rough set analysis of a multi-attribute or multicriteria classification problem.

1. First, prepare the data set so it is composed of objects (examples) described by a set of attributes. In the set of attributes, distinguish the decision attribute from other (condition) attributes. For example, in Sect. 19.4.3, we considered a set of firms evaluated by financial and managerial criteria, assigned to three classes of bankruptcy risk. In terms of the size of the data set, in the case of, say, five condition attributes and three decision classes, the number of objects should not be less than a dozen per class.
2. Check if the decision classes labeled by the decision attribute are preference-ordered. Check also whether or not, among the condition attributes, there is at least one whose domain is also preference ordered such that there is a semantic correlation between this condition attribute and the decision attribute (e.g. the bankruptcy risk of a firm and its “net income/net worth” ratio). If the check is positive, then the DRSA should be used, otherwise, the Indiscernibility-Based Rough Set Approach is sufficient. In the latter case, in order to avoid getting decision rules which are too specific, you may need to group some values of particular attributes (say, to at most seven values per attribute). This step is called discretization.
3. Choose the appropriate software (web addresses for free download are given in the next section) and proceed with your calculations.
4. Calculate the quality of approximation of the classification for the complete set of condition attributes/criteria. A quality value above 0.75 is usually satisfactory. In the case of a lower-quality value, there are too many inconsistencies in the data. So try to get data about the evaluation of the objects on additional attributes/criteria, or eliminate some extremely inconsistent objects from the doubtful region of the classification, or add some new and consistent objects. For example, in Sect. 19.2.1 (the traffic signs example), we added one additional attribute—secondary color (SC). Of course, you may continue the analysis even if the quality is low, but then you will get weaker decision rules.
5. Calculate the minimal subsets of attributes/criteria conveying the relevant knowledge contained in the data (reducts) and the set of non-reducible attributes/criteria (core). You may continue the analysis with a data set confined to a chosen reduct —then the decision rules induced from the reduced data set will represent knowledge contained in the data in terms of attributes/criteria from the reduct only. For example, with the traffic signs, one could eliminate from the data table the column of either shape (S) or primary color (PC), without decreasing the quality of knowledge representation.
6. Using the lower and upper approximations of either decision classes (Indiscernibility-Based Rough Set Approach) or unions of preference-ordered decision classes (DRSA), induce decision rules from the reduced or original decision table. You may either induce a minimal set of rules covering all the objects from the decision table or choose from all induced decision rules a subset of the most inter-

esting rules. For example, this might be the rules with a minimal support of 50 % of objects per class or per union of classes, or rules with no more than three elementary conditions in the premise (see the example of traffic signs in Sect. 19.2.1 and the example of bankruptcy risk in Sect. 19.4.3). Usually, the “minimal cover” set of rules is chosen in the perspective of prediction and the “most interesting” set of rules is chosen in the perspective of explanation. At this stage, an expert may disagree with some rules, but they say nothing apart from the truth hidden in the decision table, so you can show what objects from the decision table support the rules in question and the expert may want to eliminate at least some of them from the data. It is also possible that decision rules seem strange for the expert because there are not enough examples in the decision table.

7. If the expert finds your decision rules too specific and/or too numerous, you may use the variable-precision (Indiscernibility-Based Rough Set Approach) model or the variable-consistency (DRSA) model. Then, you have to specify the required precision or consistency level, say 80 %, and you will finally get fewer decision rules. However, their confidence will vary between 80 and 100 %.

Sources of Additional Information

The community of researchers and practitioners interested in rough set theory and applications is organized in the International Rough Set Society. The society’s web page (roughsets.home.pl/www/) includes information about rough set conferences, about the *Transactions on Rough Sets* published in the Springer Lecture Notes in Computer Science series, and about the *International Journal of Granular Computing, Rough Sets and Intelligent Systems*. This page also includes slides of tutorial presentations on rough sets.

A database of rough set references can be found at <http://rsds.univ.rzeszow.pl>.

The following software is available free in the Internet:

- RSES—Rough Set Exploration System <http://logic.mimuw.edu.pl/~rses>,
- ROUGH Set data Explorer <http://idss.cs.put.poznan.pl/site/rose.html>,
- jMAF—java Multi-criteria and Multi-attribute Analysis Framework, available at <http://www.cs.put.poznan.pl/~jblaszczyński/Site/jRS.html>,
- jRank—ranking generator using the DRSA, see www.cs.put.poznan.pl/mszelag/Software/jRank/jRank.html.

References

- Agrawal R, Mannila H, Srikant R, Toivinen H, Verkamo I (1996) Fast discovery of association rules. In: Fayyad UM et al (eds) Advances in knowledge discovery and data mining. AAAI, Palo Alto, pp 307–328

- Blaszczyński J, Greco S, Slowinski R (2007) Multi-criteria classification—a new scheme for application of dominance-based decision rules. *Eur J Oper Res* 181:1030–1044
- Blaszczyński J, Greco S, Slowinski R, Szelag M (2009) Monotonic variable consistency rough set approaches. *Int J Approx Reason* 50:979–999
- Blaszczyński J, Greco S, Slowinski R (2012) Inductive discovery of laws using monotonic rules. *Engineering Applications of Artificial Intelligence*, 25:284–294
- Blaszczyński J, Slowinski R, Szelag M (2010b) Sequential covering rule induction algorithm for variable consistency rough set approaches. *Inform Sci* 181:987–1002
- Dembczynski K, Greco S, Slowinski R (2002) Methodology of rough-set-based classification and sorting with hierarchical structure of attributes and criteria. *Control Cybern* 31:891–920
- Dembczynski K, Greco S, Slowinski R (2009) Rough set approach to multiple criteria classification with imprecise evaluations and assignments. *Eur J Oper Res* 198:626–636
- Fortemps P, Greco S, Slowinski R (2008) Multicriteria decision support using rules that represent rough-graded preference relations. *Eur J Oper Res* 188:206–223
- Giove S, Greco S, Matarazzo B, Slowinski R (2002) Variable consistency monotonic decision trees. In: Alpigini JJ et al (eds) *Rough sets and current trends in computing*. LNAI 2475. Springer, Berlin, pp 247–254
- Greco S, Matarazzo B, Slowinski R (1998a) A new rough set approach to evaluation of bankruptcy risk. In: Zopounidis C (ed) *Operational tools in the management of financial risk*. Kluwer, Dordrecht, pp 121–136
- Greco S, Matarazzo B, Slowinski R (1998b) Fuzzy similarity relation as a basis for rough approximation. In: Polkowski L, Skowron A (eds) *Rough sets and current trends in computing*. LNAI 1424. Springer, Berlin, pp 283–289
- Greco S, Matarazzo B, Slowinski R, Tsoukias A (1998c) Exploitation of a rough approximation of the outranking relation in multicriteria choice and ranking. In: Stewart TJ, van den Honert RC (eds) *Trends in multicriteria decision making*. LNEMS 465. Springer, Berlin, pp 45–60
- Greco S, Matarazzo B, Slowinski R (1999a) Rough approximation of a preference relation by dominance relations. *Eur J Oper Res* 117:63–83
- Greco S, Matarazzo B, Slowinski R (1999b) The use of rough sets and fuzzy sets in MCDM. In: Gal T et al (eds) *Advances in multiple criteria decision making*. Kluwer, Dordrecht, pp 14.1–14.59
- Greco S, Matarazzo B, Slowinski R (1999c) Handling missing values in rough set analysis of multi-attribute and multi-criteria decision problems. In: Zhong N et al (eds) *New directions in rough sets, data mining and granular-soft computing*. LNAI 1711. Springer, Berlin, pp 146–157
- Greco S, Matarazzo B, Slowinski R (2000a) Dealing with missing data in rough set analysis of multi-attribute and multi-criteria decision problems. In: Zanakis SH et al (eds) *Decision making: recent developments and worldwide applications*. Kluwer, Dordrecht, pp 295–316

- Greco S, Matarazzo B, Slowinski R (2000b) Rough set processing of vague information using fuzzy similarity relations. In: Calude CS, Paun G (eds) *Finite versus infinite—contributions to an eternal dilemma*. Springer, Berlin, pp 149–173
- Greco S, Matarazzo B, Slowinski R (2000c) Fuzzy extension of the rough set approach to multicriteria and multiattribute sorting. In: Fodor J et al (eds) *Preferences and decisions under incomplete knowledge*. Physica, Heidelberg, pp 131–151
- Greco S, Matarazzo B, Slowinski R (2000d) Extension of the rough set approach to multicriteria decision support. *INFOR* 38:161–196
- Greco S, Matarazzo B, Slowinski R (2001a) Rough sets theory for multicriteria decision analysis. *Eur J Oper Res* 129:1–47
- Greco S, Matarazzo B, Slowinski R (2001b) Conjoint measurement and rough set approach for multicriteria sorting problems in presence of ordinal criteria. In: Colorni A et al (eds) *A-MCD-A: aide multi-critère à la décision—multiple criteria decision aiding*. European Commission Report, EUR 19808 EN, pp 117–144
- Greco S, Matarazzo B, Slowinski R (2001c) Rule-based decision support in multicriteria choice and ranking. In: Benferhat S, Besnard P (eds) *Symbolic and quantitative approaches to reasoning with uncertainty*. LNAI 2143. Springer, Berlin, pp 29–47
- Greco S, Matarazzo B, Slowinski R (2001d) Assessment of a value of information using rough sets and fuzzy measures. In: Chocjan J, Leski J (eds) *Fuzzy sets and their applications*. Silesian University of Technology Press, Gliwice, pp 185–193
- Greco S, Matarazzo B, Slowinski R (2001e) Rough set approach to decisions under risk. In: Ziarko W, Yao Y (eds) *Rough sets and current trends in computing*. LNAI 2005. Springer, Berlin, pp 160–169
- Greco S, Matarazzo B, Slowinski R, Stefanowski J (2001f) Variable consistency model of dominance-based rough set approach. In: Ziarko W, Yao Y (eds) *Rough sets and current trends in computing*. LNAI 2005. Springer, Berlin, pp 170–181
- Greco S, Matarazzo B, Slowinski R, Stefanowski J (2001g) An algorithm for induction of decision rules consistent with dominance principle. In: Ziarko W, Yao Y (eds) *Rough sets and current trends in computing*. LNAI 2005. Springer, Berlin, pp 304–313
- Greco S, Matarazzo B, Slowinski R, Stefanowski J (2002a) Mining association rules in preference-ordered data. In: Hacid M-S et al (eds) *Foundations of intelligent systems*. LNAI 2366. Springer, Berlin, pp 442–450
- Greco S, Matarazzo B, Slowinski R (2002b) Rough sets methodology for sorting problems in presence of multiple attributes and criteria. *Eur J Oper Res* 138:247–259
- Greco S, Matarazzo B, Slowinski R (2002c) Multicriteria classification. In: Kloesgen W, Zytkow J (eds) *Handbook of data mining and knowledge discovery*, chap 16.1.9. Oxford University Press, Oxford, pp 318–328
- Greco S, Matarazzo B, Slowinski R (2002d) Preference representation by means of conjoint measurement and decision rule model. In: Bouyssou D et al (eds) *Aiding decisions with multiple criteria—essays in honor of Bernard Roy*. Kluwer, Dordrecht, pp 263–313

- Greco S, Inuiguchi M, Slowinski R (2002e) Dominance-based rough set approach using possibility and necessity measures. In: Alpigini JJ et al (eds) Rough sets and current trends in computing. LNAI 2475. Springer, Berlin, pp 85–92
- Greco S, Inuiguchi M, Slowinski R (2004c) A new proposal for fuzzy rough approximations and gradual decision rule representation. Trans rough sets II. LNCS 3135, Springer, Berlin, pp 319–342
- Greco S, Matarazzo B, Slowinski R (2004a) Axiomatic characterization of a general utility function and its particular cases in terms of conjoint measurement and rough-set decision rules. *Eur J Oper Res* 158:271–292
- Greco S, Pawlak Z, Slowinski R (2004b) Can Bayesian confirmation measures be useful for rough set decision rules? *Eng Appl Artif Intell* 17:345–361
- Greco S, Inuiguchi M, Slowinski R (2005a) Fuzzy rough sets and multiple-premise gradual decision rules. *Int J Approx Reason* 41:179–211
- Greco S, Matarazzo B, Pappalardo N, Slowinski R (2005b) Measuring expected effects of interventions based on decision rules. *J Exp Theor Artif Intell* 17:103–118
- Greco S, Matarazzo B, Slowinski R (2006) Dominance-based rough set approach to decision involving multiple decision makers. In: Greco S et al (eds) Rough sets and current trends in computing. LNCS 4259. Springer, Berlin, pp 306–317
- Greco S, Matarazzo B, Slowinski R (2007) Dominance-based rough set approach as a proper way of handling graduality in rough set theory. Trans rough sets VII. LNCS 4400. Springer, Berlin, pp 36–52
- Greco S, Matarazzo B, Slowinski R (2008a) Granular computing for reasoning about ordered data: the dominance-based rough set approach, chap 15 In: Pedrycz W et al (eds) *Handbook of granular computing*, chap 15. Wiley, Chichester, pp 347–373
- Greco S, Matarazzo B, Slowinski R (2008b) Parameterized rough set model using rough membership and Bayesian confirmation measures. *Int J Approx Reason* 49:285–300
- Greco S, Matarazzo B, Slowinski R (2008c) Dominance-based rough set approach to interactive multiobjective optimization. In: Branke J et al (eds) Multiobjective optimization: interactive and evolutionary approaches. LNCS 5252. Springer, Berlin, pp 121–156
- Greco S, Matarazzo B, Slowinski R (2008d) Case-based reasoning using gradual rules induced from dominance-based rough approximations. In: Wang G et al (eds) *Rough sets and knowledge technology*. LNAI 5009. Springer, Berlin, pp 268–275
- Greco S, Matarazzo B, Slowinski R (2009) Granular computing and data mining for ordered data—the dominance-based rough set approach. In: Meyers RA (ed) *Encyclopedia of complexity and systems science*. Springer, New York, pp 4283–4305
- Greco S, Matarazzo B, Slowinski R (2010a) Algebra and topology for dominance-based rough set approach. In: Ras ZW, Tsay L-S (eds) *Advances in intelligent information systems. Studies in computational intelligence* 265. Springer, Berlin, pp 43–78

- Greco S, Matarazzo B, Slowinski R (2010b) On topological dominance-based rough set approach. *Trans Rough Sets XII. LNCS 6190*. Springer, Berlin, pp 21–45
- Greco S, Matarazzo B, Slowinski R (2010c) Dominance-based rough set approach to decision under uncertainty and time preference. *Ann Oper Res* 176:41–75
- Greco S, Matarazzo B, Slowinski R (2010d) Dominance-based rough set approach to interactive evolutionary multiobjective optimization. In: Greco S et al (eds) *Preferences and decisions: models and applications. Studies in fuzziness and soft computing 257*. Springer, Berlin, pp 225–260
- Grzymala-Busse JW (1992) LERS—a system for learning from examples based on rough sets. In: Slowinski R (ed) *Intelligent decision support. Handbook of applications and advances of the rough sets theory*. Kluwer, Dordrecht, pp 3–18
- Grzymala-Busse JW (1997) A new version of the rule induction system LERS. *Fund Inform* 31:27–39
- Kotlowski W, Dembczynski K, Greco S, Slowinski R (2008) Stochastic dominance-based rough set model for ordinal classification. *Inform Sci* 178:4019–4037
- Krawiec K, Slowinski R, Vanderpooten D (1998) Learning of decision rules from similarity based rough approximations. In: Polkowski L, Skowron A (eds) *Rough sets in knowledge discovery 2*. Physica, Heidelberg, pp 37–54
- Luce RD (1956) Semi-orders and a theory of utility discrimination. *Econometrica* 24:178–191
- Marcus S (1994) Tolerance rough sets, Cech topologies, learning processes. *Bull Pol Acad Sci Tech Sci* 42:471–487
- Michalski RS, Bratko I, Kubat M (eds) (1998) *Machine learning and data mining—methods and applications*. Wiley, New York
- Nieminen J (1988) Rough tolerance equality. *Fund Inform* 11:289–296
- Pawlak Z (1982) Rough sets. *Int J Inform Comput Sci* 11:341–356
- Pawlak Z (1991) *Rough sets. Theoretical aspects of reasoning about data*. Kluwer, Dordrecht
- Pawlak Z, Slowinski R (1994) Rough set approach to multi-attribute decision analysis. *Eur J Oper Res* 72:443–459
- Pawlak Z, Grzymala-Busse JW, Slowinski R, Ziarko W (1995) Rough sets. *Commun ACM* 38:89–95
- Polkowski L (2002) *Rough sets: mathematical foundations*. Physica, Heidelberg
- Polkowski L, Skowron A (1999) Calculi of granules based on rough set theory: approximate distributed synthesis and granular semantics for computing with words. In: Zhong N et al (eds) *New directions in rough sets, data mining and soft-granular computing. LNAI 1711*. Springer, Berlin, pp 20–28
- Polkowski L, Skowron A, Zytkow J (1995) Rough foundations for rough sets. In: Lin TY, Wildberger A (eds) *Soft computing. Simulation Councils*, San Diego, pp 142–149
- Roy B (1996) *Multicriteria methodology for decision aiding*. Kluwer, Dordrecht
- Skowron A (1993) Boolean reasoning for decision rules generation. In: Komorowski J, Ras ZW (eds) *Methodologies for intelligent systems. LNAI 689*. Springer, Berlin, pp 295–305

- Skowron A, Polkowski L (1997) Decision algorithms: a survey of rough set-theoretic methods. *Fund Inform* 27:345–358
- Skowron A, Stepaniuk J (1995) Generalized approximation spaces. In: Lin TY, Wildberger A (eds) *Soft computing. Simulation Councils*, San Diego, pp 18–21
- Słowiński R (1992a) A generalization of the indiscernibility relation for rough set analysis of quantitative information. *Rivista di Matematica per le Scienze Economiche e Sociali* 15:65–78
- Słowiński R (ed) (1992b) Intelligent decision support. *Handbook of applications and advances of the rough sets theory*. Kluwer, Dordrecht
- Słowiński R (1993) Rough set learning of preferential attitude in multi-criteria decision making. In: Komorowski J, Ras ZW (eds) *Methodologies for intelligent systems. LNAI 689*. Springer, Berlin, pp 642–651
- Słowiński R, Vanderpooten D (1997) Similarity relation as a basis for rough approximations. In: Wang PP (ed) *Advances in machine intelligence and soft-computing IV*. Duke University Press, Durham, pp 17–33
- Słowiński R, Vanderpooten D (2000) A generalised definition of rough approximations. *IEEE Trans Data Knowl Eng* 12:331–336
- Słowiński R, Zopounidis C (1995) Application of the rough set approach to evaluation of bankruptcy risk. *Intell Syst Account Finance Manage* 4:27–41
- Słowiński R, Stefanowski J, Greco S, Matarazzo B (2000) Rough sets based processing of inconsistent information in decision analysis. *Control Cybern* 29:379–404
- Słowiński R, Greco S, Matarazzo B (2002a) Rough set analysis of preference-ordered data. In: Alpigini JJ et al (eds) *Rough sets and current trends in computing. LNAI 2475*. Springer, Berlin, pp 44–59
- Słowiński R, Greco S, Matarazzo B (2002b) Mining decision-rule preference model from rough approximation of preference relation. In: Proceedings of the 26th IEEE annual international conference on computer software and applications, Oxford, pp 1129–1134
- Słowiński R, Greco S, Matarazzo B (2002c) Axiomatization of utility, outranking and decision-rule preference models for multiple-criteria classification problems under partial inconsistency with the dominance principle. *Control Cybern* 31:1005–1035
- Słowiński R, Greco S, Matarazzo B (2009) Rough sets in decision making. In: Meyers RA (ed) *Encyclopedia of complexity and systems science*. Springer, New York, pp 7753–7786
- Słowiński R, Greco S, Matarazzo B (2012) Rough set and rule-based multicriteria decision aiding. *Pesqui Oper* 32:213–269
- Stefanowski J (1998) On rough set based approaches to induction of decision rules. In: Polkowski L, Skowron A (eds) *Rough sets in data mining and knowledge discovery 1*. Physica, Heidelberg, pp 500–529
- Stepaniuk J (2000) Knowledge discovery by application of rough set models. In: Polkowski L et al (eds) *Rough set methods and application*. Physica, Heidelberg, pp 137–231

- Thomas LC, Crook JN, Edelman DB (eds) (1992) Credit scoring and credit control. Clarendon, Oxford
- Tversky A (1977) Features of similarity. *Psychol Rev* 84:327–352
- Tsoukiàs A, Vincke Ph (1995) A new axiomatic foundation of partial comparability. *Theory and Decision* 39:79–114
- Yao Y, Wong S (1995) Generalization of rough sets using relationships between attribute values. In: Proceedings of the 2nd annual joint conference on information science, Wrightsville Beach, pp 30–33
- Ziarko W (1993) Variable precision rough sets model. *J Comput Syst Sci* 46:39–59
- Ziarko W (1998) Rough sets as a methodology for data mining. In: Polkowski L, Skowron A (eds) *Rough sets in knowledge discovery 1*. Physica, Heidelberg, pp 554–576
- Ziarko W, Shan N (1994) An incremental learning algorithm for constructing decision rules. In: Ziarko WP (ed) *Rough sets, fuzzy sets and knowledge discovery*. Springer, Berlin, pp 326–334

Chapter 20

Hyper-heuristics

Peter Ross

20.1 Introduction

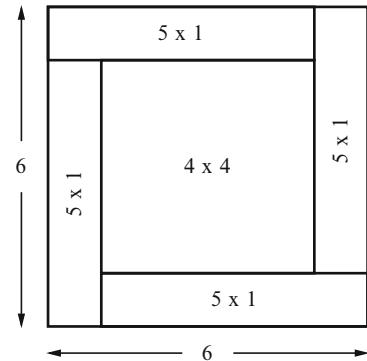
Many practical problems are awkward to solve computationally. Whether you are trying to find any solution at all, or perhaps to find a solution that is optimal or close to optimal according to some criteria, exact methods can be unfeasibly expensive. In such cases it is common to resort to heuristic methods, which are typically derived from experience but are inexact or incomplete. For example, in packing and cutting problems a very simple heuristic might be to try to pack the items in some standardized way starting with the largest remaining one first, on the reasonable grounds that the big ones tend to cause the most trouble. But such heuristics can easily lead to suboptimal answers.

Suppose you are given a supply of 6×6 sheets of metal and have to cut them to produce forty 5×1 sheets and ten larger, 4×4 , sheets (you are not restricted to guillotine cuts, that is, to edge-to-edge straight cuts). The total area of the required pieces is 360, which suggests that at least 10 sheets are needed. If you start with the biggest ones, by cutting out a 4×4 sheet from a corner, you are doomed to waste material; the sole optimal arrangement per sheet (apart from reflection) is shown in Fig. 20.1, which wastes no material.

However, it is not easy to conceive of a straightforward and broadly useful heuristic that would suggest such a layout. Creating new heuristics, whether by detailed study of sample problems and their solutions or by personal introspection based on past experience, is unreliable and difficult (see for example Neth et al. 2009). More generally, both intuition and experience suggest that any given heuristic

P. Ross (✉)
School of Computing, Edinburgh Napier University, Edinburgh, UK
e-mail: P.Ross@napier.ac.uk

Fig. 20.1 The optimal cutting layout



has some weaknesses and will recommend some bad decisions in certain cases. *Hyper-heuristics* tries to address such issues, in two main ways:

- By exploring whether some suitable combination of existing heuristics can offset the weaknesses of any one of them, so that each is only applied when it is not weak;
- Or, by trying to discover new heuristics through some kind of meta-heuristic search process (tabu search, genetic algorithm, genetic programming, etc.).

Some authors (for example Cowling et al. 2001) describe hyper-heuristics simply as “heuristics to choose heuristics”. A recent survey and classification of different approaches can be found in Burke et al. (2009a), which also proposes as a definition that a “hyper-heuristic is an automated methodology for selecting or generating heuristics to solve hard computational problems”. Chakhlevitch and Cowling (2008) present another survey with a different classification.

Broadly speaking, some studies aim to produce *constructive* heuristics that will build a solution to a problem step by step: heuristics are used to decide how to extend a partial solution. Other studies aim to produce *refinement* or *perturbing* heuristics that work on a fully-specified candidate solution but try to improve its quality. And some studies aim to produce a hybrid of these, both extending and modifying partial solutions. Constructive methods tend to be fast and have a natural stopping point. Solution-refinement methods can be appreciably slower, perhaps stopping only when no further progress seems likely, but can deliver better final results.

This chapter is an introduction to hyper-heuristics. It discusses the issues that make hyper-heuristics research distinctive, presents some illustrative examples and a brief survey of past research, and offers some suggestions about interesting directions for future research. Finally, there is a selection of useful links to relevant material.

20.2 The Need for Hyper-heuristics

There is a vast literature on methods for tackling problems in combinatorial optimization and operations research (OR), including commercially significant ones such as scheduling and timetabling, vehicle routing, and packing as well as rather more generic (if no less valuable) ones such as constraint satisfaction. In many cases researchers have concentrated their efforts on trying to find one or more solutions that are optimal according to some chosen criteria. However, as Chambers Twentieth Century Dictionary (1974 edition) puts it:

optimal, etc. See **optimism**.

The online Chambers 21st Century Dictionary has updated this to

adj most favourable; optimum. [...].

The earlier version slyly embodies a pragmatic truth: often it is unreasonable to hope to find the very best possible answer. The search may be far too expensive. Or, the optimal answer may be too dependent on the chosen criteria, so that some seemingly small, circumstantial change in the problem or in the criteria renders the discovered optimum useless or seriously sub-optimal. The criteria themselves may be unreasonable or unrealistic. In practice, what is wanted is often good answers, found fairly or very quickly, where “good” means meeting some minimum acceptance criteria or, perhaps, usefully better than present practice can deliver. Also, there is often a requirement that the method of finding such answers must be sustainable for business use.

For example, suppose a large supermarket chain wants to get into the business of providing home deliveries of orders placed over the internet. Their competitors already do it, they feel they must do so as well. Customers will expect that deliveries should be made within some reasonable time window, chosen by the customer from a set of current possibilities. This looks at first sight like a classic instance of a vehicle routing problem with time windows: given a set of deliveries, each to be made within some time window, find the minimum number of vehicles required; the vehicles are assumed all to have a certain average speed and a given maximal capacity. There are many papers available that study such problems; for example, Bräysy (2003) describes a four-stage approach. The first stage uses some route construction heuristics to propose an initial solution. The second stage also uses heuristic methods—it tries to reduce the total number of routes by, repeatedly, selecting a route and seeing if those customers can be squeezed into other routes, perhaps involving some cascade of shifts of customers from those routes to others. The third and fourth stages keep the number of routes fixed but try to improve them, reducing the distance traveled and then exploring changes to a cost function that combines distances and waiting times in case the search has become stuck in a local minimum. The overall aim is to minimize total distance traveled and number of vehicles required, and the approach was tested on a number of standard benchmark problems involving up to 400 customers and also 2 real-world problems with 417 customers each. Although results were very good when compared with previous approaches,

all the approaches took anywhere from half an hour to several hours to solve the real-world instances (see Table 5 in that paper). Those were much more demanding than the benchmark problems because the customers were less spread out, necessitating much more local search effort.

Even if you are not directly interested in vehicle routing problems, it is worth looking at the recent literature because it illustrates various points about solving combinatorial and OR problems:

- The more effective approaches tend to employ two or more stages each of which utilizes one or more heuristics, rather than being monolithic;
- The widely used artificially generated benchmark problems may not be representative of real-world ones;
- If the approach involves some randomized elements, even if only to break ties, then different runs can produce significantly different results and it may be necessary to do several runs because the first one or two may be unreasonably poor—which you don't know until you try.

But there are more pragmatic considerations too. A supermarket probably does not want to spend CPU-hours per day solving one problem for one locality, and doing that for each locality where it operates. Even if it has its own IT department and in-house expertise, it will also have to factor in the cost of maintaining the necessary software and periodically upgrading or replacing it as the situation demands. And the problem itself is not static: solutions can be partly built as orders arrive: late-comers may not be offered the full range of delivery windows if some are already full. There are legal constraints on driver hours; drivers can fall ill at short notice; vehicles can break down; changing traffic patterns will affect average speeds; any delivery can fail because the customer forgets to be there to accept it; and so on. The supermarket is likely to want to be able to produce good, reasonably low-cost answers that are not too fragile in the face of changing situations, rather than solutions that are genuinely optimal with respect to just one or two criteria. Hyper-heuristics offers a possible way to discover an effective, fast process for scheduling the deliveries that can be used day to day, on ever-changing problems, with relatively little effort. [Garrido and Castro \(2009\)](#) used a hybrid hyper-heuristics method to find stable, good-quality solutions for certain kinds of vehicle routing problems.

More generally, this example raises some important issues:

- Some problems are *offline* problems: their full details are available at the start. Others are *online* problems: their details only emerge over time, and a solution must be built incrementally;
- In practice, it will often be important to be able to offer some performance guarantees. These might not be very strong, perhaps just “better than X” rather than (say) “within 5% of optimal according to criterion C”;
- There can be major differences between academic and real-world problems, and between academic and real-world concerns.

It is also useful to consider how success is to be judged. Standards in many fields can sometimes be ad hoc. A recent useful classification, which considers heuristic

methods applied to forest planning, can be found in [Bettinger et al. \(2009\)](#); it offers a multi-level categorization:

Level 1: *No validation or performance is established.*

Level 2: *Self-validation* using basic statistics on the distribution of results. This includes worst-case (level 2a), best-case (level 2b) and average-case (level 2c) results, performance variation (level 2d) and sensitivity analysis of parameters (level 2e).

Level 3: *Comparison with other heuristic results.*

Level 4: *Comparison with an estimated global optimum solution*, for example by applying results from extreme value theory.

Level 5: *Comparison with optimal solutions generated for similar problems*, for example by relaxing the problem to get something more tractable.

Level 6: *Comparison with solutions provided by exact techniques.*

When thinking specifically about hyper-heuristics, these levels need to be modified slightly. It is important to look at performance on a range of test problems, which should be different from any problems used to discover a new candidate heuristic in the first place. After all, the whole rationale of a heuristic is to suggest what to do when faced with an unfamiliar problem, and so it should be evaluated against previously unseen cases. If the discovery process involves iterative testing as well as learning, then (as usual) there should be separate learning, validation and test sets. Each visit to a validation set after a bout of learning produces some feedback that influences later learning episodes, and so the performance on the validation set is likely to increase in subsequent stages. In order to avoid such ‘data snooping’ the test set should be used only once, at the very end. It can also be useful to include an extra level to the above categorization:

Level 7: *Stress-testing*, for example by trying to devise new problems that “break” the heuristic in some way, delivering poor results or introducing previously unconsidered aspects.

Assuming that the aim is to produce a heuristic that has some general applicability, it is also wise to include some easy problems in the test set as well as hard problems. For some types of problem domain, there can be heuristics that *only* do well on hard problems but which struggle when presented with an easy problem!

The next sections present some illustrative examples of hyper-heuristics.

20.3 Hyper-heuristics for Boolean Satisfiability

20.3.1 The Problem Area

Suppose that x_1 , x_2 and x_3 are Boolean variables—that is, each one is either true or false. Find an assignment of true or false to each of the three variables such that the two clauses:

$$\begin{aligned}x_1 \vee \neg x_2 \vee \neg x_3 \\x_1 \vee x_2\end{aligned}$$

are both true (where \vee means “or” and \neg means “not”). Clearly, any assignment in which x_1 is true works; also, any other assignment in which x_3 is false and x_2 is true works. This is a trivial example of a Boolean satisfiability problem. In conjunctive normal form, the problem consists of a number of clauses all of which are required to be true, and each clause is a disjunct (that is, joined by \vee) of some variables and/or negated variables. If each clause contains at most two variables, the problem can be solved in polynomial time; if the clauses contain three or more variables, the problem type is NP-complete. Boolean satisfiability problems occur in many practical applications, such as automated planning, automated software testing, hardware design and biology (Marques-Silva 2008). For example, a newly-fabricated integrated circuit may contain a fault in which the output of a certain gate is stuck at logic 1 or at logic 0. Such a fault may be detected by generating a set of 0/1 inputs that would, if the circuit were correct, produce a set of outputs different from those actually observed; the task of finding a suitable set of inputs can be formulated as an instance of a Boolean satisfiability problem.

If the problem has n variables then there are 2^n possible assignments so that if n is not small exhaustive search, even with some smart pruning, may not be possible. Various heuristics are commonly used. For example:

GSAT: flip the variable that will produce the greatest increase in the number of satisfied clauses (call this the highest gain). If there is more than one such variable, choose randomly between those variables.

GWSAT(p): with probability p , randomly choose an unsatisfied clause and flip a randomly-chosen variable in it. With probability $(1 - p)$, use GSAT.

WalkSat(p): randomly choose an unsatisfied clause. Find those variables which, if flipped, produce no net change in the number of satisfied clauses. If there is at least one such variable, choose one and flip it. If there are none, then with probability p apply GSAT to the variables in the chosen clause, and with probability $(1 - p)$ choose a random variable from that clause.

20.3.2 The Heuristic Generation Process

Bader-El-Den and Poli (2008) analyzed several such heuristics and created a simple recursive grammar that was capable of expressing them as well as many more candidate heuristics. They then used a grammar-constrained form of genetic programming (Poli et al. 2008) to search for a new heuristic that would give good performance on a range of benchmark problems.

The grammar used the terminals and functions shown in Table 20.1, and the set of functions and terminals also happens to imply the grammar actually used, although this need not always be the case. Some functions that select from a list have an optional argument, shown as {op}, used to break ties; the default is to break ties by

Table 20.1 Functions and terminals for heuristics for Boolean satisfiability (Bader-El-Den and Poli 2008)

Functions	
FLIP v	The fixed, top-level-only action: flip the variable v
RANDOM 1	Return a random variable from the list
MAX_SCR 1 {op}	Return the variable with the highest gain.
MIN_SCR 1 {op}	Return the variable with lowest gain.
SCR_Z 1 {op}	Return a variable that, if flipped, produces zero net change in the number of satisfied clauses.
MAX_AGE 1 {op}	Return the variable that has not been flipped for the longest time.
IFV prob v1 v2	With the given probability return variable v1 else v2
IFL prob l1 l2	With the given probability return list l1 else l2
PROB	A probability: one of 0.2, 0.4, 0.5, 0.7, 0.8 or 0.9
Terminals	
ALL	A list of all clauses
ALL_USC	A list of all currently unsatisfied clauses
USC	A random unsatisfied clause (the same at each appearance in the whole expression)
RAND_USC	A random unsatisfied clause (not necessarily the same each time)
TIE_RAND	Flag: break ties at random
TIE_AGE	Flag: break ties by selecting the least-recently flipped.
TIE_SCR	Flag: break ties by gain.
NOT_ZERO_AGE	Flag: skip the just-flipped variable when breaking ties.

random choice. Where appropriate, a clause is treated as a list of variables, whether negated or not, that it contains. For example, GWSAT(0.5) could be expressed in this language as

```
(FLIP (IFV 0.5 (MAX_SCR ALL TIE_RAND) (RANDOM USC))).
```

The initial randomly generated population had some existing good heuristics injected into it, thus providing some known-good material from which to breed. At each stage, each member of the population was evaluated on a set of training problems by using it repeatedly to modify an initial assignment of false to each variable in the problem. Fitness was based on the number of problems solved, the number of flips done and the size of the heuristic.

The training and testing problems were all taken from a standard benchmark collection of hard problems (Satlib 2012), in particular those involving between 25 and 100 variables each, with 3 variables per clause (3-SAT), and all known to have at least one solution. Overall, the results were very encouraging: generated heuristics, trained on one subset of problem, also performed well on other subsets and produced results “that are on par with some of the best-known SAT solvers”. Figure 20.2 shows an example of a generated heuristic.

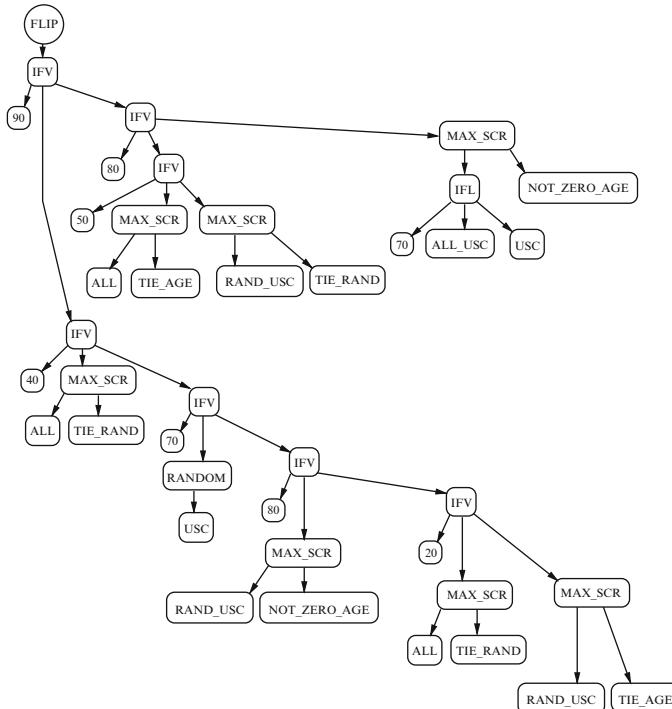


Fig. 20.2 A generated heuristic for 3-SAT problems, from [Bader-El-Den and Poli \(2008\)](#) (amended)

20.3.3 Remarks

[Bader-El-Den and Poli \(2008\)](#) also introduced the notion of a *disposable* heuristic, namely a heuristic intended to be used on a specific sort of subset of a more general family of problems. In this case, the aim was to focus on relatively small problems with three variables per clause that are known to have solutions. In this usage *disposable* means not fully general-purpose, just tailored to a subclass of some kind. Later work by different authors has occasionally abused this notion of *disposable* by focusing on just one problem, trying to generate a heuristic that will solve only a single instance. But, as mentioned earlier, one of the hallmarks of a heuristic is that it should usefully be applicable to unseen problems.

This example is a good illustration of hyper-heuristics: it starts with some well-studied heuristics and devises a search space that includes them and many more variants. [Fukunaga \(2008\)](#) describes a similar study, also using genetic programming but involving a more expressive grammar and a larger search space, which was able to generate heuristics competitive with some of the best SAT algorithms to date. See [Fukunaga \(2002\)](#) for a shorter overview of the grammar and some early results using a much-simplified form of genetic programming.

A somewhat different approach is outlined by [Bittle and Fox \(2009\)](#). They use a version of the SOAR cognitive architecture to generate a constructive hyper-heuristic comprising a large number of condition-action rules, in which the conditions encode aspects of the current state of a solution that is under construction, and the action encodes how to select a variable and a value for it.

20.4 Hyper-heuristics for Timetabling

20.4.1 *The Problem Area*

Timetabling problems are often highly constrained and incompletely specified. In a typical problem the task is to timetable a number of weekly lectures or a number of exams. Some events can only happen in specific rooms, perhaps because they require unusual facilities or because the room must be large enough to hold all the participants. Each person (staff or student) may only be available at some limited times. There are also *soft* constraints that should be honored if possible but can be violated if necessary. For example, it may be desirable to try to space events out or to cluster certain events, but travel timings may make it inconvenient to put some pairs of events too close together. And, of course, it may not be possible to have a full and accurate specification by the time that the timetable has to be finalized.

Heuristics for timetabling problems are similar in spirit to those for SAT and for SAT's close relation, constraint satisfaction problems. In constraint satisfaction problems, each variable has a domain of possible values and there are constraints between pairs of variables, typically in the form of a pair-specific set of disallowed pairs of values. In all three problem areas, solutions are typically constructed by repeatedly choosing a variable and then choosing a value for it, in some heuristic fashion.

It is also common to view timetabling as a graph-coloring problem at heart. Events are represented as nodes and two nodes are linked by an edge if they cannot, for whatever reason, be placed in the same timeslot. The basic task is then to color the nodes of the graph in such a way that no two nodes linked by an edge have the same color—the set of colors represents the set of timeslots. There is a large body of theory about graph coloring, see for example [Kubale \(2004\)](#).

20.4.2 *The Heuristic Generation Process*

[Burke et al. \(2007b\)](#) used up to six heuristics based on graph-coloring notions to choose events. A chosen event would then be inserted into the lowest-cost available timeslot, cost being determined by the soft constraints. The possible heuristics were:

1. LD: (largest degree) choose the event with the largest number of hard constraints;
2. LWD: (largest weighted degree) as LD, but the hard constraints are weighted in some way rather than being regarded as equal;
3. CD: (color degree) choose the event that has the most hard constraints involving already-placed events;
4. LE: (largest enrolment) choose the event involving the most people;
5. SD: (saturation degree) choose the event with the fewest available timeslots;
6. RO: (random order) choose at random.

and different mixes of these were tried.

A solution is represented by a sequence of these heuristics, applied in turn to build a complete solution. Each heuristic was used to place two events before moving on to the next heuristic in the sequence, thus if there were n events to be scheduled the sequence would be $\lfloor \frac{n}{2} \rfloor$ long (no selection method is needed for the final event). The decision to use each heuristic twice was guided by the empirical observation that, if the list was n long then runs often occurred, and then by some experimental exploration of possible choice of repetition factor (2–5).

To begin with, tabu search was used to find the best sequence of heuristics for a family of benchmark timetabling problems. Each sequence was evaluated by constructing a complete timetable, then using a local search to try to improve the timetable further by moving events around and evaluating that improved result. If a sequence produced an invalid timetable, the offending sub-sequence of heuristics would be added to the tabu list.

Later work ([Qu and Burke 2009](#)) tried replacing tabu search by steepest descent, iterated local search and variable neighborhood search. The latter two were found to be better than tabu search for the purpose. [Ochoa et al. \(2009\)](#) analyzed the fitness landscape, observing that there were many plateaus in the landscape but it was also globally convex, suggesting that the landscape does contain the sort of information that could help to guide a search towards the optimum.

[Cowling and Chakhlevitch \(2007\)](#) tackled practical personnel scheduling problems by using a large set of low-level perturbing heuristics, some of which choose an event to try altering and others of which decide how to alter a chosen event. The problems involved 50 or more training staff to be scheduled to handle between 147 and 224 training events in 16 different locations over some 3-month period; when scheduled by hand, they were taking around 9 days of a manager's time. Cowling and Chakhlevitch explored a broad range of ways of combining the heuristics using greedy or mildly greedy (*peckish*) and tabu search strategies. They conducted extensive experiments and concluded that hyper-heuristic methods were able to produce very good solutions, and in a very small fraction of the time previously taken by managers.

20.4.3 Remarks

One of the observations in [Burke et al. \(2007b\)](#) was that, assuming enough search time was allowed, it tended to be beneficial to include random ordering (RO) as one of the heuristics. But it is important to remember that if a solution method includes any non-deterministic components then different runs on the same data can produce different results, and it may then be important to factor in the cost of doing multiple runs in order to sample the space of possible outputs. If speed or reproducibility are of primary importance in your application area, it may be wise to omit any random-based component from the final product.

However, in the development stage, it can be helpful to try including a random-choice heuristic. If its inclusion improves the results, that is suggestive evidence that the other heuristics considered by the search process all failed to offer the good choice that the random-choice heuristic actually made. In that case, perhaps the basic set of heuristics or heuristic ingredients needs to be modified in some way.

20.5 Hyper-heuristics for Packing

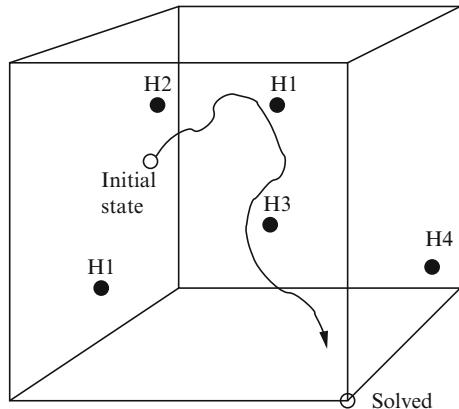
20.5.1 The Problem Area

Given a set of containers, usually all of the same size, how can you pack a given set of objects into them so as to use as few containers as possible? There are many variants on this basic theme. The containers may be one-, two- or three-dimensional. In two or three dimensions, the objects may be rectilinear or may be of any shape. There may be a cost or weight associated with each object so that the task is instead to minimize that cost or weight while packing as many items as possible. The task may be to cut specified shapes from stock material, using only guillotine cuts, or using only cuts parallel to the edges of the material. A taxonomy of such problems can be found in [Dyckhoff \(1990\)](#); see also [Martello and Toth \(1990\)](#) and [Coffman et al. \(1996\)](#).

As before, the basic constructive approach is iterative: select an item and decide where to put it, repeatedly. Some approaches create a rating heuristic that evaluates any given placement of an item, and this rating heuristic is applied at each step to every remaining item in every possible position. Many heuristics have been suggested, for example:

- (For offline problems) select items, largest first, and put them in the first container that will hold them. If M is the minimum number of containers needed, this is known to require no more than $11M/9 + 4$ containers ([Johnson 1973](#));
- (For offline problems) taking largest items first, pack a container until it is at least one-third full, then conduct a search for any single item that will fill the container, or else a combination of two or three items that will fill the bin;

Fig. 20.3 A messy GA approach: the basic idea



- (For online problems) put the next item into the fullest container that will accept it.

There are also fairly obvious perturbing heuristics. For example, if a container is nearly full, such that it will not accept any other item, then search the other containers for a single item or any pair of items that will fill it.

20.5.2 Heuristic Generation Processes

Ross et al. (2003) describe a distinctive approach to solving offline one-dimensional packing problems, using a hyper-heuristic approach to create a very fast constructive heuristic. The idea is to associate heuristics with points or regions in a simplified problem-state space, for example encoding the current state of the partial solution as a vector of five real numbers: the proportion of small items, medium items, large and huge items remaining to be packed, and finally the proportion of the total number of items still to be packed. The definitions of *small*, *medium*, *large* and *huge* are somewhat ad hoc. A messy GA (Goldberg et al. 1989) is used to try to find a number of points in this five-dimensional space, each with an associated heuristic, that can be used to guide the packing process. Each chromosome contains a variable number of genes, and each gene is a block containing a five-dimensional vector and a named heuristic that can be regarded as the *label* for that five-dimensional point. A chromosome therefore describes a set of labeled points in five-dimensional space, as suggested by Fig. 20.3 and is decoded into a complete packing by the algorithm shown in Fig. 20.4.

Fitness of each chromosome was based on using it for a number of training problems, using a rolling regime of sampling problems rather than trying every training problem every time. The single final *algorithm* was able to deliver very creditable results on each of a large number of test problems. Moreover, since the simplified state space could be divided up into a suitably large number of cubes, each marked

```

Until( every item has been packed ) {
    Encode the current problem state;
    In the simplified state-space, find
        the nearest labeled point to the
        current state;
    Apply the heuristic on that label;
    Update the problem state;
}

```

Fig. 20.4 A state-space-guided packing algorithm

```

Fitness = 0;
for( each training problem ) {
    for( each item in the problem, in order ) {
        rating = negativeInfinity;
        for( each container ) {
            value = heuristic( item, container );
            if( value > rating ) {
                rating = value;
                bestContainer = container;
            }
        }
        pack item in bestContainer;
    }
    Fitness += solutionQuality( problem );
}

```

Fig. 20.5 Applying a candidate rating heuristic

with a chosen one of its nearest labeled points, a very slightly modified form of this algorithm could be applied to any new problem without any search being involved whatever.

A somewhat similar messy GA approach was used by [Terashima-Marín et al. \(2010\)](#) to tackle two-dimensional stock-cutting problems involving both regular and irregular shapes. Stock-cutting problems are simply packing problems involving the cutting of specified shapes out of standardized stock material; practical examples include tasks such as stamping car body parts out of sheet steel. Each block in the chromosome contained a vector identifying a point in an eight-dimensional simplified state space, and also two heuristics—one to select the next shape to be cut and one to decide whereabouts on the stock sheet it should be cut from.

[Burke et al. \(2007a\)](#) tackled the online version of the one-dimensional packing problem by using genetic programming to evolve a rating heuristic to decide where best to put the next item. An adequately large array of containers was used, but only non-empty ones counted at the end. Any candidate heuristic was evaluated by using it to tackle a number of training problems; each problem was handled as shown in Fig. 20.5.

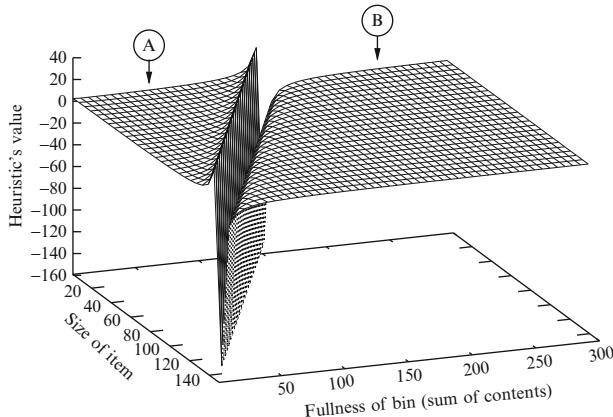


Fig. 20.6 A plot of the generated rating function for online packing

Unusually, candidate heuristics were allowed to overfill containers, but any such packing was penalized heavily so that the system learns not to overfill. The genetic programming used the standard $+, -, \times, /, \leq$ functions, where “ $/$ ” is protected division that returns 1 if the denominator is zero and “ \leq ” returns 1 or -1 . The terminals included the size of the item S , the fullness of the container F (that is, the sum of all that it currently holds) and the capacity C of the container. The `solutionQuality` function was

$$\text{solutionQuality} = \begin{cases} 1 - \frac{\sum(F_i/C)^2}{n} & : \text{if legal} \\ \text{big constant} & : \text{if not,} \end{cases}$$

where F_i is the fullness of container i and there are n non-empty containers. The approach was developed using a number of modest-sized benchmark packing problems, all using containers with size $C = 150$. Burke et al. suggested that a heuristic developed using a specific subset of the problems tended to show rather less good performance on other subsets with different characteristics, and gave the following example of a generated rating function:

$$\frac{2S + F}{S + F} + \frac{C}{((\frac{F}{C}) \leq 2C - F) + (C - S - F)}.$$

Figure 20.6 shows the value of this rating function for values of F from 1 to 300 (since the container could be overfilled beyond its capacity of 150 by a bad heuristic) and values of S from 1 to 150. The peak of the ridge lies along the line $S + F = C$. Either by careful analysis of the formula, or by inspecting a large table of the formula’s values, it can be seen that the region marked A slopes smoothly upward from the origin to the ridge, and the region marked B all lies strictly below any point in region A. This means that the heuristic rating function will never allow a container to be overfilled, because points with $S + F > C$ all have lower values than

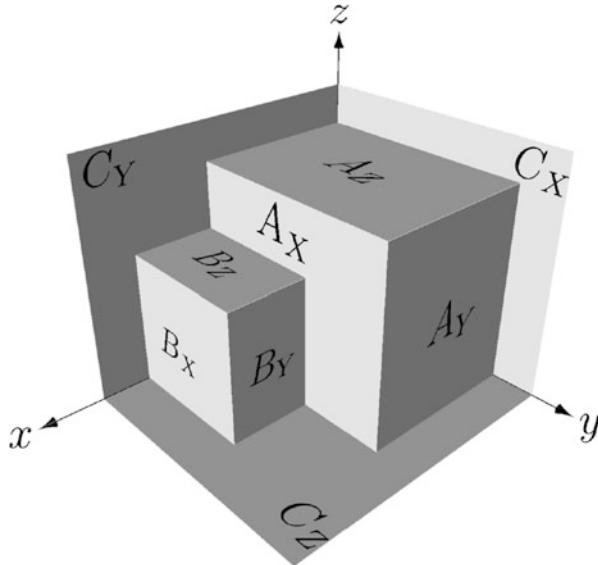


Fig. 20.7 Three-dimensional packing

any points satisfying $S + F \leq C$. Assuming that the items are never larger than the container, the rating function is exactly equivalent to the sensible online heuristic that says “put the item in the fullest container that can accept it”.

Allen et al. (2009) have extended this idea to three-dimensional packing of rectangular objects in rectangular containers. A more complicated notion of packing is used. Figure 20.7 shows a view of a container with two items already packed. There are five places where the next object might be placed, namely the five places where three back-surfaces meet. Items should be fully supported from below rather than overlapping the lower items: an item placed on surface A_z should not overhang B_z or C_z . If nothing can be used to infill a gap, such as the gap to the left of B_x , then the gap is deemed to be filled and the supporting surface B_z is deemed to expand over the gap. However, the basic notion, of trying every item in every possible corner and in every possible distinctive orientation, remains the same. The functions used in the genetic programming are more elaborate than in the one-dimensional case, and are not repeated here.

20.5.3 Remarks

See Burke et al. (2009b) for a survey of genetic programming applied in hyper-heuristics research. Arguably, there are numerous papers which have used genetic programming to create a good rating function and yet which do not mention hyper-heuristics. For example, Hauptman and Sipper (2005) used genetic programming to

create a good board evaluation function for certain kinds of chess end-game, namely those in which either side has a queen or a rook or both. A large number of chess-specific terminals were used, such as “is my king protecting one of my pieces?”, “number of legal moves for opponents king” and “distance of my king from edge of board” as well as basic logical operations such as and/if/or/not. Each function was evaluated by having it play against some other members of the population with a randomly-generated initial end-game position. The final result was able to draw against a world-class (2004) chess program. Hyper-heuristics is still a fairly new topic; perhaps in future more such examples will come to be known as instances of the topic.

However, if you want to explore hyper-heuristics, packing is a better place to start. There are many benchmark problems available, with many results available for comparison. And as yet, not many authors have tried using hybrid hyper-heuristics for online problems—that is, not only deciding where to put the next item but also revisiting earlier decisions from time to time so as to shuffle various items between containers. After all, that is what human supermarket bag-packers often do, while trying to pack bags fast and avoid undue delay at the end.

In earlier work, [Ross et al. \(2002\)](#) also tried using an extended classifier system as the search engine for one-dimensional packing. Classifier systems try to discover a good set of condition-action rules by simulated evolution, but early systems tended strongly to favor rules that were very general. XCS, the extended classifier system ([Wilson 1995](#)), avoids that trap by focusing more on the prediction accuracy of a rule than on the number of test cases it covered, and has been extremely successful in areas such as data mining. See also [Marin-Blazquez and Schulenburg \(2007\)](#).

20.6 A Little History

Although hyper-heuristics has only emerged as a recognizable topic area within the last decade or so, the basic concepts are certainly older than that. For example, the COMPOSER system ([Gratch et al. 1993](#)) planned communication schedules between Earth-orbiting satellites and ground stations, with a maximum interval between communications with a given satellite. The ground stations that could be used were constrained by satellite orbits. The scheduler used heuristic methods to try to build a schedule, deciding which unsatisfied constraints to focus on next and how to try to satisfy them. Because there were several possible heuristics to use in each case, the system used a simple hill-climbing approach to investigate combinations of them, testing each on 50 different problems, and was able to discover an effective combination. [Wah et al. \(1995\)](#) developed an early heuristics-learning system named *Teacher* and discussed a number of the issues raised in this chapter.

The Adaptive Constraint Engine (ACE, [Epstein et al. 2002](#)) also implemented some interesting ideas that are now being explored in hyper-heuristics research. ACE used reinforcement learning to try to discover good heuristics for constraint satisfaction problems, using an *advisor-based* architecture named FORR, which was

an acronym for “FOr the Right Reasons”. Each of a number of advisor components looked after some specific principle, such as “prefer the variable with the fewest remaining values in its domain”, and generated explicit comments about the suitability of its principle for the legal actions in any given state. ACE then learned what to do in each state, based on such comments. ACE built upon some of the ideas embodied in cognitive architectures such as SOAR ([SOAR 2012](#)) and ACT-R ([ACT-R 2012](#)).

20.7 Some Research Issues

20.7.1 No Free Lunch?

It would be natural to wonder just how good any generated heuristic could be. Perhaps some hyper-heuristic composition of other heuristics will manage to avoid the specific weaknesses of each of the ingredients, but might the composition not have new weaknesses of its own? Or do hyper-heuristics offer the prospect of some kind of “free lunch” to operations researchers?

Consider a finite-search problem defined on a finite domain. This is not a restrictive supposition; in fact, virtually all computer-based searching is done on such problems, because only finitely many values can be represented in IEEE floating point arithmetic, or in other formats within a finite-memory computer. [Wolpert \(1995\)](#) proved a famous “No Free Lunch” theorem that said that, averaged over all such problems, all search algorithms that do not revisit already-visited points have exactly the same average performance. This result is not as surprising as it might seem. Nearly all problems have no exploitable structure that could be used to guide a search—they can only be defined by a large lookup table, and nothing about the values would tell you where the optima actually lie. You can easily generate examples of intractable search problems for yourself. For example, start with $f(x) = x^2$ in the interval $x \in 0\dots 100$ and then, in your program, redefine $f(\pi^2 + 23)$ to be 10,000,000 instead of $\approx 1,080.410893$. Now the maximum of this modified function $f()$ lies at $x = (\pi^2 + 23)$ (or rather at the closest value to this that the computer can represent internally) rather than at $x = 100$, but how could you find it without sampling that one specific value? Nothing about the other values of the function will tell you that this amazing spike is lurking there. But in practice we are not interested in such random- or random-seeming functions; the problems we humans are interested in have some internal structure and some degree of predictability or continuity about them. We therefore want to find algorithms that can perform well on the subset of problems that arise naturally in some context; the difficulty lies in characterizing that subset properly in such a way that we can design a really effective algorithm for the members of that subset. Hyper-heuristics sidesteps these issues of characterization and design to some degree by instead conducting a search for an adequately effective algorithm.

Schumacher et al. (2001) showed that the No Free Lunch theorem applies to smaller sets of problems than *all* problems defined on a finite domain, and in particular it applies to a set of problems that is closed under permutation of the set of values. To get an idea of what that means, consider the domain $x \in [1, 2, 3, 4, 5]$ and the set of values [23, 47, 51, 55, 93]. There are 120 different ways of defining a function $f()$ on that domain by assigning these values to $f(1) \dots f(5)$, thus producing a set of problem (functions) closed under permutation of the values. But more significantly, Schumacher et al. showed that the No Free Lunch theorem only applies to such problem sets; if the problem set is not closed under permutation, then the theorem is not true and there are some algorithms that have better average performance than others. As Poli and Graff (2009) have pointed out, this means that hyper-heuristics-generated algorithms may turn out to be better than average if the problem set to which they are applied is not closed under permutation.

In practice, this is still more of theoretical than practical interest, but it does highlight the research question of how best to characterize the set of problems on which a generated heuristic is capable of showing particularly good performance. There has been some practical work done on such issues already—see Sect. 20.5.2—but much more could be done. One possible approach is to investigate systematically how the performance of a generated heuristic varies as it is applied to other sets of problems in the same area but with different characteristics. Another approach might be to use search techniques to try to create problems that cause the heuristic to stumble in some way.

20.7.2 Search Methods

Many different search techniques have been tried already, such as genetic programming, tabu search, straightforward genetic algorithms, variable-neighborhood search, simulated annealing, classifier systems, ant colony methods, particle swarm methods and so on.

Genetic programming is typically used to create a rating function of some kind but the general technique still suffers from some weaknesses that researchers are trying to address, such as the issue of *bloat*: the creation of overly-elaborate functions that may be unreasonably opaque to the end users or which might bias the search undesirably. As illustrated earlier, sometimes a generated function is relatively simple and can be analyzed to simplify it even further. But it is less easy to analyze a function tree of the kind shown in Fig. 20.2 in order to try to simplify it. Various kinds of tree-pruning methods used in data mining could be used to explore whether a generated heuristic is either overly complicated or perhaps even too specific to the training examples.

The *messy GA* and classifier systems approaches mentioned in Sects. 20.5.2 and 20.5.3 typically produce a rather different sort of output, which directly associates a choice of existing heuristic with some problem-specific conditions. Few people have yet done much investigation of these search methods in the context of

hyper-heuristics—see [Terashima-Marín et al. \(2008\)](#) for an example that tackles hard constraint satisfaction problems and see [Terashima-Marín et al. \(2010\)](#) for an example that tackles regular- and irregular-shaped stock-cutting problems. Essentially these methods generate sets of condition-action rules, but the actions do not have to be simply about choosing a heuristic. In a hybrid system, the actions might also be about when to switch from a constructive phase to a solution-perturbing phase or vice versa, or might be about altering the scope of a search process, and so on.

The example described at the start of Sect. 20.3.2 searched a space defined by a grammar, but the grammar in question was essentially defined by the type of information involved. Grammatical evolution ([O'Neill 2003, 2012](#)) could be used to explore more varied kinds of grammar, although there are open questions about how best to conduct the search through a grammar-defined space ([Castle and Johnson 2010](#)).

As yet, little has been done on distributed and parallel versions of hyper-heuristic search, but see [Biazzini et al. \(2009\)](#) and [León et al. \(2009\)](#) for examples.

20.7.3 Representation Issues

Various researchers have begun to explore how the choice of representation affects the results, for instance by studying which base-level heuristics to include or omit ([Burke et al. 2007b](#)) or which problem-features to include ([Ross et al. 2003](#)). In many instances the choice of representation has been guided by a study of existing heuristics, although the use of sensitivity analysis to investigate what really matters is still relatively rare. It is also conceivable that the choice of representation could itself be heuristically guided and problem specific. Think of trying to solve a rectangular jigsaw puzzle. It is commonplace to start with the four corner pieces and then the edge pieces. Thereafter, the choice of what to focus on tends to be guided by the specific details of the puzzle itself. For instance, if the puzzle contains lots of relatively undifferentiated blue sky then that might best be left until most of the rest of the puzzle has been completed, because then there will be more guidance available from the surrounding pieces. The same sort of notion might be applied in hyper-heuristics.

20.7.4 Performance Guarantees

Although real-world users of heuristics often like to have performance guarantees, little has as yet been done to try to generate heuristics that have formally provable performance bounds, and indeed relatively few human-generated heuristics seem to offer such guarantees either. Section 20.5.1 mentioned one example: the “largest first” heuristic that uses no more than $11M/9 + 4$ containers where M is the minimum.

One possible way to make progress in this area might be to try combining hyper-heuristics ideas with ideas from the study of *parameterized algorithms*, sometimes also referred to as *fixed-parameter algorithms* (Niedermeier 2006; Downey and Fellows 1999). The key idea in fixed-parameter algorithms is to introduce an additional, fixed parameter that will help to focus the search for a solution and will significantly simplify the process. Here is an example. Imagine that you have been asked to help modernize a large national railway system—see Weihe (1998) for a motivating real-world example. It is very costly to have a ticket office and ticket-checking barriers at every station; all you really need is to have such facilities at one end or the other of every possible journey. So you need to find the fewest possible number of stations at which to install such facilities; this is known as the *vertex cover problem* and it is known to be NP-hard. In graph theory terms, if a graph has N nodes, representing stations, and E edges, representing journeys, then the task is to find a minimal subset of nodes such that every edge meets at least one of them. Clearly the size of such a subset will be at most $\lfloor E/2 \rfloor$ but might well be much smaller, and even some kind of non-exhaustive search could be very costly. But if you introduce a new parameter k and search only for covers of size at most k , the search can be made much more focused. This is because any node that has $e > k$ edges attached to it must be in the cover; if it were not in the cover, then all of the e nodes at the other ends of those edges would have to be in the cover, thus making the cover larger than k in size. By studying the degree sequence—that is, the ordered sequence of the number of edges meeting each node—it is often possible to home in on a modest number of possible choices for k , each of which may involve a far less costly search than in the unparameterized case and which involve an explicit bound on the solution size.

It may be possible to unite hyper-heuristic ideas with specific ideas about parameterized algorithms in order to generate heuristics with specific performance characteristics. This also raises the possibility of using hyper-heuristic ideas to generate new kinds of metaheuristic algorithms which will be able to announce their performance characteristics when applied to any specific problem.

20.8 Getting Started: The HyFlex Framework

The best way to learn more about hyper-heuristics is by trying some practical experiments. The HyFlex software framework (Ochoa et al. 2012) is a convenient tool for the purpose, written in Java. At the time of writing it includes four kinds of problem:

- One-dimensional bin-packing problems;
- Satisfiability problems;
- Personnel scheduling problems, not unlike the timetabling problems described earlier;
- Permutation flow shop problems. These are work-scheduling problems in which there are n jobs to be completed and each job needs to visit m machines, in each case visiting machine 1 first, machine 2 second and so on (so that the work-flow

is the same for each job), but possibly visiting each machine for differing job-specific amounts of time. Therefore the issue is to find that permutation of the n jobs which minimizes the total time to completion.

In each case a number of benchmark problems is supplied, and also a number of heuristics. The heuristics are black boxes: they can be applied to a problem but their internal workings are not available at the level of the system's API. There are four kinds of heuristic: mutational/perturbing ones; ruin–recreate ones that make large-scale changes by partially destroying a solution and rebuilding it; local search ones that stop when a local optimum is found or a stopping condition is met; and crossover ones that construct a new solution in some way from two current ones.

The API provides hooks to initialize and manage a population of candidate solutions, set and monitor a time limit, and apply any of the relevant heuristics. Two parameters, described as *depth of search* and *intensity of mutation*, make it possible to control certain aspects of the heuristics' operations. The main task is therefore to create your own program that learns through experience which heuristic to apply, with what parameter settings.

The framework provides a simple and convenient entry-point into some of the practical aspects of hyper-heuristics. However, it enforces a strict separation between the domain-specific aspects, hidden from the end user behind a so-called domain barrier, and a domain-independent form of hyper-heuristic exploration. For large real-world applications, it is often desirable to use far more domain information than a tool such as HyFlex can conveniently offer.

20.9 Tricks of the Trade

Hyper-heuristics is an excellent area for research: there are still many issues awaiting thorough explorations. This section lists some advice.

20.9.1 The Problem Area

If you have a choice, look for an area where a hyper-heuristic approach is likely to bring useful benefits. Finding new ways to slingshot a spaceship around the inner planets may be fun and that research may have valuable theoretical consequences, but the space agencies are not necessarily interested in fast-and-cheap, good-enough algorithms at the mission-planning stage. On the other hand, there may be useful practical scope for hyper-heuristic approaches in dynamic robot control applications: a moving robot may not be able to afford to wait for the completion of some expensive optimization routine. Whatever the area, remember that you will need a significant number and variety of problem instances.

If your focus is on very practical problems, perhaps provided by a commercial partner, you may not have that many problem instances to work with. It is possible to

generate additional examples, both by using random methods to create completely new problems and by making random changes of various sizes in existing problems. In many areas of combinatorics, generating random problems by some naive method may create examples that are not particularly hard, so it can be worthwhile to put some thought into how to generate problems. It can also pay to do a systematic study to try to discover what makes certain problems especially hard, at least for certain heuristics or search methods. It is sometimes possible to create problems for which you know the optimal answer. For example, in two-dimensional packing, you can start by dissecting the shape to be packed into a number of pieces, and then change some pieces a little by removing small amounts, such that the total amount removed adds up to less than the smallest piece. Thus all the pieces will still be needed.

20.9.2 Success Criteria

You should not simply be trying to produce better results than any recently published ones. Hyper-heuristic methods are unlikely ever to beat hand-crafted, CPU-intensive problem-specific methods. Your new hyper-heuristic methods may of course produce better results than some earlier hyper-heuristic methods, but that should not be your final goal. As a scientist, your aim should be to develop a new and greater understanding of what is going on, and to tell others about what you learned. Rather than simply obtaining good results, try to explore what made the difference and also, what the limitations of your methods are. More specifically, if your interest is in developing good-enough, fast-enough methods for some class of problems, there is going to be some kind of trade-off between speed and quality. Despite many decades of research, that kind of Pareto frontier is still poorly charted territory. It can be helpful to think carefully at the outset about the success criteria for your research, not least because you then have a clear goal to work towards. For example, a good hyper-heuristic might be one that is significantly cheaper to use than the alternative of trying every available heuristic in turn, and also provides results that can be better than the best heuristic but never significantly worse. Once you have found a good hyper-heuristic, investigate what features contributed to its success, for example by seeing what happens if you omit individual ingredients. Remember Danth's Law, paraphrased here as "*if you are forced to resort to declaring victory, you have probably already lost*".

20.9.3 On-line or Off-line

Many papers discuss some class of off-line problems, in which all relevant information is available at the start, and others discuss on-line problems and assume that the task is to build a solution by deciding what to do with the latest information. Few authors have yet considered hybrid approaches in which, when new information arrives, a very limited time is spent on reconsidering the whole of the solution so

far and possibly reorganising it. This would make good use of the time between arrivals.

20.9.4 A Good Set of Heuristic Ingredients

Finding a good set of heuristics, or a set of heuristics components, can be tricky. Having too large a set can cripple the search process by making the search space too enormous. Look for heuristics that complement each other in some way. In solution-constructing approaches, remember that a single application of a heuristic does not have to do just one step of the construction: a heuristic can also be of the form *do ... until ...* or some other such looping construct. In solution-modifying approaches, it may be helpful to include “destructive” ingredients that may shift the focus to some other part of the space.

20.9.5 Fitness

Although the ultimate aim may be to find some algorithm that is capable of performing well on a wide variety of problems, it is not always necessary to evaluate each candidate on every problem. The computational burden can be reduced by evaluating each candidate on some randomly selected subset of problems, biasing the selection in favour of those problems that have participated least in the evaluations so far, and perhaps also in favour of those that have been hard.

Of course, you should adopt good practices such as having separate sets of problems for training and for eventual validation, or use cross-validation. It is also good practice to keep a set of problems apart for use as a final test set, that ideally should only be used once: you should try to avoid “data snooping”, sometimes referred to as “data dredging”, as far as you can.

20.9.6 A Good Set of Tools

There are many tool-kits available that implement search methods such as genetic algorithms, genetic programming or tabu search. It takes time to learn to use such tool-kits properly, and you will probably still need to do some programming work to add the features that you need, such as specific heuristics. It can often be worthwhile to build your own system rather than relying on a toolkit: that way, you get only what you need, your system does what you want, and you learn more than you would if you relied on someone else’s ideas and skills. Aim to become a good programmer: it takes time, but is a very valuable talent.

Resist any temptation to build your own very elaborate graphical user interface (GUI). GUIs are limited when it comes to handling and analyzing large amounts of

data (and GUI-building is often simply a form of work-avoidance). Hyper-heuristic research often generates very large amounts of data, that are better handled using a powerful scripting language such as *Perl* or *Python*. Open-source tools such as *gnuplot* or the python-specific *matplotlib* are excellent for graph plotting, and *graphviz* is excellent for tree or network visualization; becoming thoroughly comfortable with such tools does take time but pays big dividends.

20.9.7 Attitude

Try to be skeptical about your results. Subtle programming errors, whether made by you or by someone else, can affect your results, so try to find ways to check them that do not depend on reusing critical parts of your code.

Aim to keep up to date with related research, such as developments in search technology as well as more specific things such as new ideas in your chosen problem areas. Reviewers routinely reject papers that claim good results on the basis of a comparison with outdated results or with obsolete algorithms. Try to be your own strongest critic.

Sources of Additional Information

This section lists some places to look.

- The *Handbook of Metaheuristics* ([Glover and Kochenberger 2003](#)) contains a lot of information about different kinds of heuristics, and includes a chapter about hyper-heuristics ([Burke et al. 2003](#)).
- The ASAP group at Nottingham University has a good website ([ASAP 2012](#)) that includes research publications about hyper-heuristics and links to timetabling problems and other resources.
- The *Journal of Heuristics*, published by Kluwer, contains many papers about heuristic methods generally. The tables of contents and the abstracts of papers are available online; full papers are available to subscribers to Kluwer Online.
- The *European Journal of Operational Research* also contains many papers relating to heuristics and to problems that might be tackled by hyper-heuristic methods. Again, abstracts are freely available online.
- The Metaheuristics Network site at www.metaheuristics.org provides information about various meta-heuristic techniques, references to papers and links to sets of problems in several areas: quadratic assignment, maximum-satisfiability, timetabling, scheduling, vehicle routing and an industrial hose-optimization problem. The aim of the Metaheuristics Network is to conduct scientific comparisons of performance between various metaheuristic techniques in different problem areas. Although hyper-heuristic methods are not explicitly considered, the site is

valuable because the problems have been generated or contributed by the members and performance results are being made available.

- The *OR-Library* ([OR-library 2012](#)) is a large repository of benchmark OR problems.
- [Kendall \(2012\)](#) has a useful online bibliography of papers about hyper-heuristics.
- The European Space Agency ([ESA 2012](#)) has some difficult spacecraft trajectory optimization problems available online.
- There is an online collection of frequency assignment problems ([FAP 2012](#)).
- See [University of Melbourne data \(2012\)](#) for some real-world university exam timetabling problems.

References

- Allen S, Burke EK, Hyde M, Kendall G (2009) Evolving reusable 3D packing heuristics with genetic programming. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 931–938
- ASAP research group (2012) <http://www.asap.cs.nott.ac.uk/>
- Bader-El-Den MB, Poli R (2008) Generating SAT local-search heuristics using a GP framework. In: Monmarche N et al (eds) Artificial Evolution: Proceedings of the 8th International Conference EA 2007, Tours, France. Springer LNCS 4926, 37–49, 2008
- Bettinger P, Sessions J, Boston K (2009) A review of the status and use of validation procedures for heuristics used in forest planning. Int J Math Comput Forest Nat Resour Sci 1:26–37. <http://mcfns.com>
- Biazzini M, Bánhegyi B, Montresor A, Jelasity M (2009) Distributed hyper-heuristics for real parameter optimization. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 1339–1346
- Bittle S, Fox M (2009) Learning and using hyper-heuristics for variable and value ordering in constraint satisfaction problems. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 2209–2212
- Bräysy O (2003) A reactive variable neighborhood search for the vehicle routing problem with time windows. INFORMS J Comput 15:347–368
- Burke E, Hart E, Kendall G, Newall J, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. In: Glover F, Kochenberger G (eds) Handbook of meta-heuristics. Kluwer, Dordrecht, pp 457–474
- Burke EK, Hyde M, Kendall G, Woodward J (2007a) Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In: Proceedings of the GECCO 2007, London. ACM, New York, pp 1559–1565
- Burke EK, McCollum B, Meisels A, Petrovic S, Qu R (2007b) A graph-based hyper-heuristic for educational timetabling problems. Eur J Oper Res 176:177–192

- Burke EK, Hyde M, Kendall G, Ochoa G, Ozcan E, Woodward J (2009a) A classification of hyper-heuristics approaches. In: Gendreau M, Potvin J-Y (eds) *Handbook of metaheuristics*, 2nd edn. Springer, Berlin, pp 449–468
- Burke EK, Hyde MR, Kendall G, Ochoa G, Ozcan E, Woodward JR (2009b) Exploring hyper-heuristic methodologies with genetic programming. In: Mumford CL, Jain LC (eds) *Computational intelligence: collaboration, fusion and emergence*. Springer, Berlin, pp 177–201
- Castle T, Johnson CG (2010) Positional effect of crossover and mutation in grammatical evolution. In: Esparcia-Alcazar AI et al (eds) *Proceedings of the EuroGP 2010*, Istanbul. LNCS 6021. Springer, Berlin, pp 26–37
- Chakhlevitch K, Cowling PI (2008) Hyperheuristics: recent developments. In: Cotta C, Sevaux M, Sørensen K (eds) *Adaptive and multilevel metaheuristics. Studies in computational intelligence*, 136. Springer, Berlin, pp 3–29
- Coffman EG, Garey MR, Johnson DS (1996) Approximation algorithms for bin packing: a survey. In: Hochbaum D (ed) *Approximation algorithms for NP-hard problems*. PWS, Boston, pp 46–93
- Cowling PI, Chakhlevitch K (2007) Using a large set of low-level heuristics in a hyperheuristic approach to personnel scheduling. In: Dahal KP, Tan KC, Cowling PI (eds) *Evolutionary scheduling. Studies in computational intelligence*, 49. Springer, Berlin, pp 543–576
- Cowling P, Kendall G, Soubeiga E (2001) A hyperheuristic approach for scheduling a sales summit. In: PATAT 2000, Konstanz. LNCS 2079. Springer, Berlin, pp 176–190
- Downey RG, Fellows MR (1999) *Parameterized complexity*. Springer, New York
- Dyckhoff H (1990) A topology of cutting and packing problems. *Eur J Oper Res* 44:145–159
- Epstein SL, Freuder EC, Wallace RJ, Morozov A, Samuels B (2002) The adaptive constraint engine. In: Van Hentenryck P (ed) *Principles and Practice of Constraint Programming – CP 2002*, Ithaca. LNCS 2470. Springer, Berlin, pp 525–540
- ESA (2012) Global trajectory optimisation problems. C++ and Matlab code available. <http://www.esa.int/gsp/ACT/inf/op/globopt.htm>
- Frequency assignment problems (2012) fap.zib.de/
- Fukunaga A (2002) Automated discovery of composite SAT variable-selection heuristics. In: *Proceedings of the AAAI 2002*, AAAI Press, Edmonton, pp 641–648
- Fukunaga AS (2008) Automated discovery of local search heuristics for satisfiability testing. *Evol Comput* 16:31–61
- Garrido P, Castro C (2009) Stable solving of CVRPs using hyperheuristics. In: *Proceedings of the GECCO 2009*, Montreal, pp 255–262
- Glover F, Kochenberger G (eds) (2003) *Handbook of meta-heuristics*. Kluwer, Dordrecht
- Goldberg DE, Deb K, Kargupta H, Harik G (1989) Messy genetic algorithms: motivation, analysis and first results. *Complex Syst* 3:493–530

- Gratch J, Chein S, de Jong G (1993) Learning search control knowledge for deep space network scheduling. In: Proceedings of 10th international conference on machine learning, Amherst, pp 135–142
- Hauptman A, Sipper M (2005) GP-endchess: using genetic programming to evolve chess endgame players. In: Keijzer M et al (eds) Proceedings of the 8th EuroGP, Lausanne. LNCS 3447. Springer, Berlin, pp 120–131
- Johnson DS (1973) Near-optimal bin-packing algorithms. PhD thesis, MIT Department of Mathematics
- Kendall G (2012) A bibliography of hyper-heuristics and related approaches. <http://www.cs.nott.ac.uk/~gxo/hhbibliography.html>
- Kubale M (ed) (2004) Graph colorings. AMS, Providence
- León C, Miranda G, Segura C (2009) A memetic algorithm and a parallel hyper-heuristic island-based model for a 2D packing problem. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 1371–1378
- Marin-Blazquez JG, Schulenburg S (2007) A hyper-heuristic framework with XCS: learning to create novel problem-solving algorithms constructed from simpler algorithmic ingredients. In: Learning classifier systems. LNCS 4399. Springer, Berlin, pp 193–218
- Marques-Silva J (2008) Practical applications of Boolean satisfiability. In: Workshop on discrete event systems, Gothenberg, pp 74–80
- Martello S, Toth P (1990) Knapsack problems. Algorithms and computer implementations. Wiley, New York
- Neth H et al. (2009) Analysis of human search strategies. Technical report, Large Knowledge Collider Consortium. Deliverable 4.2.2. www.larkc.eu
- Niedermeier R (2006) Invitation to fixed-parameter algorithms. Oxford lecture series in mathematics and its applications, 31. Oxford University Press, Oxford/New York
- Ochoa G, Qu R, Burke EK (2009) Analyzing the landscape of a graph based hyper-heuristic for timetabling problems. In: Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 341–348
- Ochoa G, Hyde M, Curtois T, Vazquez-Rodriguez JA, Walker J, Gendreau M, Kendall G, McCollum B, Parkes AJ, Petrovic S, Burke EK (2012) HyFlex: A benchmark framework for cross-domain heuristic search. In: Hao J-K, Middendorf M (eds), European conference on evolutionary computation in combinatorial optimisation EvoCOP 2012. LNCS 7245. Springer, Berlin, pp 136–147
- O'Neill M, Ryan C (2003) Grammatical evolution: evolutionary automatic programming in an arbitrary language. Springer, Berlin
- O'Neill M (2012) The grammatical evolution page. <http://www.grammatical-evolution.org>
- OR-library (2012) <http://people.brunel.ac.uk/~mastjeb/jeb/orlib/>
- Poli R, Graff M (2009) There is a free lunch for hyper-heuristics, genetic programming, and computer scientists. In: Vanneschi L et al (eds) Genetic programming. Proceedings of the 12th EuroGP, Tübingen. LNCS 5481. Springer, Berlin, pp 195–207

- Poli R, Langdon WB, McPhee N (2008) A field guide to genetic programming. Lulu, Raleigh. Also available as a free PDF from lulu.com
- Qu R, Burke EK (2009) Hybridisations within a graph based hyper-heuristic framework for university timetabling problems. *J Oper Res Soc* 60:1273–1285
- Ross P, Schulenburg S, Marín-Blázquez JG, Hart E (2002) Hyper-heuristics: learning to combine simple heuristics in bin packing problems. In: Langdon WB et al. (eds) *Proceedings of the GECCO 2002*, New York. Morgan Kaufman, San Mateo, pp 942–948
- Ross P, Marín-Blázquez JG, Schulenburg S, Hart E (2003) Learning a procedure that can solve hard bin-packing problems: a new GA-based approach to hyper-heuristics. In: Cantú-Paz E et al (eds) *Proceedings of the GECCO 2003*, Chicago. LNCS 2724. Springer, Berlin, pp 1295–1306
- Satlib—the satisfiability library (2012) <http://www.satlib.org>
- Schumacher C, Vose MD, Whitley LD (2001) The no free lunch and problem description length. In: *Proceedings of the GECCO 2001*, San Francisco. Morgan Kaufman, San Mateo, pp 565–570
- The ACT-R home page (2012) <http://act-r.psy.cmu.edu/>
- The SOAR home page (2012) <http://sitemaker.umich.edu/soar/home>
- Terashima-Marín H, Ortiz-Bayliss JC, Ross P, Valenzuela-Rendón M (2008) Using hyper-heuristics for the dynamic variable ordering in hard constraint satisfaction problems. In: *Proceedings of the MICAI 2008*, Atizapán de Zaragoza. LNCS 5317. Springer, Berlin, pp 407–417
- Terashima-Marín H, Ross P, Farías-Zárate CJ, López-Camacho E, Valenzuela-Rendón M (2010) Generalized hyper-heuristics for solving 2D regular and irregular packing problems. *Ann Oper Res* 179:369–392
- University of Melbourne (2012) <http://www.or.ms.unimelb.edu.au/timetabling/>. Exam timetabling data
- Wah BW, Ieumwananonthachai A, Chu LC, Aizawa A (1995) Genetics-based learning of new heuristics: rational scheduling of experiments and generalization. *IEEE Trans Knowl Data Eng* 7:763–785
- Weihe K (1998) Covering trains by stations or the power of data reduction. In: Battiti R, Bertossi AA (eds) *Proceedings of the ALEX 1998*, pp 1–8. <http://rtm.science.unin.it/alex98/book/weihe.ps.gz>
- Wilson SW (1995) Classifier systems based on accuracy. *Evol Comput* 3:149–175
- Wolpert D, MacReady WG (1995) No free lunch theorems for search. Technical report SFI-TR-92-02-010, Santa Fe Institute

Chapter 21

Approximations and Randomization

Carla P. Gomes and Ryan Williams

21.1 Introduction

Most interesting real-world optimization problems are very challenging from a computational point of view. In fact, quite often, finding an optimal or even a near-optimal solution to a large-scale optimization problem may require computational resources far beyond what is practically available. Computer scientists study the computational properties of optimization problems by considering how the computational demands of a solution method grow with the size of the problem instance to be solved. A key distinction is made between problems that require computational resources that grow polynomially with problem size versus those for which the required resources grow exponentially. The former category of problems are called efficiently solvable, whereas problems in the latter category are deemed *intractable* because the exponential growth in required computational resources renders all but the smallest instances of such problems unsolvable.

A large class of common optimization problems are classified as *NP-hard*. It is widely believed—though not yet proven (Clay Mathematics Institute 2003)—that NP-hard problems are intractable, which means that there does not exist an efficient algorithm (i.e. one that scales polynomially) that is guaranteed to find an optimal solution for such problems. Examples of NP-hard optimization tasks are the minimum traveling salesman problem (TSP), the minimum graph coloring problem, and the minimum bin packing problem. As a result of the nature of NP-hard problems, progress that leads to a better understanding of the structure, computational properties, and ways of solving one of them, *exactly* or *approximately*, also leads to better

C.P. Gomes
Department of Computer Science, Cornell University, Ithaca, NY, USA
e-mail: gomes@cs.cornell.edu

R. Williams (✉)
Computer Science Department, Stanford University, Stanford, CA, USA
e-mail: rww@cs.stanford.edu

algorithms for solving hundreds of other different but related NP-hard problems. Several thousand computational problems, in areas as diverse as economics, biology, operations research, computer-aided design and finance, have been shown to be NP-hard.

A natural question to ask is whether *approximate* (i.e. near-optimal) solutions can possibly be found efficiently for such hard optimization problems. Heuristic local search methods, such as simulated annealing and tabu search (see Chaps. 9 and 10), are often quite effective at finding near-optimal solutions. However, these methods do not come with rigorous guarantees concerning the quality of the final solution or the required maximum run-time. In this chapter, we will discuss a more theoretical approach to this issue consisting of so-called approximation algorithms, which are efficient algorithms that can be proven to produce solutions of a certain quality. We also discuss classes of problems for which no such efficient approximation algorithms exist, thus leaving an important role for the quite general, heuristic local search methods.

The design of good approximation algorithms is a very active area of research where one continues to find new methods and techniques. It is quite likely that these techniques will become of increasing importance in tackling large real-world optimization problems.

In the late 1960s and early 1970s a precise notion of approximation was proposed in the context of multiprocessor scheduling and bin packing (Graham 1966; Garey et al. 1972; Johnson 1974). Approximation algorithms generally have two properties. First, they provide a feasible solution to a problem instance in polynomial time. In most cases, it is not difficult to devise a procedure that finds *some* feasible solution. However, we are interested in having some assured quality of the solution, which is the second aspect characterizing approximation algorithms. The quality of an approximation algorithm is the maximum *distance* between its solutions and the optimal solutions, evaluated over all the possible instances of the problem. Informally, an algorithm approximately solves an optimization problem if it always returns a feasible solution whose measure is close to optimal, for example within a factor bounded by a constant or by a slowly growing function of the input size. Given a constant α , an algorithm \mathcal{A} is an α -approximation algorithm for a given minimization problem Π if its solution is at most α times the optimum, considering all the possible instances of problem Π .

The focus of this chapter is on the design of approximation algorithms for NP-hard optimization problems. We will show how standard algorithm design techniques such as greedy and local search methods, dynamic programming, and classical methods of discrete optimization such as linear programming and semidefinite programming have been used to devise good approximation algorithms.

We will also show how randomization is a powerful tool for designing approximation algorithms. Randomized algorithms are interesting because in general such approaches are easier to analyze and implement, and faster than deterministic algorithms (Motwani and Raghavan 1995). A randomized algorithm is simply an algorithm that performs some of its choices randomly; it “flips a coin” to decide what to do at some stages. As a consequence of its random component, different executions of a randomized algorithm may result in different solutions and runtime,

even when considering the same instance of a problem. We will show how one can combine randomization with approximation techniques in order to efficiently approximate NP-hard optimization problems. In this case, the approximation solution, the approximation ratio and the runtime of the approximation algorithm may be random variables. Confronted with an optimization problem, the goal is to produce a randomized approximation algorithm with runtime provably bounded by a polynomial and whose feasible solution is close to the optimal solution, *in expectation*. Note that these guarantees hold for every instance of the problem being solved. The only randomness in the performance guarantee of the randomized approximation algorithm comes from the algorithm itself, and not from the instances.

Since we do not know of efficient algorithms to find optimal solutions for NP-hard problems, a central question is whether we can efficiently compute good approximations that are close to optimal. It would be very interesting (and practical) if one could go from exponential to polynomial time complexity by relaxing the constraint on optimality, especially if we guarantee at most a relatively small error.

Good approximation algorithms have been proposed for some key problems in combinatorial optimization. The so-called APX complexity class includes the problems that allow a polynomial-time approximation algorithm with a performance ratio bounded by a constant. For some problems, we can design even better approximation algorithms. More precisely we can consider a family of approximation algorithms that allows us to get as close to the optimum as we like, as long as we are willing to trade quality with time. This special family of algorithms is called an approximation scheme and the so-called PTAS class is the class of optimization problems that allow for a *polynomial time approximation scheme* that scales polynomially in the size of the input. In some cases we can devise approximation schemes that scale polynomially, both in the size of the input and in the magnitude of the approximation error. We refer to the class of problems that allow such *fully polynomial time approximation schemes* as FPTAS.

Nevertheless, for some NP-hard problems, the approximations that have been obtained so far are quite poor, and in some cases no one has ever been able to devise approximation algorithms within a constant factor of the optimum. Initially it was not clear if these weak results were due to our lack of ability in devising good approximation algorithms for such problems or to some inherent structural property of the problems that excludes them from having good approximations. We will see that indeed there are limitations to approximation, *intrinsic* to some classes of problems. For example, in some cases there is a lower bound on the constant factor of the approximation, and in other cases we can provably show that there are no approximations within *any* constant factor from the optimum. Essentially, there is a wide range of scenarios going from NP-hard optimization problems that allow approximations to *any* required degree, to problems not allowing approximations at all. We will provide a brief introduction to proof techniques used to derive non-approximability results.

We believe that the best way to understand the ideas behind approximation and randomization is to study instances of algorithms with these properties, through examples. Thus in each section, we will first introduce the intuitive concept, then

reinforce its salient points through well-chosen examples of prototypical problems. Our goal is far from trying to provide a comprehensive survey of approximation algorithms or even the best approximation algorithms for the problems introduced. Instead, we describe different design and evaluation techniques for approximation and randomized algorithms, using clear examples that allow for relatively simple and intuitive explanations. For some problems discussed in the chapter there are approximations with better performance guarantees but requiring more sophisticated proof techniques that are beyond the scope of this introductory tutorial. In such cases we will point the reader to the relevant literature results. In summary, our goals for this chapter are as follows:

1. Present the fundamental ideas and concepts underlying the notion of approximation algorithms.
2. Provide clear examples that illustrate different techniques for the design and evaluation of efficient approximation algorithms. The examples include accessible proofs of the approximation bounds.
3. Introduce the reader to the classification of optimization problems according to their polynomial-time approximability, including basic ideas on polynomial-time inapproximability.
4. Show the power of randomization for the design of approximation algorithms that are in general faster and easier to analyze and implement than the deterministic counterparts.
5. Show how we can use a randomized approximation algorithm as a heuristic to guide a complete search method (empirical results).
6. Present promising application areas for approximation and randomized algorithms.
7. Provide additional sources of information on approximation and randomization methods.

In Sect. 21.2 we introduce precise notions and concepts used in approximation algorithms. In this section we describe key design techniques for approximation algorithms. We use clear prototypical examples to illustrate the main techniques and concepts, such as the minimum vertex cover, the knapsack problem, the maximum satisfiability problem, the TSP, and the maximum cut problem. As mentioned earlier, we are not interested in providing the best approximation algorithms for these problems, but rather in illustrating how standard algorithm techniques can be used effectively to design and evaluate approximation algorithms. In Sect. 21.3 we provide a tour of the main approximation classes, including a brief introduction to techniques to proof lower bounds on approximability. In Sect. 21.4 we describe some promising areas of application of approximation algorithms. Section 21.5 summarizes the chapter and we conclude by suggesting additional sources of information on approximation and randomization methods.

21.2 Approximation Strategies

21.2.1 Preliminaries

21.2.1.1 Optimization Problems

We will define optimization problems in a traditional way (Aho et al. 1979; Ausiello et al. 1999). Each optimization problem has three defining features: the structure of the input instance, the criterion of a feasible solution to the problem, and the measure function used to determine which feasible solutions are considered to be optimal. It will be evident from the problem name whether we desire a feasible solution with a minimum or maximum measure. To illustrate, the minimum vertex cover problem may be defined in the following way:

Minimum Vertex Cover

Instance: An undirected graph $G = (V, E)$. *Solution:* A subset $S \subseteq V$ such that for every $\{u, v\} \in E$, either $u \in S$ or $v \in S$. *Measure:* $|S|$. We use the following notation for items related to an instance I .

$Sol(I)$ is the set of feasible solutions to I ,

$m_I : Sol(I) \rightarrow R$ is the measure function associated with I , and

$Opt(I) \subseteq Sol(I)$ is the feasible solutions with optimal measure (be it minimum or maximum).

Hence, we may completely specify an optimization problem Π by giving a set of tuples $\{(I, Sol(I), m_I, Opt(I))\}$ over all possible instances I . It is important to keep in mind that $Sol(I)$ and I may be over completely different domains. In the above example, the set of I is all undirected graphs, while $Sol(I)$ is all possible subsets of vertices in a graph.

21.2.1.2 Approximation and Performance

Roughly speaking, an algorithm approximately solves an optimization problem if it always returns a feasible solution whose measure is close to optimal. This intuition is made precise below.

Let Π be an optimization problem. We say that an algorithm A feasibly solves Π if given an instance $I \in \Pi$, $A(I) \in Sol(I)$; that is, A returns a feasible solution to I .

Let A feasibly solve Π . Then we define the *approximation ratio* $\alpha(A)$ of A to be the minimum possible ratio between the measure of $A(I)$ and the measure of an optimal solution. Formally,

$$\alpha(A) = \min_{(I \in \Pi)} \frac{m_I(A(I))}{m_I(Opt(I))}.$$

For minimization problems, this ratio is always at least 1. Respectively, for maximization problems, it is always at most 1.

21.2.1.3 Complexity Background

We define a decision problem as an optimization problem in which the measure is 0–1 valued. That is, solving an instance I of a decision problem corresponds to answering a yes/no question about I . Note we may also represent a decision problem as a subset S of the set of all possible instances: members of S represent instances where the measure is 1.

Informally, P (polynomial time) is defined as the class of decision problems Π for which there exists a corresponding algorithm A_Π , such that every instance $I \in \Pi$, is solved by A_Π within a polynomial ($|I|k$ for some constant k) number of steps on any *reasonable* model of computation. Reasonable models include single-tape and multi-tape Turing machines, random access machines, pointer machines, etc.

NP (non-deterministic polynomial time) is defined as the class of decision problems Π for which there exists a corresponding decision problem Π' in P and constant k satisfying

$$I \in \Pi \text{ if and only if there exists } C \in \{0, 1\}^{|I|k} \text{ such that } (I, C) \in \Pi'.$$

In other words, we can determine if I is in an NP problem efficiently if, given an instance I , one is also provided with a short “proof” C , which is of length polynomial in I . Notice that while a short proof always exists if $I \in \Pi$, it need not be the case that short proofs exist for instances not in Π . Thus, while P problems are considered to be those which are *efficiently decidable*, NP problems are those considered to be *efficiently verifiable* via a short proof.

We will also consider the optimization counterparts to P and NP, which are PO and NPO, respectively. Informally, PO is the class of optimization problems where there exists a polynomial time algorithm that always returns an optimal solution to every instance of the problem, whereas NPO is the class of optimization problems where the measure function is polynomial time computable, and an algorithm can determine whether or not a possible solution is feasible in polynomial time.

Our focus here will be on approximating solutions to the *hardest* of NPO problems, those problems where the corresponding decision problem is NP-hard. Interestingly, some NPO problems of this type can be approximated very well, whereas others can hardly be approximated at all.

21.2.2 The Greedy Method

Greedy approximation algorithms are designed with a simple philosophy in mind: repeatedly make choices that get one closer and closer to a feasible solution for

the problem. These choices will be optimal according to an imperfect but easily computable heuristic. In particular, this heuristic tries to be as opportunistic as possible in the short run. (This is why such algorithms are called *greedy*—a better name might be *short-sighted*). For example, suppose my goal is to find the shortest walking path from my house to the theater. If I believed that the walk via Forbes Avenue is about the same length as the walk via Fifth Avenue, then if I am closer to Forbes than Fifth, it would be reasonable to walk towards Forbes and take that route.

Clearly, the success of this strategy depends on the correctness of my belief that the Forbes path is indeed just as good as the Fifth path. We will show that for some problems, choosing a solution according to an opportunistic, imperfect heuristic achieves a non-trivial approximation algorithm.

21.2.2.1 Greedy Vertex Cover

The minimum vertex cover problem was defined in the previous section. Variants on the problem come up in many areas of optimization research. A simple greedy algorithm is a 2-approximation to the problem, and no better approximation algorithms are known! In fact, it is widely believed that one cannot approximate minimum vertex cover better than $2 - \varepsilon$ for any $\varepsilon > 0$, unless $P = NP$ (see Khot and Regev 2003). The 2-approximation is as follows.

Greedy-VC: Initially, let S be an empty set. Choose an arbitrary edge $\{u, v\}$. Add u and v to S , and remove u and v from the graph. Repeat until no edges remain in the graph. Return S as the vertex cover.

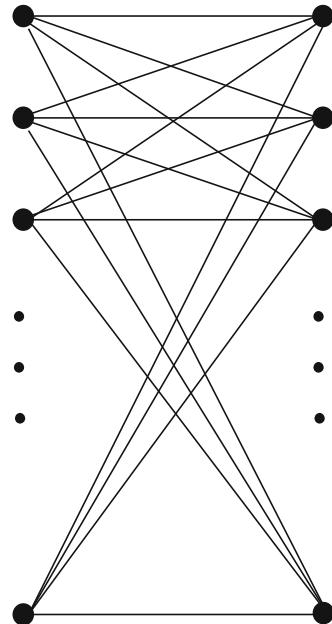
Theorem 21.1. *Greedy-VC is a 2-approximation algorithm for Minimum Vertex Cover.*

Proof. First, we claim S is indeed a vertex cover. Suppose not; then there exists an edge e which was not covered by any vertex in S . Since we only remove vertices from the graph that are in S , an edge e would remain in the graph after the algorithm had completed, which is a contradiction.

The next step is to show that any vertex cover of the graph contains at least $|S|/2$ vertices. $|S|/2$ is the number of edges we chose during the run of the algorithm, and none of them share any endpoints. Hence at least one vertex in any vertex cover of the given graph must be assigned to each edge we choose. It follows that any optimal solution has at least $|S|/2$ nodes, so our algorithm has a $|S|/|S^*| = 2$ approximation ratio. \square

Sometimes when one proves that an algorithm has a certain approximation ratio, the analysis is somewhat *loose*, and may not reflect the best possible ratio that can be derived. It turns out that Greedy-VC is no better than a 2-approximation. In particular, there is an infinite set of vertex cover instances where Greedy-VC provably chooses exactly twice the number of vertices necessary to cover the graph, namely in the case of complete bipartite graphs; see Fig. 21.1.

Fig. 21.1 Instances that correspond to bipartite graphs $K_{n,n}$. When running greedy-VC on these instances, the algorithm will select all $2n$ vertices



21.2.2.2 Greedy MAX-SAT

The MAX-SAT problem has been very well-studied; variants of it arise in many areas of discrete optimization. To introduce it requires a bit of terminology.

We will deal solely with Boolean variables (that is, those which are either true or false), which we will denote by x_1, x_2 , etc. A *literal* is defined as either a variable or the negation of a variable (e.g. $x_7, \neg x_{11}$ are literals). A clause is defined as the OR of some literals (e.g. $(\neg x_1 \vee x_7 \vee \neg x_{11})$ is a clause). We say that a Boolean formula is in conjunctive normal form (CNF) if it is presented as an AND of clauses (e.g. $(\neg x_1 \vee x_7 \vee \neg x_{11}) \wedge (x_5 \vee \neg x_2 \vee \neg x_3)$ is in CNF).

Finally, the MAX-SAT problem is to find an assignment to the variables of a Boolean formula in CNF such that the maximum number of clauses are set to true, or are *satisfied*. Formally:

MAX-SAT

Instance: A Boolean formula F in CNF.

Solution: An assignment a , which is a function from each of the variables in F to $\{\text{true}, \text{false}\}$.

Measure: The number of clauses in F that are set to true (are satisfied) when the variables in F are assigned according to a .

What might be a natural greedy strategy for approximately solving MAXSAT? One approach is to pick a variable that satisfies many clauses if it is set to a certain value. Intuitively, if a variable occurs negated in several clauses, setting the variable to *false* will satisfy several clauses; hence this strategy should approximately solve the problem well. Let $n(l_i, F)$ denote the number of clauses in F where the literal l_i appears:

Greedy-MAXSAT: Pick a literal l_i with maximum $n(l_i, F)$ value. Set its corresponding variable in such a way that all clauses containing it are satisfied, yielding a reduced F . Repeat until no variables remain in F .

It is easy to see that Greedy-MAXSAT runs in polynomial time (roughly quadratic time, depending on the computational model chosen for analysis). It is also a *good* approximation for the MAX-SAT problem.

Theorem 21.2. *Greedy-MAXSAT is a $1/2$ -approximation algorithm for MAXSAT.*

Proof. Proof by induction on the number of variables n in the formula F . Let m be the total number of clauses in F . If $n = 1$, the result is obvious. For $n > 1$, let l_i have maximum $n(l_i, F)$ value, and v_i be its corresponding variable. Let m_{POS} and m_{NEG} be the number of clauses in F that contain l_i and $\frac{1}{2}l_i$, respectively. After v_i is set so that l_i is true (so both l_i and $\frac{1}{2}l_i$ disappear from F), there are at least $m - m_{POS} - m_{NEG}$ clauses left, on $n - 1$ variables.

By induction hypothesis, Greedy-MAXSAT satisfies $(m - m_{POS} - m_{NEG})/2$ of these clauses at least, therefore the total number of clauses satisfied is at least $(m - m_{POS} - m_{NEG})/2 + m_{POS} = m/2 + (m_{POS} - m_{NEG})/2 = m/2$, by our greedy choice of picking the l_i that occurred most often. \square

21.2.2.3 Greedy MAX-CUT

Our next example shows how local search may be employed in designing approximation algorithms. Local search is inherently a greedy strategy: when we have a feasible solution x , we try to improve it by choosing some feasible y that is *close* to x , but has a better measure (lower or higher, depending on minimization or maximization). Repeated attempts at improvement often result in *locally* optimal solutions that have a good measure relative to a globally optimal solution (i.e. a member of $Opt(I)$). We illustrate local search by giving an approximation algorithm for the NP-complete MAX-CUT problem:

MAX-CUT

Instance: An undirected graph $G = (V, E)$.

Solution: A cut of the graph, i.e. a pair (S, T) such that $S \subseteq V$ and $T = V - S$.

Measure: The *cut size*, which is the number of edges crossing the cut, i.e.

$$|\{\{u, v\} \in E \mid u \in S, v \in T\}|.$$

Our local search algorithm repeatedly improves the current feasible solution by changing one vertex's place in the cut, until no more improvement can be made. We will prove that at such a local maximum, the cut size is at least $m/2$:

Local-Cut: Start with an arbitrary cut of V . For each vertex, determine if moving it to the other side of the partition increases the size of the cut. If so, move it. Repeat until no such movements are possible.

First, observe that this algorithm repeats at most m times, as each movement of a vertex increases the size of the cut by at least 1, and a cut can be at most m in size.

Theorem 21.3. *Local-Cut is a $1/2$ -approximation algorithm for MAX-CUT.*

Proof. Let (S, T) be the cut returned by the algorithm, and consider a vertex v . After the algorithm finishes, observe that the number of edges adjacent to v that cross (S, T) is more than the number of adjacent edges that do not cross, otherwise v would have been moved. Let $\deg(v)$ be the degree of v . Then our observation implies that at least $\deg(v)/2$ edges out of v cross the cut returned by the algorithm.

Let m^* be the total number of edges crossing the cut returned. Each edge has two endpoints, so the sum $\sum_{v \in V} (\deg(v)/2)$ counts each crossing edge at most twice, i.e.

$$\sum_{v \in V} (\deg(v)/2) \leq 2m^*.$$

Using the well-known degree-edge equation $\sum_{v \in V} \deg(v) = 2m$, we conclude that

$$m = \sum_{v \in V} (\deg(v)/2) \leq 2m^*.$$

It follows that the approximation ratio of the algorithm is

$$\frac{m^*}{m} \geq \frac{1}{2}.$$

□

It turns out that MAX-CUT admits much better approximation ratios than $1/2$; a relaxation of the problem to a semidefinite linear program yields a 0.8786 approximation (see [Goemans and Williamson 1995](#)). However, like many optimization problems, MAX-CUT cannot be approximated arbitrarily well ($1 - \epsilon$, for all $\epsilon > 0$) unless $P = NP$. That is to say, it is unlikely that MAX-CUT is in the PTAS complexity class.

21.2.2.4 Greedy Knapsack

The knapsack problem and its special cases have been extensively studied in operations research. The premise behind it is classic: you have a knapsack of capacity C , and a set of items $1, \dots, n$. Each item has a particular cost c_i of carrying it, along

with a profit p_i that you will gain by carrying it. The problem is then to find a subset of items with cost at most C , having maximum profit:

Maximum Integer Knapsack

Instance: A capacity $C \in N$, and a number of items $n \in N$, with corresponding costs and profits $c_{-i}, p_i \in \mathbb{N}$ for all $i = 1, \dots, n$.

Solution: A subset $S \subseteq \{1, \dots, n\}$ such that $\sum_{j \in S} c_j \leq C$.

Measure: The total profit $\sum_{j \in S} p_j$.

Maximum Integer Knapsack, as formulated above, is NP-hard. There is also a *fractional* version of this problem (we call it Maximum Fraction Knapsack), which can be solved in polynomial time. In this version, rather than having to pick the entire item, one is allowed to choose fractions of items, like 1/8 of the first item, 1/2 of the second item, and so on. The corresponding profit and cost incurred from the items will be similarly fractional (1/8 of the profit and cost of the first, 1/2 of the profit and cost of the second, and so on).

One greedy strategy for solving these two problems is to pack items with the largest profit-to-cost ratio first, with the hopes of getting many small-cost high-profit items in the knapsack. It turns out that this algorithm will not give any constant approximation guarantee, but a tiny variant on this approach will give a 2-approximation for Integer Knapsack, and an exact algorithm for Fraction Knapsack. The algorithms for Integer Knapsack and Fraction Knapsack are, respectively:

Greedy-IKS: Choose items with the largest profit-to-cost ratio first, until the total cost of items chosen is greater than C . Let j be the last item chosen, and S be the set of items chosen before j . Return either $\{j\}$ or S , depending on which one is more profitable.

Greedy-FKS: Choose items as in Greedy-IKS. When the item j makes the cost of the current solution greater than C , add the fraction of j such that the resulting cost of the solution is exactly C .

The following is left as an exercise for the reader.

Lemma 21.1. *Greedy-FKS solves Maximum Fraction Knapsack in polynomial time.*

We entitled the result for Fraction Knapsack as a lemma, because we will use it to analyze the approximation algorithm for Integer Knapsack.

Theorem 21.4. *Greedy-KS is a 1/2-approximation for Maximum Integer Knapsack.*

Proof. Fix an instance of the problem. Let $P = \sum_{j \in S} p_j$, the total profit of items in S , and j be the last item chosen (as specified in the algorithm). We will show that $P + p_j$ is greater than or equal to the profit of an optimal Integer Knapsack solution. It follows that one of S or $\{j\}$ has at least half the profit of the optimal solution. Let S_I^* be an optimal Integer Knapsack solution to the given instance, with total profit

P_I^* . Similarly, let S_F^* and P_F^* correspond to an optimal Fraction Knapsack solution. Observe that $P_F^* = P_I^*$. By the analysis of the algorithm for Fraction Knapsack, $P_F^* = P + p_j$, where $\epsilon \in (0, 1]$ is the fraction chosen for item j in the algorithm. Therefore,

$$P + p_j \geq P + \epsilon p_j = P_F^* = P_I^*$$

and we are done. \square

Later, we will see how this algorithm can be extended (in conjunction with dynamic programming) to get a PTAS for Maximum Integer Knapsack. A PTAS is quite powerful; such a scheme can approximately solve a problem with arbitrarily close ratios to the optimal solution. However, many problems provably do not have a PTAS, unless $P = NP$.

21.2.3 Sequential Algorithms

Sequential algorithms are used for approximations on problems where a feasible solution is a partitioning of the instance into subsets. A sequential algorithm *sorts* the items of the instance in some manner, and selects partitions for the instance based on this ordering.

21.2.3.1 Sequential Bin Packing

We first consider the problem of Minimum Bin Packing, which is similar in nature to the knapsack problems.

Minimum Bin Packing

Instance: A set of items $S = r_1, \dots, r_n$, where $r_i \in (0, 1]$ for all $i = 1, \dots, n$.

Solution: Partition of S into bins B_1, \dots, B_M such that $\sum_{r_j \in B_i} r_j \leq 1$ for all $i = 1, \dots, M$.

Measure: M .

An obvious algorithm for Minimum Bin Packing is an online strategy. Initially, let $j = 1$ and have a bin B_1 available. As one runs through the input (r_1, r_2 , etc.), try to pack the new item r_i into the last bin used, B_j . If r_i does not fit in B_j , create another bin B_{j+1} and put r_i in it. This algorithm is *online* as it processes the input in a fixed order, and thus adding new items to the instance while the algorithm is running does not change the outcome. Call this heuristic Last-Bin.

Theorem 21.5. *Last-Bin is a 2-approximation to Minimum Bin Packing.*

Proof. Let R be the sum of all items, so $R = \sum_{r_i \in S} r_i$. Let m be the total number of bins used by the algorithm, and let m^* be the minimum number of bins possible for the given instance. Note that $m^* \geq R$, as the total number of bins needed is at least

the total size of all items (each bin holds one unit). Now, given any pair of bins B_i and $B_i + 1$ returned by the algorithm, the sum of items from S in B_i and $B_i + 1$ is at least 1; otherwise, we would have stored the items of $B_i + 1$ in B_i instead. This shows that $m \leq 2R$. Hence $m \leq 2R \leq 2m^*$, and the algorithm is a 2-approximation. \square

An interesting exercise for the reader is to construct a series of examples demonstrating that this approximation bound, like the one for Greedy-VC, is tight.

As one might expect, there exist algorithms that give better approximations than the above. For example, we do not even consider the previous bins B_1, \dots, B_{j-1} when trying to pack an a_i , only the last one is considered.

Motivated by this observation, consider the following modification to Last-Bin. Select each item a_i in decreasing order of size, placing a_i in the first available bin out of B_1, \dots, B_j . (So a new bin is only created if a_i cannot fit in any of the previous j bins.) Call this new algorithm First-Bin. An improved approximation bound may be derived, via an intricate analysis of cases.

Theorem 21.6. *First-Bin is a $11/9$ -approximation to Minimum Bin Packing.*

21.2.3.2 Sequential Job Scheduling

One of the major problems in scheduling theory is how to assign jobs to multiple machines so that all of the jobs are completed efficiently. Here, we will consider job completion in the shortest amount of time possible. For the purposes of abstraction and simplicity, we will assume the machines are identical in processing power for each job.

Minimum Job Scheduling

Instance: An integer k and a multi-set $T = \{t_1, \dots, t_n\}$ of times, $t_i \in \mathbb{Q}$ for all $i = 1, \dots, n$ (i.e. the t_i are fractions).

Solution: An assignment of jobs to machines, i.e. a function a from $\{1, \dots, n\}$ to $\{1, \dots, k\}$.

Measure: The completion time for all machines, assuming they run in parallel: $\max\{\sum_{i:a(i)=j} t_i \mid j \in \{1, \dots, k\}\}$.

The algorithm we propose for Job Scheduling is also on-line: when reading a new job with time t_i , assign it to the machine j that currently has the least amount of work; that is, the j with minimum $\sum_{i:a(i)=j} t_i$. Call this algorithm Sequential-Jobs.

Theorem 21.7. *Sequential Jobs is a 2-approximation for Minimum Job Scheduling.*

Proof. Let j be a machine with maximum completion time, and let i be the index of the last job assigned to j by the algorithm. Let $s_{i,j}$ be the sum of all times for jobs prior to i that are assigned to j . (This may be thought of as the time that job i begins on machine j). The algorithm assigned i to the machine with the least amount of

work, hence all other machines j' at this point have larger $\sum_{i:a(i)=j'} t_i$. Therefore, $s_{i,j} \leq 1/k \sum_{i=1}^n t_i$, i.e. $s_{i,j}$ is less $1/k$ of the total k time of all jobs (recall that k is the number of machines).

Notice $B = 1/k \sum_{i=1}^n t_i = m^*$, the completion time for an optimal solution, as the sum corresponds to the case where every machine takes exactly the same fraction of time to complete. Thus the completion time for machine j is

$$s_{i,j} + t_i = m^* + m^* = 2m^*.$$

So the maximum completion time is at most twice that of an optimal solution. \square

Minimum Job Scheduling also has a PTAS, as we will discover in the next section.

21.2.4 Dynamic Programming

Dynamic programming is, in essence, a formal name given to the tried-and-true principle of *saving your work*. It is often the case in algorithmic design that, when optimal solutions to small pieces of an instance are known, those solutions can be merged in a way that yields an optimal solution for the entire instance. This is precisely the idea behind dynamic programming: we solve pieces of the problem and save their solutions in order to solve the entire problem later on. We will demonstrate how dynamic programming may be used to extend the previous 2-approximation for Minimum Job Scheduling into a PTAS for the problem. Dynamic programming will also be employed to get a FPTAS for Maximum Integer Knapsack.

21.2.4.1 PTAS for Minimum Job Scheduling

The algorithm we present is based on one given by [Vazirani \(2004\)](#). It has the property that, for any fixed $\varepsilon > 0$ provided beforehand, it returns a $(1 + \varepsilon)$ -approximate solution. Further, the runtime is polynomial in the input size, *provided that ε is constant*. This allows us to specify a runtime that has $1/\varepsilon$ in the exponent (which we will do). It is typical to view this algorithm as a *family* of successively better (but also slower) approximation algorithms, each running with a successively smaller $\varepsilon > 0$. This is intuitively why they are called an approximation scheme; *scheme* is meant to suggest that a variety of algorithms are used.

Our treatment here will be lighter than in other sections due to the complexity of the algorithm. We will name a special case problem that is polynomial time solvable. The polytime algorithm will serve as a subroutine to a parameterized Job Scheduling procedure, where we are allowed to specify the completion time in the input. The PTAS will use this parameterized procedure to perform a binary search over the possible completion times, finding an approximate minimum completion time.

The special case problem we consider is Minimum Unit-Time c -Job Scheduling, a variant on Minimum Job Scheduling which is polynomial time solvable. It is

similar to job scheduling, except (a) the number of distinct possible times for jobs is a constant c , and (b) the completion time is fixed to be 1. (We stress that the number of jobs is not constant—the number of distinct times for jobs is.) The problem now is to minimize the number of machines necessary to schedule these jobs so they may be completed in one time unit. (The astute reader will find that this modified problem is essentially bin packing.)

Theorem 21.8. *Minimum Unit-Time c -Job Scheduling is polynomial time solvable.*

Proof. Arbitrarily order the possible job times T_1, \dots, T_c . Observe that any instance (a multi-set $S = \{t_1, \dots, t_n\}$ of items) of this problem may be specified as a c -tuple (n_1, \dots, n_c) of natural numbers: the i th component of the c -tuple (n_i) gives the number of job with time T_i .

Define $OPT(n_1, \dots, n_c)$ to be the optimal solution (minimum number of machines necessary) for the instance (n_1, \dots, n_c) . We will determine OPT for a given instance via dynamic programming, i.e. by determining its value on all possible (i_1, \dots, i_c) , where each i_j ranges from 0 to n_j . (Note there are only $O(n^c)$ such c -tuples.)

Initially, we determine for all $1 \leq j \leq c$, and $0 \leq i_j \leq n_j$, those (i_1, \dots, i_c) for which $OPT(i_1, \dots, i_c) = 1$ (exactly one machine suffices). Doing this takes constant time for each of the $O(n^c)$ c -tuples. Add these c -tuples to a *saved value* set, S .

For the remaining (i_1, \dots, i_c) (those whose OPT value is larger than 1), we determine their OPT solution via the inductive assignment

$$OPT(i_1, \dots, i_c) := 1 + \min_{(j_1, \dots, j_c) \in S} OPT(i_1 - j_1, \dots, i_c - j_c)$$

provided that $\forall l = 1, \dots, c, i_l \geq j_l$. A bit of thought shows that every possible c -tuple's OPT value may be computed in this way, provided the c -tuples are processed in the correct order.

An OPT value for a fixed c -tuple, assuming all of the other necessary OPT values are present, requires at most $O(n^c)$ time to compute. As there are $O(n^c)$ possible c -tuples, the overall runtime is $O(n^{2c})$. \square

As expected, this dynamic programming algorithm will be a component in our description of the PTAS. However, before we give the PTAS, we need to describe another procedure that will use the dynamic programming algorithm.

Let $\varepsilon > 0$ be fixed, and $(T = \{t_1, \dots, t_n\}, k)$ be a job scheduling instance (where k is number of machines to be used). From the analysis of the 2-approximation, there exists a (polytime computable) lower bound B on the optimum, and an upper bound, which is $2B$. Let $V \in [B, 2B]$. The next algorithm, Parameter-JS, returns a $(1 + \varepsilon)$ -approximate solution for T , given that the parameter V is the desired completion time. Let Unit-cJS be the algorithm from the above theorem.

Parameter-JS(S, V, ε): Define a job time t_j to be short if $t_j < V$. Remove the short jobs from T . For the remaining jobs, round their sizes down to the nearest power of $(1 + \varepsilon)$, times εV . More precisely, for each t_j remaining in T , compute the unique x such that $t_j \in [(1 + \varepsilon)x \cdot \varepsilon V, (1 + \varepsilon)^{x+1} \cdot \varepsilon V]$, and set $t'_j := (1 + \varepsilon)^x \cdot \varepsilon V$.

Let $T' = \{t'_j | t_j \in T\}$. Now the possible times for jobs in T' are $\varepsilon V, (1 + \varepsilon)V, (1 + \varepsilon)2V$, and so on. There are therefore

$$k = \left\lceil \frac{\log(1/\varepsilon)}{\log(1 + \varepsilon)} \right\rceil$$

distinct possible times. Further, all of these times are a multiple of V .

Let $T'' = \{t_j/V | t_j \in T\}$. Call Unit-cJS on T'' and the above k , getting back a solution S . As Unit-cJS assumes the completion time to be 1, the resulting Unit-cJS solution for T'' is also a solution for T' , where the completion time is V . Greedily schedule the short jobs back in this solution, on whatever machines will accommodate them (without exceeding the completion time V). Introduce new machines for handling short jobs only when the job cannot be scheduled on any of the existing machines without exceeding V . Return the resulting solution.

Let us observe a lemma, which follows from a simple analysis of the short jobs' impact on the Unit-kJS solution for T' .

Lemma 21.2. *Parameter-JS on T', V , and ε returns a job schedule for a minimum number of machines, with completion time that is at most $(1 + \varepsilon)V$.*

Proof. (Sketch) Each job's time $t'_j \in T'$ is rounded down by at most a $(1 + \varepsilon)$ factor from the original t_j , the solution returned by the algorithm is feasible if the minimum completion time in the original instance T (modulo short jobs) is $(1 + \varepsilon)V$. \square

Let $OPT(T, V)$ be the minimum number of machines needed to schedule the jobs of instance T when the completion time is exactly V , and let $M(T, V)$ be the number of machines used by Parameter-JS(S, V, ε).

Lemma 21.3. $M'(T, V) \leq OPT(T, V)$.

The proof involves an analysis of short jobs, and is left to the reader.

Finally, we are in a position to describe the PTAS. We wish to find the optimal completion time V^* for a given number of machines k , which is located somewhere in the interval $[B, 2B]$ (recall that B and $2B$ are lower and upper bounds on the optimal, respectively). The above lemma implies

$$\min\{V | M(T, V) = k\} = \min\{V | OPT(T, V) = k\} = V^*.$$

To find V^* , the PTAS will perform a binary search over fractions in the interval $[B, 2B]$, calling Parameter-JS on the current V -value, until the remaining interval size is at most B . At this point, we will argue that the solution obtained is approximately close to an optimal one:

PTAS-MJS(T, k, ε):

Initially, $F := B$ (first), $L := 2B$ (last).

Repeat $\lceil \log_2(3/\varepsilon) \rceil$ times:

Set $M := (F + L)/2$, the midpoint of the interval.

Call Parameter-JS($S, M, \varepsilon/3$), getting a solution S' .

If S' uses at most k machines, then set $L := M$; if not, set $F := M$. End Repeat.
Return L and S .

Based on the runtime of Parameter-JS, it is easy to verify that PTAS-MJS runs in $O(\log_2(1/\varepsilon)n^{\lceil 2\log_{1+\varepsilon}(1/\varepsilon) \rceil})$ time.

Theorem 21.9. *PTAS-MJS is a $(1 + \varepsilon)$ -approximation for Minimum Job Scheduling.*

Proof. We first prove that the L returned is at most $(1 + \varepsilon)V^*$. When the repeat-loop is completed, $|F - L| \leq \varepsilon B$, as the initial length of the interval is B and each iteration of the loop decreases that length by a factor of 2.

Hence

$$L - B = \min\{V \mid M'(T, V) \leq k\} \leq L$$

implying (via the lemma)

$$L = \min\{V \mid M'(T, V) \leq k\} + \varepsilon B \leq \min\{V \mid OPT(T, V) \leq k\} + \varepsilon V^* = (1 + \varepsilon)V^*.$$

Applying Lemma 21.2, we infer that the value of S returned by PTAS-MJS is

$$M'(T, V) \leq (1 + \varepsilon/3)M \leq (1 + \varepsilon/3)L \leq (1 + \varepsilon/3)2V^* \leq (1 + \varepsilon)V^*.$$

Therefore, the S' returned is within $1 + \varepsilon$ of the optimal minimum solution. \square

21.2.4.2 FPTAS for Knapsack

Our next example for which we can get arbitrarily good approximations will have the added benefit that, as the approximation ratio of the algorithm improves, the runtime does not get larger than a fixed polynomial in n . This is known as a *fully polynomial time approximation scheme*.

The strategy we will use for developing the FPTAS is similar to that for Job Scheduling, but the algorithm itself will be much simpler. We first look at a special case of Maximum Integer Knapsack where some parameter in the problem is held constant, and show it to be polynomial time solvable via dynamic programming. Then we use this polynomial time algorithm to derive an approximation algorithm for the general problem.

Recall that in the Minimum Integer Knapsack problem we have a capacity $C \in \mathbb{N}$, and a number of items $n \in \mathbb{N}$, with corresponding costs and profits $c_i, p_i \in \mathbb{N}$ for all $i = 1, \dots, n$. The objective is to pack a knapsack of capacity C with items of cost at most C , and maximum profit. Here, the special case problem will be Maximum k -Profit Integer Knapsack; in this restriction, all of the items' profits are bounded from above by a constant k .

Theorem 21.10. *Maximum k -Profit Integer Knapsack is solvable in polynomial time.*

Proof. Given an instance of the problem with n items (labeled $1, \dots, n$), since all profits are bounded from above by k , kn is an upper bound on the optimum profit for the instance. We will now set up a dynamic program that computes optimum solution for subsets of the instance, from 0 profit up to kn profit.

For every $i = 1, \dots, n$ and $j = 1, \dots, kn$, define $OPT(i, j)$ to be the minimum capacity of a packing over items $1, \dots, i$ that has profit exactly j (and $OPT(i, j) = \infty$ if no packing exists).

Note that $OPT(1, j) = c_1$ if $p_1 = j$, and ∞ otherwise. To determine OPT for $i > 1$, we use the following inductive equation:

$$OPT(i, j) := \begin{cases} \min\{OPT(i-1, j), OPT(i, j-p_i) + c_i\} & \text{if } p_i < j \\ OPT(i-1, j) & \text{otherwise.} \end{cases}$$

Intuitively, this just means that we either add the i th item to the solution or not, depending on which is smaller and whether or not the profit of item i exceeds the total profit j . For each $OPT(i, j)$ value, we also save a subset of $\{1, \dots, i\}$ with profit j that achieves the $OPT(i, j)$ value.

Using the inductive equation, we can compute all OPT values in $O(kn^2)$ time. Notice that the maximum profit possible for the instance is

$$\max\{j : j = 1, \dots, kn, OPT(n, j) < \infty\}$$

i.e. the maximum profit over the feasible OPT values. Hence, after the OPT values are computed, it takes $O(kn)$ time to find an optimal solution S' corresponding to the maximum profit. \square

The central idea behind our FPTAS for the general Integer Knapsack problem is to *discard least significant bits* of the profits of items. That is, we divide all of the profits of items by some parameter (depending on ε), reducing the instance to one that has a small upper bound k on the profit size. Then the above algorithm will be used to determine an exact solution S' for this reduced instance, which roughly corresponds to an approximate solution for the original instance. Finally, we will use a trick from the 2-approximation for Knapsack: we will either return the solution S' , or the most profitable item of cost at most C , whichever is better.

Fix an $\varepsilon > 0$. Let Dyn-MBIK be the polynomial-time algorithm for Maximum k -Profit Integer Knapsack from Theorem 21.10.

FPTAS-MIK: For all $i = 1, \dots, n$, re-define $p_i := \lceil \frac{p_i \cdot n}{\max_j \varepsilon p_j} \rceil$. Run Dyn $\tilde{\cup}_{i=1}^n$ MBIK, and get a solution S' . Let i be s.t. $p_i = \max_j p_j$. Return S' or $\{i\}$, depending on which has higher profit.

Theorem 21.11. *FPTAS-MIK runs in $O(n^3/\varepsilon)$ time, and is a $(1 - \varepsilon)$ -approximation.*

Notice that the runtime is now polynomial in n and $1/\varepsilon$, so this algorithm indeed qualifies as an FPTAS for the problem.

Proof. First, from the analysis of Dyn-MBIK, the runtime of FPTAS-MIK is

$$O\left(n^2 \max_i \left\lceil \frac{p_i \frac{1}{2} n}{\max_j \epsilon p_j} \right\rceil\right),$$

i.e. $O(n^3/\epsilon)$ time.

Now we will show that the algorithm is a $(1 - \epsilon)$ -approximation. Let S^* be a subset of $\{1, \dots, n\}$ that is an optimal solution for an instance, let S' be the subset returned by the Dyn-MBIK call, and let $P(S^*)$ (respectively $P(S')$) be the corresponding profits. Our goal is to show that the profit of the set returned by the algorithm (call it P') is at least $(1 - \epsilon)P(S^*)$.

Let $P(S^*)$ be the profit of S^* , under the redefined profits given in the algorithm. It is straightforward to verify that

$$P(S^*) \leq \max_j \epsilon p_j + (\max_j p_j/n) \frac{1}{2} P'(S^*).$$

Dyn-MBIK returns the optimal solution under redefined profits, hence

$$P(S') \geq (\max_j \epsilon p_j/n) \frac{1}{2} P'(S^*) \geq P(S^*) - \max_j \epsilon p_j.$$

By definition of the algorithm, the profit of the solution returned P' is at least $\max_j p_j$. Since $P' \leq P(S^*)$, it follows that

$$P' \geq P(S') \geq P(S^*) - \max_j \epsilon p_j \geq P(S^*) - \epsilon P' \geq (1 - \epsilon)P(S^*). \quad \square$$

21.2.5 LP-Based Algorithms

Linear programming (LP) plays an important role on the design of algorithms for combinatorial optimization problems (see e.g. Chvatal 1983; Papadimitriou and Steiglitz 1982; Schrijver 2003). The use of LP for the design and analysis of approximation algorithms for NP-hard problems dates back to the 1970s (Chvatal 1979; Lovasz 1975; see also Wolsey 1980). In this section we show how we can use LP techniques to design approximation algorithms. We start by showing how LP rounding can be used to derive a 2-approximation algorithm for the minimum-weight vertex cover problem. We then describe the primal–dual method, based on LP duality. We provide background material on LP duality and its most important theorems.

21.2.5.1 LP Rounding

A simple way of obtaining an approximation algorithm based on linear programming is to solve the problem as a linear program and convert its fractional solution into an integral solution, by *rounding* it. We use LP rounding to approximate the Minimum-Weight Vertex Cover problem (Hochbaum 1982, 1983).

Minimum-Weight Vertex Cover

Instance: An undirected graph $G = (V, E)$, and a positive weight function, $W : V \rightarrow \mathbb{R}^+$ on the vertices.

Solution: A subset $S \subseteq V$ such that for every $\{u, v\} \in E$, either $u \in S$ or $v \in S$.

Measure: $\sum_{v \in S} w(v)$.

In order to formulate this problem as a linear program, let us associate a variable $x(v)$ with each vertex $v \in V$, and require that $x(v) \in \{0, 1\}$, for each $v \in V$. The interpretation of the $x(v)$ is the following: if $x(v) = 1$, the vertex v belongs to the vertex cover; if $x(v) = 0$, the vertex v does not belong to the vertex cover. The constraint that requires that for any edge (u, v) , at least one of the u and v must be in the vertex cover, can be stated as $x(u) + x(v) = 1$. We obtain the following integer program:

0–1 integer program

$$\begin{aligned} &\text{Minimize } \sum_{v \in V} w(v)x(v) \\ &\text{subject to : } x(u) + x(v) \geq 1 \text{ for each } v \in V \\ &\quad x(v) \in \{0, 1\} \quad \text{for each } v \in V. \end{aligned}$$

Finding the solution to this integer program is NP-hard. However, if we relax the constraint $x(v) \in \{0, 1\}$, and replace with $0 \leq x(v) \leq 1$, we obtain what is called its LP relaxation:

LP relaxation of the 0–1 integer program

$$\begin{aligned} &\text{Minimize } \sum_{v \in V} w(v)x(v) \\ &\text{subject to : } x(u) + x(v) \geq 1 \text{ for each } v \in V \\ &\quad x(v) \geq 0 \quad \text{for each } v \in V \\ &\quad x(v) \leq 1 \quad \text{for each } v \in V. \end{aligned}$$

Note that any feasible solution to the 0–1 integer program is also a feasible solution to its LP relaxation. Therefore, an optimal solution to the LP relaxation is a lower bound on the optimal solution of the 0–1 integer program.

LP-Rounding-MVC: Formulate the minimum-weight vertex cover as the above relaxed linear program. Compute its optimal solution x^* . Initially, we have an empty set S . For each vertex $v \in V$, if $x^*(v) \geq 1/2$, add v to S . Return S as the minimum-weight vertex cover.

Theorem 21.12. *LP-Rounding-MVC is a 2-approximation algorithm for Minimum Weight Vertex Cover.*

Proof. We start by noting that we can solve a linear program in polynomial time (see for example Papadimitriou and Steiglitz 1982). Let us now show that S is indeed a vertex cover. Suppose not; then there exists an edge $e = (u, v)$ which was not covered by any vertex in S . That implies that $x(u) < 1/2$ and $x(v) < 1/2$. But, since we know that x^* is a feasible solution to the relaxed linear program, it has to satisfy the constraint $x(u) + x(v) \geq 1$, which leads to a contradiction if we assume that both $x(u) < 1/2$ and $x(v) < 1/2$.

Next, we show that our procedure is a 2-approximation algorithm for the minimum weight vertex cover problem. Let us denote the value of the solution obtained with the LP-Rounding-MVC procedure by $m_{LPR-MWVC}$, the optimal value of the minimum weight vertex cover by m_{MWVC} , and the optimal value of the LP-relaxation of the formulation of the minimum weight vertex cover as an integer program by $m_{LP-MWVC}$:

$$\begin{aligned} m_{LPR-MWVC} &= \sum_{v \in S} w(v) \\ &\leq \sum_{v \in S} 2x^*(v)w(v) \leq \sum_{v \in V} 2x^*(v)w(v) \\ &= 2m_{LP-MWVC}. \end{aligned}$$

Since $m_{LP-MWVC} \leq m_{MWVC}$, it follows that

$$m_{LPR-MWVC} \leq 2m_{MWVC}.$$

□

21.2.6 Primal–Dual Method

Some of the most fundamental exact polynomial time algorithms exploit min–max relations that characterize the structure of several combinatorial problems. Most of such min–max relations are special cases of the duality theorem in LP. A great deal of the theory of approximation algorithms also exploits such min–max relations and is based on LP and the LP-duality theory. In this section we start by briefly reviewing some key concepts of the LP-duality. We then describe the primal–dual method and apply it to the minimum weight vertex cover problem.

21.2.6.1 The LP Duality Theorem

Let us illustrate the importance of LP duality through an example:

$$\begin{array}{ll} \text{Minimize} & 15x_1 + 7x_2 + 5x_3 \\ \text{subject to :} & \begin{array}{ll} 3x_1 + x_2 - x_3 & \geq 2 \\ x_1 + x_2 + x_3 & \geq 1 \\ x_1, x_2, x_3 & \geq 0. \end{array} \end{array}$$

This problem is the standard form of an LP minimization problem, with all the constraints of the form “ \geq ”, and all the variables constrained to be non-negative. Any minimization LP problem can be written in this format.

Instead of solving this linear program, we can try to provide bounds on its optimal solution, z^* . Let us consider a question of the form: “is z^* at most 15?”

A feasible solution to our problem, with value at most 15, is a Yes certificate to that question. For example, the solution $x_1 = 1$, $x_2 = 1$, and $x_3 = 1$ is such a certificate since it satisfies all the constraints of the problem and the objective function value associated with that solution is $12 < 15$. In other words, any feasible solution of our problem provides an upper bound on z^* . On the other hand, if we are interested in lower bounds for z^* , a good estimate can be based on the bounds associated with the constraints. For example, if we consider the first constraint, given that all its coefficients are smaller than the coefficients of the objective function (we are dealing with a minimization problem), and that all the x_i are non-negative, we can infer that

$$15x_1 + 7x_2 + 5x_3 = 3x_1 + x_2 - x_3 = 2.$$

In other words, the objective function value is at least 2. We can improve this bound by multiplying the first equation by 4 and the second equation by 2, obtaining a lower bound of 10, i.e.

$$15x_1 + 7x_2 + 5x_3 = (12x_1 + 4x_2 - 4x_3) + (2x_1 + 2x_2 + 2x_3) = 10.$$

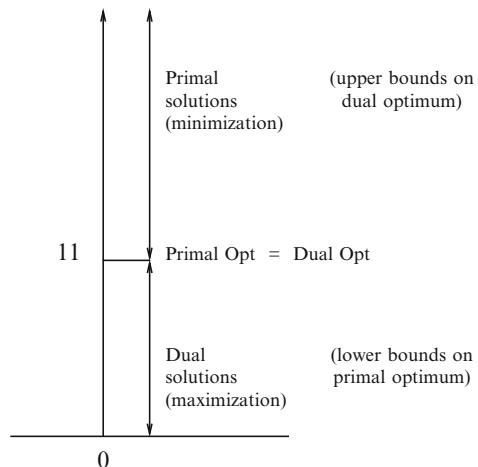
Basically, we are considering linear combinations of the constraints, multiplying the first constraint by some multiplier y_1 and the second by y_2 . Because we have a minimization problem our goal is to have the coefficient of each x_i in the linear combination of the constraints as close to the corresponding coefficient of x_i in the objective function as possible, but not greater than it. Furthermore, the multipliers have to be non-negative to make sure that the direction of the inequality is not reversed. And of course, we would like our lower bound, given by the sum of the right-hand sides of the constraints multiplied by the corresponding multiplier, to be as large as possible.

This problem of finding the largest lower bound for our minimization problem can be formulated also as a linear program:

$$\begin{aligned} \text{Maximize } & 2y_1 + y_2 \\ \text{subject to : } & 3y_1 + y_2 \leq 15 \\ & y_1 + y_2 \leq 7 \\ & -y_1 + y_2 \leq 5 \\ & y_1, y_2, y_3 \geq 0. \end{aligned}$$

This second linear program is called the *dual* of the original one. The original linear program is called the *primal*. Note that y_1 can be interpreted as the multiplier of the first constraint of our primal program, the original minimization problem, and y_2 as the multiplier of the second constraint. In this framework, the first constraint of the dual program states that the linear combination of the constraint coefficients

Fig. 21.2 Primal–dual relationships



of x_1 in the primal program (3 and 1) cannot be greater than x_1 's objective function coefficient (15). The objective function of this dual program states that we want to maximize our primal lower bound, i.e. the right-hand sides of our primal program, multiplied by the corresponding multipliers.

Every minimization LP problem in the standard form has a dual LP maximization problem. More interestingly, as we observed before, every feasible solution of the dual provides a lower bound on the optimum of the primal. The dual relationship also holds: every primal feasible solution is an upper bound on the optimum of the dual. In fact, if one has an optimal solution so does the other and the two optimal values coincide. In our example the optimal value is 11, which corresponds to the primal solution $x_1 = 0.5$, $x_2 = 0.5$, and $x_3 = 0$ and the dual solution $y_1 = 4$ and $y_2 = 3$ (see Fig. 21.2). The dual of a maximization LP problem is a minimization LP problem. The dual of the dual of a given problem is the problem itself.

What we have just observed in the example corresponds to a key theorem in linear programming, the duality theorem. In a more formal way, let us consider the pair of primal and dual linear programs:

Primal linear program

$$\begin{aligned} & \text{Minimize } \sum_{j=1}^n c_j x_j \\ & \text{subject to : } \sum_{j=1}^n a_{ij} x_j \geq b_i, i = 1, \dots, m \\ & \quad x_j \geq 0, j = 1, \dots, n \end{aligned}$$

Dual linear program

$$\begin{aligned} & \text{Minimize } \sum_{i=1}^m b_i y_i \\ & \text{subject to : } \sum_{i=1}^m a_{ij} y_i \geq c_j, j = 1, \dots, n \\ & \quad y_i \geq 0, i = 1, \dots, m \end{aligned}$$

where a_{ij} , c_j , and b_i are given rational numbers. We can now state the duality theorem in a formal way.

Theorem 21.13. Duality Theorem *The primal program has an optimal solution if and only if its dual has an optimal solution. Furthermore, if $x^* = (x_1^*, \dots, x_n^*)$ and $y = (y_1^*, \dots, y_m^*)$ are the optimal solutions of the primal and the dual program, then*

$$\sum_{j=1}^n c_j x_j^* = \sum_{i=1}^m a_{ij} y_i^*.$$

Note that given the duality relationships we could have the primal problem as a maximization problem and the dual problem as a minimization problem. The duality theorem provides a succinct way of proving optimality: an optimal solution of the dual problem provides a certificate of optimality for an optimal solution of the primal, and vice versa. As in our example, every feasible solution of the dual provides a lower bound on the optimal value of the primal and of course on the objective function value of any feasible primal solution. This corresponds to half of the duality theorem, referred to as the weak duality theorem. In the design of approximation algorithms, in general, the weak duality theorem is sufficient.

Theorem 21.14. Weak Duality Theorem *If $x = (x_1, \dots, x_n)$ and $y = (y_1, \dots, y_m)$ are the feasible solutions of the primal and the dual program, respectively, then*

$$\sum_{j=1}^n c_j x_j \geq \sum_{i=1}^m b_i y_i.$$

Proof. Given the non-negativity of the x_j and given that y is feasible:

$$\begin{aligned} \sum_{j=1}^n c_j x_j &\geq \sum_{j=1}^n \left(\sum_{i=1}^m a_{ij} y_i \right) x_j \\ \sum_{j=1}^n \left(\sum_{i=1}^m a_{ij} y_i \right) x_j &= \sum_{i=1}^m \left(\sum_{j=1}^n a_{ij} x_j \right) y_i \\ \sum_{i=1}^m \left(\sum_{j=1}^n a_{ij} x_j \right) y_i &\geq \sum_{i=1}^m b_i y_i. \end{aligned}$$

Therefore,

$$\sum_{j=1}^n c_j x_j \geq \sum_{i=1}^m b_i y_i. \quad \square$$

From the duality theorem we know that x and y are optimal solutions if and only if the objective function value of the primal and the dual are equal. We can break this condition down into important conditions, known as the complementary slackness conditions.

Theorem 21.15. Complementary slackness conditions *Let x and y be primal and dual feasible solutions. Necessary and sufficient conditions for the optimality of x and y are:*

- Primal complementary slackness conditions

$$\sum_{i=1}^m a_{ij}y_i = c_j \quad \text{or} \quad x_j = 0 \quad (\text{or both}) \text{ for each } j = 1, \dots, n.$$

- Dual complementary slackness conditions

$$\sum_{j=1}^n a_{ij}x_j = b_i \quad \text{or} \quad y_i = 0 \quad (\text{or both}) \text{ for each } i = 1, \dots, m.$$

Proof. From the definitions of the primal and the dual:

$$\begin{aligned} c_j x_j &\geq \left(\sum_{i=1}^m a_{ij}y_i \right) x_j \quad (j = 1, \dots, n), \\ \left(\sum_{j=1}^n a_{ij}x_j \right) y_i &\geq b_i y_i \quad (i = 1, \dots, m). \end{aligned}$$

Therefore,

$$\sum_{j=1}^n c_j x_j \geq \sum_{j=1}^n \left(\sum_{i=1}^m a_{ij}y_i \right) x_j = \sum_{i=1}^m \left(\sum_{j=1}^n a_{ij}x_j \right) y_i \geq \sum_{i=1}^m b_i y_i.$$

For this equation to hold throughout, the two equations above have to hold in equality. For the first equation to hold in equality we have to have $x_j = 0$ or $c_j = \sum_{i=1}^m a_{ij}y_i$. Similarly, for the second equation to hold in equality we need $y_i = 0$ or $\sum_{j=1}^n a_{ij}x_j = b_i$. \square

The complementary slackness conditions play a very important role in the design of both exact and approximation algorithms. Below we discuss how we can use them to design an approximation algorithm based on the primal–dual method for the minimum weighted vertex cover.

21.2.7 Primal–Dual Method Applied to Minimum Weight Vertex Cover

Our approximation algorithm for the Minimum Weight Vertex Cover based on LP rounding requires that we solve the LP relaxation of its integer program formulation. If we are dealing with large graphs with many edges, this procedure is relatively expensive, since the number of constraints of our linear program corresponds to the

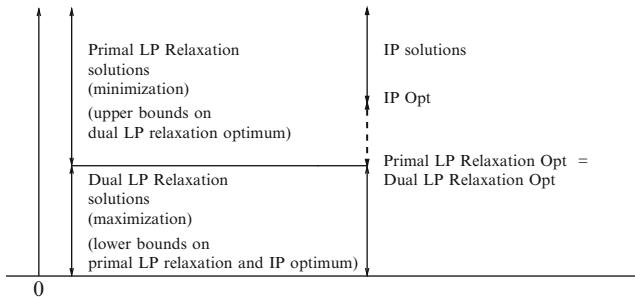


Fig. 21.3 Feasible solutions of integer program and its primal and dual relaxations

number of edges in the graph. An alternative to this procedure is the primal–dual method, also based on linear programming.

The primal–dual method was originally proposed by [Dantzig et al. \(1956\)](#), inspired by work on a min–max relation for the assignment problem that led to the primal–dual *Hungarian Method* for solving the assignment problem ([Egervary 1931; Kuhn 1955](#)). Although the primal–dual method as proposed originally is no longer used to solve LP problems, it has found several applications to develop algorithms for combinatorial optimization problems. In fact, several fundamental algorithms in combinatorial optimization are based on the primal–dual method or can be understood in terms of it. Examples include, in addition to the Hungarian algorithm, Dijkstra’s shortest-path algorithm, and Ford and Fulkerson’s network flow algorithm. The primal–dual method has also been used to obtain approximation algorithms for several NP-hard optimization problems—see [Goemans and Williamson \(1997\)](#) for a survey.

The primal–dual method provides, in fact, a general framework to devise approximation algorithms for several NP-hard optimization problems. Most primal–dual approximation algorithms enforce one of the complementary slackness conditions relaxing the other. The method starts with the LP relaxation of the primal program and iteratively builds an integral solution to the primal and a feasible solution to the dual program. The primal and the dual programs are used to guide this procedure. At each iteration, the algorithm improves the feasibility of the primal solution and the optimality of the dual solution, in such a way that at the end, the final primal solution is integral and feasible.

As mentioned in the previous section, any feasible solution to the dual provides a lower bound to the optimal solution of the primal. The performance guarantee is obtained by comparing the two solutions (see Fig. 21.3). For many problems, the performance guarantee of primal–dual methods is similar to the one obtained with LP rounding, close to the *integrality gap* of the relaxation. The runtimes of primal–dual methods are in general much faster than solving the LP relaxation since such methods are more versatile and exploit the combinatorial structure of the problem. In fact, for several problems, once we formulate them as linear programs and have in place the duality framework, a simple combinatorial algorithm can be used without further need of the linear programming tools.

Let us consider again the LP relaxation of the integer program formulation of the minimum-weight vertex cover problem.

LP relaxation of minimum-weight vertex cover IP problem

$$\begin{aligned} & \text{Minimize } \sum_{v \in V} w(v)x(v) \\ & \text{subject to: } x(u) + x(v) \geq 1 \quad \text{for each } v \in V \\ & \quad x(v) \geq 0 \quad \text{for each } v \in V. \end{aligned}$$

We obtain the dual of the LP relaxation of the IP by associating a multiplier to each of its constraints. Note that each constraint corresponds to an edge in the original graph. Therefore, to each edge $e \in E$ we associate a dual variable denoted by $y(e)$. The constraints of the dual problem state that, for each node $v \in V$, the sum of the dual variables associated with the edges incident to it have to be less than the weight of the node v , $w(v)$. The objective function of the dual is to maximize the lower bound, which corresponds to the sum of the multipliers $y(e), e \in E$.

Dual LP relaxation of minimum-weight vertex cover IP problem

$$\begin{aligned} & \text{Maximize } \sum_{e \in E} y(e) \\ & \text{subject to: } \sum_{e=(u,v) \in E} y(e) \leq w(v) \quad \text{for each } v \in V \\ & \quad y(e) \geq 0 \quad \text{for each } e \in E. \end{aligned}$$

The weak-duality theorem states that

$$\text{Dual-LP-cost} = \text{Primal-LP-cost}.$$

Therefore,

$$\text{Dual-LP-cost} \leq \text{Dual-LP-cost}^* \leq \text{Primal-LP-cost}^*.$$

Since $\text{Primal-LP-cost}^* = \text{OPT-IP}$, it is not necessary to find the optimal solution to the dual problem. We just need to find a feasible solution that will allow us to upper-bound the cost of the vertex cover as a function of the dual feasible solution cost.

The primal–dual weight vertex cover algorithm (PD-MVC) starts from a feasible dual solution in which all the $y(e)$ are set to zero and an infeasible primal solution corresponding to the empty set. It then improves the dual solution while moving towards a feasible primal solution using the complementary slackness conditions.

PD-MVC: Initially, we have an empty set S and set all the dual variables $y(e)$ to 0. Choose an edge $e = (u, v)$ not covered by S . Increase the value of the dual variable $y(e)$ until a constraint of the Dual-LP relaxation of the IP becomes tight (i.e. satisfied in equality). The vertex for which the constraint becomes tight, say v is then added to the cover S . All the edges incident to the vertex v are removed from the graph. Repeat until no edges remain in the graph. Return S as the minimum-weight vertex.

Theorem 21.16. *PD-MVC is a 2-approximation algorithm for Minimum Weight Vertex Cover.*

Proof. By construction, S is a feasible solution. We now show that indeed this procedure provides a solution at most twice the value of the optimal solution to the Minimum Weight Vertex Cover problem.

We start by observing that

$$w(v) = \sum_{u:e=(u,v) \in E} y(e) \quad \text{for each } v \in S. \quad (21.1)$$

Let us denote by m_{PD-MVC} the value obtained by the primal–dual method and, as before, by m_{MWVC} the optimal solution value of the Minimum Weight Vertex Cover problem:

$$m_{PD-MVC} = \sum_{v \in S} w(v) = \sum_{v \in S} \sum_{u:e=(u,v) \in E} y(e) \leq \sum_{v \in V} \sum_{u:e=(u,v) \in E} y(e) = 2 \sum_{e \in E} y(e). \quad (21.2)$$

Given that

$$\sum_{e \in E} y(e) \leq m_{MWVC} \quad (21.3)$$

it follows that

$$m_{PD-MVC} = 2m_{MWVC}. \quad (21.4)$$

□

21.2.8 Randomization

Randomness is a powerful resource for algorithmic design. Upon the assumption that one has access to unbiased coins that may be flipped and their values (heads or tails) extracted, a wide array of new mathematics may be employed to aid the analysis of an algorithm. It is often the case that a simple randomized algorithm will have the same performance guarantees as a complicated deterministic (i.e. non-randomized) procedure.

One of the most surprising discoveries in the area of algorithm design is that the addition of randomness into the computational process can sometimes lead to a significant speedup over purely deterministic methods. This may be intuitively explained by the following set of observations. A randomized algorithm can be viewed as a probability distribution on a set of deterministic algorithms. The behavior of a randomized algorithm can vary on a given input, depending on the random choices made by the algorithm; hence when we consider a randomized algorithm, we are implicitly considering a randomly chosen algorithm from a family of algorithms. If a substantial fraction of these deterministic algorithms perform well on the given

input, then a strategy of restarting the randomized algorithm after a certain point in runtime will lead to a speed-up ([Gomes et al. 1998](#)).

Some randomized algorithms are able to efficiently solve problems for which no efficient deterministic algorithm is known, such as polynomial identity testing (see [Motwani and Raghavan 1995](#)). Randomization is also a key component in the popular simulated annealing method for solving optimization problems ([Kirkpatrick et al. 1983](#)). For a long time, the problem of determining if a given number is prime (a fundamental problem in modern cryptography) was only efficiently solvable using randomization ([Goldwasser and Kilian 1986](#); [Rabin 1980](#); [Solovay and Strassen 1977](#)). More recently, a deterministic algorithm for primality has been discovered ([Agrawal et al. 2004](#)).

21.2.8.1 Random MAX-CUT Solution

We saw earlier a greedy strategy for MAX-CUT that yields a 2-approximation. Using randomization, we can give an extremely short approximation algorithm that has the same performance in approximation, and runs in expected polynomial time.

Random-Cut: Choose a random cut (i.e. a random partition of the vertices into two sets). If there are $< m/2$ edges crossing this cut, repeat.

Theorem 21.17. *Random-Cut is a $1/2$ -approximation algorithm for MAX-CUT that runs in expected polynomial time.*

Proof. Let X be a random variable denoting the number of edges crossing a cut. For $i = 1, \dots, m$, X_i will be an indicator variable that is 1 if the i th edge crosses the cut, and 0 otherwise. Then $X = \sum_{i=1}^m X_i$, so by linearity of expectation, $E[X] = \sum_{i=1}^m E[X_i]$.

Now for any edge u, v , the probability it crosses a randomly chosen cut is $1/2$. (Why? We randomly placed u and v in one of two possible partitions, so u is in the same partition as v with probability $1/2$.) Thus, $E[X_i] = 1/2$ for all i , so $E[X] = m/2$.

This only shows that by choosing a random cut, we expect to get at least $m/2$ edges crossing. We want a randomized algorithm that always returns a good cut, and its running time is a random variable whose expectation is polynomial. Let us compute the probability that $X = m/2$ when a random cut is chosen. In the worst case, when $X = m/2$ all of the probability is weighted on m , and when $X < m/2$ all of the probability is weighted on $m/2 - 1$. This makes the expectation of X as high as possible, while making the likelihood of obtaining an at-least- $m/2$ cut small. Formally,

$$m/2 = E[X] \leq (1 - Pr[X \geq m/2])(m/2 - 1) + Pr[X \geq m/2]m.$$

Solving for $Pr[X \geq m/2]$, it is at least $2/(m+2)$. It follows that the expected number of iterations in the above algorithm is at most $(m+2)/2$; therefore the algorithm runs in expected polynomial time, and always returns a cut of size at least $m/2$. \square

Had we simply specified our approximation as “pick a random cut and stop”, we would say that the algorithm runs in linear time, and has an expected approximation ratio of $1/2$.

21.2.8.2 Random MAX-SAT Solution

Previously, we studied a greedy approach for MAX-SAT that was guaranteed to satisfy half of the clauses. Here we will consider MAX-Ak-SAT, the restriction of MAX-SAT to CNF formulas with *at least* k literals per clause. Our algorithm is analogous to the one for MAX-CUT: Pick a random assignment to the variables. It is easy to show, using a similar analysis to the above, that the expected approximation ratio of this procedure is at least $1 - 1/2^k$. More precisely, if m is the total number of clauses in a formula, the expected number of clauses satisfied by a random assignment is $m - m/2^k$.

Let c be an arbitrary clause of at least k literals. The probability that each of its literals were set to a value that makes them false is at most $1/2^k$, since there is a probability of $1/2$ for each literal and there are at least k of them. Therefore, the probability that c is satisfied is at least $1 - 1/2^k$. Using a linearity of expectation argument (as in the MAX-CUT analysis) we infer that at least $m - m/2^k$ clauses are expected to be satisfied.

21.3 A Tour of Approximation Classes

We will now take a step back from our algorithmic discussions, and briefly describe a few of the common complexity classes associated with NP optimization problems.

21.3.1 PTAS and FPTAS

21.3.1.1 Definition

PTAS and FPTAS are classes of optimization problems that some believe are closer to the proper definition of what is efficiently solvable, rather than merely P . This is because problems in these two classes may be approximated with constant ratios arbitrarily close to 1. However, with PTAS, as the approximation ratio gets closer to 1, the runtime of the corresponding approximation algorithm may grow exponentially with the ratio.

More formally, PTAS is the class of NPO problems Π that have an approximation scheme. That is, given $\epsilon > 0$, there exists a polynomial time algorithm A_ϵ such that:

- If Π is a maximization problem, A_ϵ is a $(1 + \epsilon)$ approximation, i.e. the ratio approaches 1 from the right.

- If Π is a minimization problem, it is a $(1 - \varepsilon)$ approximation (the ratio approaches 1 from the left).

As we mentioned, one drawback of a PTAS is that the $(1 + \varepsilon)$ algorithm could be exponential in $1/\varepsilon$. The class FPTAS is essentially PTAS but with the additional requirement that the runtime of the approximation algorithm is polynomial in n and $1/\varepsilon$.

21.3.1.2 A Few Known Results

It is known that some NP-hard optimization problems cannot be approximated arbitrarily well unless $P = NP$. One example is a problem we looked at earlier, Minimum Bin Packing. This is a rare case in which there is a simple proof that the problem is not approximable unless $P = NP$.

Theorem 21.18 (Aho et al. 1979). *Minimum Bin Packing is not in PTAS, unless $P = NP$. In fact, there is no $3/2 - \varepsilon$ approximation for any $\varepsilon > 0$, unless $P = NP$.*

To prove the result, we use a reduction from the Set Partition decision problem. Set Partitioning asks if a given set of natural numbers can be split into two sets that have equal sum.

Set Partition

Instance: A multi-set $S = \{r_1, \dots, r_n\}$, where $r_i \in \mathbb{N}$ for all $i = 1, \dots, n$.

Solution: A partition of S into sets S_1 and S_2 ; i.e. $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$.

Measure: $m(S) = 1$ if $\sum_{r_i \in S_1} r_i = \sum_{r_j \in S_2} r_j$, and $m(S) = 0$ otherwise.

Proof. Let $S = r_1, \dots, r_n$ be a Set Partition instance. Reduce it to Minimum Bin Packing by setting $C = 1/2 \sum_{j=1}^n r_j$ (half the total sum of elements in S), and considering a bin packing instance of items $S' = \{r_1/C, \dots, r_n/C\}$. If S can be partitioned into two sets of equal sum, then the minimum number of bins necessary for the corresponding S' is 2. On the other hand, if S cannot be partitioned in this way, the minimum number of bins needed for S' is at least 3, as every possible partitioning results in a set with sum greater than C . Therefore, if there were a polytime $(3/2 - \varepsilon)$ -approximation algorithm A , it could be used to solve Set Partition:

- If A (given S and C) returns a solution using at most $(3/2 - \varepsilon)2 = 3 - 2\varepsilon$ bins, then there exists a Set Partition for S .
- If A returns a solution using at least $(3/2 - \varepsilon)3 = 9/2 - 3\varepsilon = 4.5 - 3\varepsilon$ bins, then there is no Set Partition for S .

But for any $\varepsilon \in (0, 3/2)$,

$$3 - 2 < 4.5 - 3\varepsilon.$$

Therefore, this polynomial time algorithm distinguishes between those S that can be partitioned and those that cannot, so $P = NP$. \square

A similar result holds for problems such as MAX-CUT, MAX-SAT and Minimum Vertex Cover. However, unlike the result for Bin Packing, the proofs for these require the introduction of *probabilistically checkable proofs*, which we will be discussed later.

21.3.2 APX

APX is a (presumably) larger class than PTAS; the approximation guarantees for problems in it are strictly weaker. An NP optimization problem Π is in APX simply if there is a polynomial time algorithm A and constant k such that A is a c -approximation to Π .

21.3.2.1 A Few Known Results

It is easy to see that $\text{PTAS} \subseteq \text{APX} \subseteq \text{NPO}$. When one sees new complexity classes and their inclusions, one of the first questions to be asked is: How likely is it that these inclusions could be made into equalities? Unfortunately, it is highly unlikely. The following relationship can be shown between the three approximation classes we have seen.

Theorem 21.19 (Ausiello et al. 1999). $\text{PTAS} = \text{APX} \iff \text{APX} = \text{NPO} \iff P = NP$.

Therefore, if all NP optimization problems could be approximated within a constant factor, then $P = NP$. Further, if all problems that can have constant approximations can be arbitrarily approximated, still $P = NP$. Another way of saying this is: if NP problems are hard to solve, then some of them are hard to approximate as well. Moreover, there exists a *hierarchy* of successively harder-to-approximate problems.

One of the directions stated follows from a theorem of the previous section: earlier, we saw a constant factor approximation to Minimum Bin Packing. However, it does not have a PTAS unless $P = NP$. This shows the direction $\text{PTAS} = \text{APX} \Rightarrow P = NP$. One example of a problem that cannot be in APX unless $P = NP$ is the well-known Minimum TSP:

Minimum Traveling Salesman

Instance: A set $C = \{c_1, \dots, c_n\}$ of cities, and a distance function $d : C \times C \rightarrow \mathbb{N}$.

Solution: A path through the cities, i.e. a permutation $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$.

Measure: The cost of visiting cities with respect to the path, i.e. $\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

It is important to note that when the distances in the problem instances always obey a Euclidean metric, Minimum Traveling Salesman has a PTAS ([Arora 1998](#)). Thus, we may say that it is the generality of possible distances in the above problem

that makes it difficult to approximate. This is often the case with approximability: a small restriction on an inapproximable problem can suddenly turn it into a highly approximable one.

21.3.3 Brief Introduction to PCPs

In the 1990s, the work in probabilistically checkable proofs (PCPs) was *the* major breakthrough in proving hardness results, and arguably in theoretical computer science as a whole. In essence, PCPs only look at a few bits of a proposed proof, using randomness, but manage to capture all of NP. Because the number of bits they check is so small (a constant), when an efficient PCP exists for a given problem, it implies the hardness of *approximately solving* the same problem as well, within some constant factor.

The notion of a PCP arose from a series of meditations on proof-checking using randomness. We know NP represents the class of problems that have *short proofs* we can check efficiently. As far as NP is concerned, all of the checking we do is deterministic. When a proof is correct or incorrect, a polynomial time verifier answers “yes” or “no” with 100 % confidence.

However, what happens when we relax the notion of total correctness to include probability? Suppose we permit the verifier to toss unbiased coins, and have *one-sided error*. That is, now a randomized verifier only accepts a correct proof with probability at least 1/2, but still rejects any incorrect proof it reads. (We call such a verifier a probabilistically checkable proof system.) This slight tweaking of what it means to verify a proof leads to an amazing characterization of NP: all NP decision problems can be verified by a PCP of the above type, which only flips $O(\log n)$ coins and only checks a constant ($O(1)$) number of bits of any given proof! The result involves the construction of highly intricate error-correcting codes. We do not discuss this on a formal level here, but shall cite the above in the notation of a theorem.

Theorem 21.20 (Arora et al. 1998). $PCP[O(\log n), O(1)] = NP$.

One corollary of this theorem is that a large class of approximation problems do not admit a PTAS, in particular, as follows.

Theorem 21.21. *For $\Pi \in \{MAX-Ek-SAT, MAX-CUT, Minimum\ Vertex\ Cover\}$, there exists a c such that Π cannot be c -approximated in polynomial time, unless $P = NP$.*

21.4 Promising Areas for Future Application

21.4.1 Randomized Backtracking and Backdoors

Backtracking is one of the oldest and most natural methods used for solving combinatorial problems. In general, backtracking deterministically can take exponential time. Recent work has demonstrated that many real-world problems can be solved quite rapidly, when the choices made in backtracking are randomized. In particular, problems in practice tend to have small substructures within them. These substructures have the property that once they are solved properly, the entire problem may be solved. The existence of these so-called *backdoors* (Williams et al. 2003) to problems make them very tenable to solution using randomization. Roughly speaking, search heuristics will set the backdoor substructure early in the search, with a significant probability. Therefore, by repeatedly restarting the backtracking mechanism after a certain (polynomial) length of time, the overall runtime that backtracking requires to find a solution is decreased tremendously.

21.4.2 Approximations to Guide Complete Backtrack Search

A promising approach for solving combinatorial problems using complete (exact) methods draws on recent results on some of the best approximation algorithms based on LP relaxations and randomized rounding techniques, as well as on results that uncovered the extreme variance or *unpredictability* in the run $\tilde{\mathcal{O}}(\frac{1}{2})$ time of complete search procedures, often explained by so-called heavy-tailed cost distributions (Gomes et al. 2000). Gomes and Shmoys (2002) propose a complete randomized backtrack search method that tightly couples constraint satisfaction problem (CSP) propagation techniques with randomized LP-based approximations. They use as a benchmark domain a purely combinatorial problem, the quasi-group (or Latin square) completion problem. Each instance consists of an n by n matrix with n^2 cells. A complete quasi-group consists of a coloring of each cell with one of n colors in such a way that there is no repeated color in any row or column. Given a partial coloring of the n by n cells, determining whether there is a valid completion into a full quasi-group is an NP-complete problem (Colbourn 1984). The underlying structure of this benchmark is similar to that found in a series of real-world applications, such as timetabling, experimental design, and fiber optics routing problems (Laywine and Mullen 1998; Kumar et al. 1999).

Gomes and Shmoys compare their results for the hybrid CSP/LP strategy guided by the LP randomized rounding approximation with a CSP strategy and with a LP strategy. The results show that the hybrid approach significantly improves over the pure strategies on hard instances. This suggest that the LP randomized rounding approximation provides powerful heuristic guidance to the CSP search.

21.4.3 Average-Case Complexity and Approximation

Recently, an intriguing thread of theoretical research has explored the connections between the average-case complexity of problems and their approximation hardness ([Feige 2002](#)). For example, it is shown that if *random 3SAT* is hard to solve in polynomial time (under reasonable definitions of *random* and *hard*), then NP-hard optimization problems such as Minimum Bisection are hard to approximate in the worst-case. Conversely, this implies that improved approximation algorithms for some problems could lead to the average-case tractability of others. A natural research question to ask is: does an FPTAS imply average-case tractability, or vice versa? We suspect that some statement of this form might be the case. In our defense, a recent paper ([Beier and Vocking 2003](#)) shows that Random Maximum Integer Knapsack is exactly solvable in expected polynomial time! (Recall that there exists an FPTAS for Maximum Integer Knapsack.)

21.5 Tricks of the Trade

One major initial motivation for the study of approximation algorithms was to provide a new theoretical avenue for analyzing and coping with hard problems. Faced with a brand-new interesting optimization problem, how might one apply the techniques discussed here? One possible scheme proceeds as follows:

1. First, try to prove your problem is NP-hard, or find evidence that it is not! Perhaps the problem admits an interesting exact algorithm, without the need for approximation.
2. Often, a very natural and intuitive idea is the basis for an approximation algorithm. How good is a randomly chosen feasible solution for the problem? (What is the expected value of a random solution?) How about a greedy strategy? Can you define a neighborhood such that local search does well? Is there a relaxation of the problem (where integer solutions are relaxed, and real solutions are allowed) that can be solved efficiently? For many computational problems, there are linear programming relaxations which can be used to approximately solve the original problem.
3. Look for a problem (call it Π) that is related to yours, and is known to have good approximation algorithms. Try to use the algorithms and techniques for solving Π to obtain an approximation algorithm for your problem.
4. Try to prove that your problem cannot be well-approximated, by reducing some hard-to-approximate problem to your problem.

The first, third, and fourth points essentially hinge on one's resourcefulness: one's tenacity to scour the literature (and colleagues) for problems similar to the one at hand, as well as one's ability to see the relationships and reductions which show that a problem is indeed similar.

This chapter has been mainly concerned with the second point. To answer the questions of that point, it is crucial to prove *bounds* on optimal solutions, with respect to the feasible solutions that one's approaches obtain. For minimization (maximization) problems, one will need to prove *lower bounds* (respectively, *upper bounds*) on some optimal solution for the problem. Devising lower (or upper) bounds can simplify the proof tremendously: one only needs to show that an algorithm returns a solution with value at most c times the lower bound to show that the algorithm is a c -approximation.

We have proven upper and lower bounds repeatedly (implicitly or explicitly) in our proofs for approximation algorithms throughout this chapter—it may be instructive for the reader to review each approximation proof and find where we have done it. For example, the greedy vertex cover algorithm (of choosing a maximal matching) works because even an optimal vertex cover covers at least one of the vertices in each edge of the matching. The number of edges in the matching is a lower bound on the number of nodes in a optimal vertex cover, and thus the number of nodes in the matching (which is twice the number of edges) is at most twice the number of nodes of an optimal cover.

21.6 Conclusions

We have seen the power of randomization in finding approximate solutions to hard problems. There are many available approaches for designing such algorithms, from solving a related problem and tweaking its solution (in linear programming relaxations) to constructing feasible solutions in a myopic way (via greedy algorithms). We saw that for some problems, determining an approximate solution is vastly easier than finding an exact solution, while other problems are just as hard to approximate as they are to solve.

In closing, we remark that the study of approximation and randomized algorithms is still a very young (but rapidly developing) field. It is our sincerest hope that the reader is inspired to contribute to the prodigious growth of the subject, and its far-reaching implications for problem solving in general.

Sources of Additional Information

Books on algorithms:

- Data structures and Algorithms ([Aho et al. 1983](#))
- Introduction to Algorithms ([Cormen et al. 2001](#))
- The Design and Analysis of Algorithms ([Kozen 1992](#))
- Combinatorial Optimization: Algorithms and Complexity ([Papadimitriou and Steiglitz 1982](#))

Books on linear programming and duality:

- Linear Programming ([Chvatal 1983](#))
- Linear Programming and Extensions ([Dantzig 1998](#))
- Integer and Combinatorial Optimization ([Nemhauser and Wolsey 1988](#))
- Combinatorial Optimization: Algorithms and Complexity ([Papadimitriou and Steiglitz 1982](#))
- Combinatorial Optimization ([Cook et al. 1988](#))
- Combinatorial Optimization Polyhedra and Efficiency ([Schrijver 2003](#))

Books on approximation algorithms:

- Complexity and Approximation ([Ausiello et al. 1999](#))
- Approximation Algorithms for NP-Hard Problems ([Hochbaum 1997](#))
- Approximation algorithms ([Vazirani 2004](#))
- The Design of Approximation Algorithms ([Williamson and Shmoys 2001](#)), available at <http://www.designofapproxalgs.com/>

One of the most intriguing lines of work in approximation algorithms over the past few years has been the formulation and development of the Unique Games Conjecture of Subhash Khot ([2002](#)). The conjecture asserts that a certain combinatorial problem is hard to approximate. If the conjecture is true, then many simple approximation algorithms (like the 2-approximation for Vertex Cover) are optimal. However, the status of the conjecture is unclear. Recently, there is some interesting evidence (in the form of subexponential-time approximation algorithms) that the Unique Games Conjecture may be false ([Arora et al. 2010](#)).

Books on probabilistic and randomized algorithms:

- An Introduction to Probability Theory and Its Applications ([Feller 1971](#))
- The Probabilistic Method ([Alon and Spencer 2000](#))
- Randomized Algorithms ([Motwani and Raghavan 1995](#))
- The Discrepancy Method ([Chazelle 2001](#))

Surveys:

- Computing Near-Optimal Solutions to Combinatorial Optimization Problems ([Shmoys 1995](#))
- Approximation algorithms via randomized rounding: a survey ([Srinivasan 1999](#))

Courses and lectures notes online:

- Approximability of Optimization Problems, MIT, Fall 99 (Madhu Sudan)
<http://theory.lcs.mit.edu/madhu/FT99/course.html>
- Approximation Algorithms, Fields Institute, Fall 99 (Joseph Cheriyan)
<http://www.math.uwaterloo.ca/jcheriya/App-course/course.html>
- Approximation Algorithms, John Hopkins University Fall 1998 (Lenore Cowen)
<http://www.cs.jhu.edu/cowen/approx.html>
- Approximation Algorithms, Technion, Fall 95 (Yuval Rabani)
<http://www.cs.technion.ac.il/rabani/236521.95.wi.html>

- Approximation Algorithms, Cornell University, Fall 1998 (David Williamson) <http://www.almaden.ibm.com/cs/people/dpw/>
- Approximation Algorithms, Tel Aviv University, Fall 2001 (Uri Zwick) <http://www.cs.tau.ac.il/>
- Approximation Algorithms for Network Problems, Lecture Notes (J.Cheriyam and R.Ravi) <http://www.gsia.cmu.edu/afs/andrew/gsia/ravi/WWW/new-lecnotes.html>
- Randomized algorithms, CMU, Fall 2000 (Avrim Blum) <http://www-2.cs.cmu.edu/afs/cs/usr/avrim/www/Randalgs98/home.html>
- Randomization and optimization by Devdatt Dubhashi <http://www.cs.chalmers.se/dubhashi/ComplexityCourse/info2.html>
- Topics in Mathematical Programming: Approximation Algorithms, Cornell University, Spring 99 (David Shmoys) <http://www.orie.cornell.edu/or739/index.html>
- Course notes on online algorithms, randomized algorithms, network .ows, linear programming, and approximation algorithms (Michel Goemans) <http://www-math.mit.edu/goemans/>
- Lecture notes, www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/ (hosted by CMU)

Main conferences covering the approximation and randomization topics:

- IPCO—Integer Programming and Combinatorial Optimization
- ISMP—International Symposium on Mathematical Programming
- FOCS—Annual IEEE Symposium on Foundation of Computer Science
- SODA—Annual ACM-SIAM Symposium on Discrete Algorithms
- STOC—Annual ACM Symposium on Theory of Computing
- RANDOM—International Workshop on Randomization and Approximation Techniques in Computer Science
- APPROX—International Workshop on Approximation Algorithms for Combinatorial Optimization Problems

References

- Agrawal M, Kayal N, Saxena N (2004) PRIMES is in P. Ann Math 160:781–793
- Aho AV, Hopcroft JE, Ullman JD (1979) Computers and intractability: a guide to NP-completeness. Freeman, San Francisco
- Aho AV, Hopcroft JE, Ullman JD (1983) Data structures and algorithms. Computer science and information processing series. Addison-Wesley, Reading
- Alon N, Spencer J (2000) The probabilistic method. Wiley, New York
- Arora S (1998) Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. J ACM 45:753–782
- Arora S, Lund C, Motwani R, Sudan M, Szegedy M (1998) Proof verification and the hardness of approximation problems. J ACM 45:501–555

- Arora S, Barak B, Steurer D (2010) Subexponential algorithms for unique games and related problems. In: Proceedings of the IEEE symposium on foundations of computer science, Las Vegas, pp 563–572
- Ausiello G, Crescenzi P, Gambosi G, Kann V, Marchetti-Spaccamela A, Protasi M (1999) Complexity and approximation. Springer, Berlin
- Beier R, Vocking B (2003) Random knapsack in expected polynomial time. *J Comput Syst Sci* 69:306–329
- Chazelle B (2001) The discrepancy method. Cambridge University Press, Cambridge/New York
- Chvatal V (1979) A greedy heuristic for the set-covering. *Math Oper Res* 4:233–235
- Chvatal V (1983) Linear programming. Freeman, San Francisco
- Clay Mathematics Institute (2003) The millenium prize problems: P vs NP. <http://www.claymath.org/>
- Colbourn C (1984) The complexity of completing partial latin squares. *Discret Appl Math* 8:25–30
- Cook W, Cunningham W, Pulleyblank W, Schrijver A (1988) Combinatorial optimization. Wiley, New York
- Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms. MIT, Cambridge
- Dantzig G (1998) Linear programming and extensions. Princeton University Press, Princeton
- Dantzig GB, Ford LR, Fulkerson DR (1956) A primal-dual algorithm for linear programs. In: Kuhn HW, Tucker AW (eds) Linear inequalities and related systems, Annals of Mathematics Study No. 38, Princeton University Press, Princeton, New Jersey, pp 171–181
- Egervary E (1931) Matrixok kombinatorius tulajdonsagairol. *Matematikai es Fizikai Lapok* 38:16–28
- Feige U (2002) Relations between average case complexity and approximation complexity. In: Proceedings of the ACM symposium on theory of computing, Montreal
- Feller W (1971) An introduction to probability theory and its applications. Wiley, New York
- Garey MR, Graham RL, Ulman JD (1972) Worst case analysis of memory allocation algorithms. In: Proceedings of the 4th ACM symposium on theory of computing, Denver, pp 143–150
- Goemans MX, Williamson DP (1995) Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J ACM* 42:1115–1145
- Goemans M, Williamson DP (1997) The primal-dual method for approximation algorithms and its application to network design problems. In: Hochbaum DS (ed) Approximation algorithms for NP-hard problems. PWS, Boston
- Goldwasser S, Kilian J (1986) Almost all primes can be quickly certified. In: Proceedings of the annual IEEE symposium on foundations of computer science, Toronto, pp 316–329

- Gomes CP, Shmoys D (2002) The promise of LP to boost CSP techniques for combinatorial problems. In: Jussien N, Laburthe F (eds) Proceedings of the CP-AI-OR 2002, Le Croisic, France, pp 291–305. <http://www.emn.fr/z-info/cpaior/>
- Gomes CP, Selman B, Kautz H (1998) Boosting combinatorial search through randomization. In: Proceedings of the AAAI 1998, Madison, Wisconsin, pp 431–437. <http://www.aaai.org/Conferences/AAAI/aaai98.php>
- Gomes C, Selman B, Crato N, Kautz H (2000) Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J Autom Reason* 24:67–100
- Graham RL (1966) Bounds for certain multiprocessing anomalies. *Bell Syst Tech J* 45:1563–1581
- Hochbaum DS (1982) Approximation algorithms for the set covering and vertex cover problem. *SIAM J Comput* 11:555–556
- Hochbaum DS (1983) Efficient bounds for the stable set, vertex cover and the set packing problems. *Discret Appl Math* 6:243–254
- Hochbaum DS (ed) (1997) Approximation algorithms for NP-hard problems. PWS, Boston
- Johnson DS (1974) Approximation algorithms for combinatorial problems. *J Comput Syst Sci* 9:256–278
- Khot S (2002) On the power of unique 2-prover 1-round games. In: Proceedings of the 34th annual ACM symposium on theory of computing, Montreal, pp 767–775
- Khot S, Regev O (2003) Vertex cover might be hard to approximate within $2 - \epsilon$. In: Proceedings of the IEEE conference on computational complexity, Aarhus. *J Comput Syst Sci* 74:335–349
- Kirkpatrick S, Gelatt C, Vecchi M (1983) Optimization by simulated annealing. *Science* 220:671–680
- Kozen D (1992) The design and analysis of algorithms. Springer, New York
- Kuhn HW (1955) The Hungarian method for the assignment problem. *Nav Res Q* 2:83–97
- Kumar SR, Russell A, Sundaram R (1999) Approximating latin square extensions. *Algorithmica* 24:128–138
- Laywine C, Mullen G (1998) Discrete mathematics using latin squares. Discrete mathematics and optimization series. Wiley-Interscience, New York
- Lovasz L (1975) On the ratio of optimal integral and fractional covers. *Discret Math* 13:383–390
- Motwani R, Raghavan P (1995) Randomized algorithms. Cambridge University Press, Cambridge/New York
- Nemhauser G, Wolsey L (1988) Integer and combinatorial optimization. Wiley, New York
- Papadimitriou C, Steiglitz K (1982) Combinatorial optimization: algorithms and complexity. Prentice-Hall, Englewood Cliffs
- Rabin M (1980) Probabilistic algorithm for testing primality. *J Number Theor* 12:128–138
- Schrijver A (2003) Combinatorial optimization polyhedra and efficiency. Springer, Berlin

- Shmoys D (1995) Computing near-optimal solutions to combinatorial optimization problems. In: Cook W, Lovasz L, Seymour P (eds) Combinatorial optimization. DIMACS series. AMS, Providence, pp 355–396
- Solovay R, Strassen V (1977) A fast Monte Carlo test for primality. SIAM J Comput 6:84–86
- Srinivasan A (1999) Approximation algorithms via randomized rounding: a survey. In: Karonskin M, Promel HJ (eds) Lectures on approximation and randomized algorithms. Series in advanced topics in mathematics. Polish Scientific Publishers PWN, Warsaw, pp 9–71
- Vazirani V (2004) Approximation algorithms. Springer, Berlin
- Williams R, Gomes CP, Selman B (2003) Backdoors to typical case complexity. In: Proceedings of the IJCAI, Acapulco, pp 173–1178
- Williamson DP, Shmoys DB (2011) The design of approximation algorithms. Cambridge University Press, New York
- Wolsey LA (1980) Heuristic analysis, linear programming and branch and bound. Math Programm 28:271–287

Chapter 22

Fitness Landscapes

Colin R. Reeves

22.1 Historical Introduction

One of the most commonly used metaphors to describe the process of heuristic methods such as local search in solving a combinatorial optimization problem is that of a *fitness landscape*. However, describing exactly what we mean by such a term is not as easy as might be assumed. Indeed, many cases of its usage in both the biological and optimization literature reveal a rather serious lack of understanding.

The landscape metaphor appears most commonly in work related to evolutionary algorithms, where it is customary to trace the usage of the term back to a paper by the population geneticist Sewall Wright (1932), although Haldane had already introduced a similar notion (Haldane 1931). The metaphor has become pervasive, being cited in many biological texts that discuss evolution.

Wright's original idea of a fitness landscape was somewhat ambiguous. It appears that what he initially had in mind concerned within-species variation where the axes of a search space represented unspecified gene combinations, but Dobzhansky's subsequent enthusiastic use of the metaphor (Dobzhansky 1951) seems to have established the consensus view of the axes of the search landscape as the frequency of a particular allele of a particular gene in a particular population. This can be seen in many textbooks on the subject of evolution, such as Ridley (1993). In the hands of Simpson, who seems to have thought primarily in terms of phenotypic characters (Simpson 1953), the story became even more highly developed, although even more divorced from empirical reality. Despite Wright's later attempts to clarify the situation (Wright 1967, 1988), the ambiguity remains. There is thus an interesting paradox in evolutionary biology: according to Futuyma (1998, p. 403),

[The] adaptive landscape is probably the most common metaphor in evolutionary genetic[s]

C.R. Reeves (✉)

Department of Mathematics and Control Engineering, Coventry University, Coventry, UK
e-mail: c.reeves@coventry.ac.uk

yet nobody seems sure what exactly is the reality to which the metaphor is supposed to relate! However, it remains extremely popular: Dawkins' recent book ([Dawkins 1996](#)), for example, makes considerable use of the notion, as its title *Climbing Mount Improbable* suggests.

Although we may have a vague idea of what the search space is, it is rather harder to define any axes for such a search space, as we have seen. Moreover, fitness in evolutionary biology is also a rather slippery concept. It is discussed as if there is some objective a priori measure, yet as usually defined, *fitness* concerns an organism's reproductive success, which can only be measured *a posteriori*.¹ Add to this the confusion over what the search space axes represent, and it becomes almost impossible to relate them to some quantifiable measure of fitness. It is thus generally dealt with by prestidigitation, and so, for all its popularity, the idea of a fitness landscape in biology is a mirage, displaying what is to a mathematician a distressing lack of rigor. Happily, some biologists agree, as in the cogent arguments against the hand-waving approach in [Eldredge and Cracraft \(1980\)](#). More recently, [Skipper \(2004\)](#) has argued that it still has a certain value in illustrating behavioral aspects of evolutionary theories, but [Kaplan \(2008\)](#) has suggested that it should be abandoned completely.

A more serious approach was foreshadowed by [Eigen \(1983\)](#) and [Eigen et al. \(1989\)](#). In his work on viruses, he introduced the concept of a quasi-species—a group of similar sequences. Each sequence S_k is a string of symbols drawn from some alphabet, the natural one to consider for viruses being the RNA bases adenine, cytosine, guanine and uracil—{A, C, G, U}. Differences in members of the quasi-species correspond to point mutations—replacement of one symbol by another one.

This interpretation falls somewhat short of the grand ideas in the popular biology textbooks, but it does make a formal mathematical development of the concept of a fitness landscape much more feasible, and following pioneering work by [Weinberger \(1990\)](#) in particular, a fairly complete formal statement of landscape theory was proposed by [Stadler \(1995\)](#). His paper is a sustained development and exploration of landscape concepts, requiring fairly sophisticated mathematics. Recent work has developed and explicated these ideas further ([Reidys and Stadler 2002](#)), but some quite extensive mathematical knowledge is still needed in order to appreciate it fully. In the expectation that the mathematical background of the readers of this volume will be somewhat variable, this tutorial will try to survey some of the themes most relevant to combinatorial optimization, without using advanced mathematical ideas. Some basic ideas of set theory, matrix algebra and functional analysis will be required, but the more complex ideas found in [Reidys and Stadler \(2002\)](#) will not be covered. Illustrative numerical examples will also be used at key points in an attempt to aid understanding.

¹ The *Oxford Dictionary of Biology* defines fitness as “The condition of an organism that is well adapted to its environment, as measured by its ability to reproduce itself.”

22.2 Combinatorial Optimization

We can define combinatorial optimization problems as follows: we have a discrete search space \mathcal{X} , and a function

$$f : \mathcal{X} \mapsto \mathbb{R}.$$

The general problem is to find

$$x^* = \arg \max_{x \in \mathcal{X}} f.$$

where x is a vector of *decision variables* and f is the *objective function*. (Of course, minimization can also be the aim, but the modifications are always obvious). In the field of evolutionary algorithms, the function f is often called the *fitness*; hence the associated landscape is a *fitness landscape*. The vector \mathbf{x}^* is a global optimum: that vector which is the *fittest* of all. (In some problems, there may be several global optima—different vectors of equal fitness.)

With the idea of a fitness landscape comes the idea that there are also many local optima or false peaks, in which a search algorithm may become trapped without finding the global optimum. In continuous optimization, notions of continuity and concepts associated with the differential calculus enable us to characterize quite precisely what we mean by a landscape, and to define the idea of an optimum. It is also convenient that our own experience of hill climbing in a three-dimensional world gives us analogies to ridges, valleys, basins, watersheds, etc., which help us to build an intuitive picture of what is needed for a successful search, even though the search spaces that are of interest often have dimensions many orders of magnitude higher than three.

However, in the continuous case, the landscape is determined only by the fitness function, and the ingenuity needed to find a global optimum consists in trying to match a technique to this single landscape. There is a major difference when we come to discrete optimization, which leads to mistaken *explanations* of the efficacy of this or that local search metaheuristic: sometimes it is asserted, for example, that a given technique (e.g. using a Gray code for 0–1 vectors) allows search in *more remote* parts of the landscape, when it would be more accurate to say that the landscape has been *remodeled* by changing distances. Indeed, we really should not use the term *landscape* unless we have first defined the topological relationships of the points in the search space \mathcal{X} . Unlike the continuous case, we have some freedom to specify these relationships, and in fact, that is precisely what we do when we decide to use a particular technique.

22.2.1 An Example

In practice, one of the most commonly used search methods for a combinatorial optimization problem is *neighborhood search*. This idea is at the root of modern

metaheuristics such as simulated annealing and tabu search—as well as being much more involved in the methodology of genetic algorithms than is sometimes realized.

A *neighborhood structure* is generated by using an operator that transforms a given vector x into a new vector x' . For example, if the solution is represented by a binary vector (as is often the case for genetic algorithms, for instance), a simple neighborhood might consist of all vectors obtainable by *flipping* one of the bits. The *bit flip* neighbors of (00000), for example, would be

$$\{(10000), (01000), (00100), (00010), (00001)\}.$$

Consider the problem of maximizing a simple function

$$f(z) = z^3 - 60z^2 + 900z + 100$$

where the solution z is required to be an integer in the range $[0, 31]$. Regarding z for a moment as a continuous variable, we have a smooth unimodal function with a single maximum at $z = 10$ —as is easily found by calculus. Since this solution is already an integer, this is undoubtedly the most efficient way of solving the problem.

However, suppose we chose instead to represent z by a binary vector x of length 5. By decoding this binary vector as an integer it is possible to evaluate f , and we could then use neighborhood search, for example, to search over the binary hypercube for the global optimum using some form of hill-climbing strategy.

This discrete optimization problem turns out to have four optima (three of them local) when the bit flip operator is used. If a *steepest ascent* strategy is used (i.e. the *best* neighbor of a given vector is identified before a move is made) the local optima are as shown in Table 22.1. Also shown are the *basins of attraction*—the set of initial points from which the search leads to a specified optimum. For example, if we start the search at any of the points in the first column, and follow a strict best-improvement strategy, the search will finish up at the global optimum. However, if a *first improvement* strategy is used (where the first change that leads uphill is accepted without ascertaining if a still better one exists), the basins of attraction are rather different, as shown in Table 22.2.

In fact, there are even more complications: in Table 22.2, the order of searching the components of the vector has been termed *forward* (left-to-right). If the search is made in the reverse direction (right-to-left) the basins of attraction are different, as shown in Table 22.3.

Thus, by using flipping with this binary representation, we have created local optima that did not exist in the integer version of the problem. Further, although the optima are still the same, the chances of reaching a *particular* optimum can be seriously affected by a change in hill-climbing strategy.

Moreover, the bit flip operator is not the only mechanism for generating neighbors. An alternative neighborhood could be defined as follows: for $k = 1, \dots, 5$, flip bits $\{k, \dots, 5\}$. Thus, the neighbors of (00000), for example, would now be

$$\{(11111), (01111), (00111), (00011), (00001)\}.$$

Table 22.1 Local optima and basins of attraction for steepest ascent with the bit flip operator in the case of a simple cubic function. The bracketed figures are the fitnesses of each local optimum

Local optimum	0 1 0 1 0 (4100)	0 1 1 0 0 (3988)	0 0 1 1 1 (3803)	1 0 0 0 0 (3236)
Basin	0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1 0 1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0 0 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1	0 0 1 0 0 0 1 1 0 0 1 1 1 0 0 1 0 1 1 0 1 0 1 1 1	0 0 1 1 0 0 0 1 1 1 1 0 1 1 0 1 0 1 1 1	1 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 1 1 0 1 0 0

We shall call this the CX operator, and it creates a very different landscape. In fact, there is now only a single global optimum (01010); *every* vector is in its basin of attraction. This illustrates the point that it is not merely the choice of a binary representation that generates the landscape—the search operator needs to be specified as well.

Incidentally, there are two interesting facts about the CX operator. Firstly, it is closely related to the one-point crossover operator frequently used in genetic algorithms. (CX is actually shorthand for *complementary crossover*.) Secondly, if the 32 vectors in the search space are re-coded using a *Gray* code, it is easy to show that the bit-flip neighbors of a point in Gray-coded space are identical to those in the original binary-coded space under CX. This is an example of an *isomorphism* of landscapes.²

² An isomorphism in mathematics refers to mappings between mathematical objects that preserve structure. It comes from the Greek *iso* (equal) and *morphe* (shape). For example, two graphs are isomorphic if there is a one-to-one mapping σ between their respective sets of vertices such that for every edge (x, y) of one graph, $(\sigma(x), \sigma(y))$ is an edge of the other.

Table 22.2 Local optima and basins of attraction for first improvement (forward search) using the bit flip operator

Local optimum	0 1 0 1 0 (4100)	0 1 1 0 0 (3988)	0 0 1 1 1 (3803)	1 0 0 0 0 (3236)
Basin	0 0 1 0 1 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1 0 1 1 0 1 0 1 1 0 0 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0	0 0 1 0 0 0 1 0 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1 1	0 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 1

22.3 Mathematical Characterization

Now that some of the typical features of a landscape have been illustrated, we can provide a mathematical characterization. We define a landscape \mathcal{L} for the function f as a triple $\mathcal{L} = (\mathcal{X}, f, d)$ where d denotes a distance measure $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+ \cup \{\infty\}$ for which it is required that, $\forall s, t, u \in \mathcal{X}$,

$$d(s, t) \geq 0; \quad d(s, t) = 0 \Leftrightarrow s = t; \quad d(s, u) \leq d(s, t) + d(t, u).$$

Note that we do not need to specify the representation explicitly, since this is assumed to be implied in the description of \mathcal{X} . We have also decided, for the sake of simplicity, to ignore questions of search strategy and other matters in the definition of a landscape, unlike the more comprehensive definition of [Jones \(1995\)](#), for example.

This definition says nothing about how the distance measure arises. In fact, for many cases a *canonical* distance measure can be defined. Often, this is symmetric, i.e. $d(s, t) = d(t, s) \forall s, t \in \mathcal{X}$, so that d also defines a *metric* on \mathcal{X} . This is clearly a nice property, although it is not essential.

22.3.1 Neighborhood Structure

The distance measure is typically related to the neighborhood structure. Every solution $x \in \mathcal{X}$ has an associated set of *neighbors*, $N_{\omega}(x) \subset \mathcal{X}$, called the neighborhood of x . This neighborhood is generated by applying an operator ω to a vector s in order

Table 22.3 Local optima and basins of attraction for first improvement (reverse search) using the bit flip operator

Local optimum	0 1 0 1 0 (4100)	0 1 1 0 0 (3988)	0 0 1 1 1 (3803)	1 0 0 0 0 (3236)
Basin	0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1	0 1 1 0 0 0 1 1 0 1 0 1 1 1 0 0 1 1 1 1	0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1	1 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 0 1 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1 1 1 1 0 1 1 1 1 1 0 0 0 1 1 0 0 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1

to transform it into a vector t . What we may call a canonical distance measure d_{ω} is that induced by ω whereby

$$t \in N_{\omega}(s) \Leftrightarrow d_{\omega}(s, t) = 1.$$

The distance between non-neighbors is defined as the length of the shortest path between them (if one exists). The operator ω generally takes a parameter, which means that it is technically a one-to-many function, able to generate many neighbors from one initial vector. The size of the neighborhood will be denoted by n .

For example, if X is the binary hypercube $\{0, 1\}^{\ell}$, the bit flip operator can be defined as

$$\phi(i) : \{0, 1\}^{\ell} \times \mathbb{Z} \rightarrow \{0, 1\}^{\ell} \quad \begin{cases} z'_i = 1 - z_i \\ z'_k = z_k \quad \text{if } i \neq k, \end{cases}$$

where z is a binary vector of length ℓ , and i is the parameter specifying the bit to be flipped. It is clear that the distance metric induced by ϕ is the well-known Hamming distance

$$d_H(x, y) = \sum_{i=1}^{\ell} [x_i \neq y_i],$$

where the square brackets $[expr]$ denote an *indicator* function, which takes the value 1 if the logical expression $expr$ is true and 0 otherwise. Thus we could describe this landscape as a Hamming landscape (with reference to its distance measure), or as the bit-flip landscape (with reference to the operator). Similarly, we can define the CX operator as

$$\gamma(i) : \{0, 1\}^\ell \times \mathbb{Z} \rightarrow \{0, 1\}^\ell \quad \begin{cases} z'_k = 1 - z_k & \text{for } k \geq i \\ z'_k = z_k & \text{otherwise.} \end{cases}$$

The distance measure induced here is clearly more complicated than the Hamming landscape, and cannot be described by a simple function. In both of these cases the size of the neighborhood is $n = \ell$.

As an example of an asymmetric distance measure, consider the case where \mathcal{X} is Π_m , the space of permutations of length m , which is often relevant for scheduling problems. A familiar neighborhood is defined by the *forward shift* operator, taking two parameters in this case:

$$\mathcal{FSH}(i, j) : \Pi_m \times \mathbb{Z} \times \mathbb{Z} \rightarrow \Pi_m \quad \begin{cases} \pi'_{k-1} = \pi_k & \text{for } j < k \leq i \\ \pi'_i = \pi_j & \\ \pi'_k = \pi_k & \text{otherwise.} \end{cases}$$

The neighbors of (1234), for example, would be

$$\{(2134), (2314), (2341), (1324), (1342), (1243)\}.$$

Note that the size of this neighborhood is $n = \binom{m}{2}$. It is easily seen that (1234), however, is not a neighbor of (2314), (2341) or (1342), so \mathcal{FSH} is not symmetric. Other neighborhood operators (for example, *exchange*, where two items in the sequence are swapped) induce different distance measures, so there may be advantages in choosing operator-independent distance measures (Reeves 1999) for practical comparisons.

Distance measures may become even more complicated: for instance, in the problem of biological sequence comparison (RNA, DNA and protein sequences, see Waterman 1995), it is common to compare sequences in terms of the minimal number of genetic operations necessary to convert one string into another (the “string edit” distance). Thus, even finding the distance measure is effectively an optimization problem.

22.3.2 Local Optima

We can now give a formal statement of a fundamental property of fitness landscapes: for a landscape $\mathcal{L} = (\mathcal{X}, f, d)$, a vector $x^o \in \mathcal{X}$ is *locally optimal* if

$$f(x^o) > f(t) \quad \forall t \in N(x^o).$$

We shall denote the set of such optima as \mathcal{X}^o , and the set of *global optima* (recall that we allow the possibility of more than one) as \mathcal{X}^* where the vector $x^* \in \mathcal{X}^o$ is a global optimum if

$$f(x^*) \geq f(x^o) \quad \forall x^o \in \mathcal{X}^o.$$

Landscapes that have only one local (and thus also global) optimum are commonly called *unimodal*, while landscapes with more than one local optimum are said to be *multimodal*.

The number of local optima in a landscape clearly has some bearing on the difficulty of finding the global optimum. In our previous example, it is clearly more difficult to find the global optimum using bit-flipping than if we used CX. However, it is not the only indicator: in our example the steepest-ascent strategy increased the chance of finding the global optimum, since there were more initial solutions that led to the global optimum than under first-improvement.

22.3.3 Basins of Attraction

We can also now define more precisely the idea of a *basin of attraction*. Neighborhood search can be interpreted as a function

$$\mu : \mathcal{X} \mapsto \mathcal{X}^o,$$

where if x is the initial point, $\mu(x)$ is the optimum that it reaches. With this in mind, we can define the basin of attraction of x^o as the set

$$B(x^o) = \{x : \mu(x) = x^o\}.$$

The problem is that $B(x^o)$ is not independent of the search strategy, as the example of Sect. 22.2.1 demonstrated. In fact, it is only well defined for the case of steepest ascent. For other search strategies, such as first improvement, the order of searching may be highly influential. Our example showed that the basin of attraction of the global optimum was much larger for steepest ascent than for the other strategies, but it is possible to find examples where the opposite is the case.

22.3.4 Plateaux

The definition of a local optimum above ignored the possibility that in many problems neighboring points may have the *same* fitness value. We could ignore this by assuming that fitnesses can be *jittered* by adding a small random perturbation, but in some cases the issue causes considerable difficulty to a local search, as there may be extensive *plateaux*. A plateau \mathcal{P} is a set of points $\{x_i\}$ such that $f(x_i) = c$, a constant, and if $s, t \in \mathcal{P}$, there is a path

$$s = x_1, x_2, \dots, x_k = t \quad \text{such that } x_{i+1} \in N(x_i) \text{ and } x_i \in \mathcal{P} \quad \forall i.$$

In fact, this situation can be investigated further: in Frank et al. (1997), a whole taxonomy of plateaux is discussed, but in this brief chapter the subject will not

be explored further. Suffice it to say that understanding plateaux and methods to circumvent them is a very important part of finding efficient techniques for problems such as MAXSAT.

22.3.5 Graph Representation

Neighborhood structures are clearly just another way of defining a graph Γ , which can be described by its $(n \times n)$ adjacency matrix A . The elements of A are given by $a_{ij} = 1$ if the indices i and j represent neighboring vectors, and $a_{ij} = 0$ otherwise. For example, the graph induced by the bit flip ϕ on binary vectors of length 3 has adjacency matrix

$$A_\phi = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

where the vectors are indexed from 0 to 7 in the usual binary-coded integer order (i.e. (000), (001), etc.). By way of contrast, the adjacency matrix for the CX operator is

$$A_\gamma = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

It is simply demonstrated that permuting the rows and columns so that they are in the order 0, 1, 3, 2, 7, 6, 4, 5 reproduces the adjacency matrix A_ϕ —another way of demonstrating the isomorphism mentioned earlier. In other words,

$$P^{-1} A_\phi P = A_\gamma$$

where P is the associated permutation matrix of the binary-to-Gray transformation. It is also clear that the eigenvalues and eigenvectors are the same (up to a permutation).

As a final example, we may consider the adjacency matrix for \mathcal{FSH} in the space Π_3 :

$$A_{\mathcal{FSH}} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix},$$

where the permutations have been indexed in lexicographic order $(123), \dots, (321)$. The lack of symmetry in the distance measure is of course reflected in an asymmetric matrix.

22.3.6 Laplacian Matrix

The *graph Laplacian* Δ is defined as

$$\Delta = A - D,$$

where D is a diagonal matrix such that d_{ii} is the degree of vertex i . Usually, these matrices are vertex-regular and $d_{ii} = n \forall i$, so that

$$\Delta = A - nI.$$

This notion recalls that of a Laplacian operator in the continuous domain; the effect of this matrix, applied at the point s to the fitness function f is

$$\Delta f(s) = \sum_{t \in N(s)} (f(t) - f(s)),$$

so it functions rather like a finite-differencing operator in the continuous domain. In particular, $\Delta f(s)/n$ is the average difference in fitness between the vector s and its neighbors. [Grover \(1992\)](#) observed that the landscapes of several combinatorial optimization problems satisfy a difference equation of the form

$$\Delta f + \frac{C}{n} f = 0,$$

where C is a problem-specific constant. This is essentially a discrete analog of the *wave equation* familiar in physics. From this it can be deduced that *all* local optima are better (i.e. larger, in the case of maximization) than the mean \bar{f} over all points on the landscape. Furthermore, it can also be shown that under mild conditions on the nature of the fitness function, the time taken by neighborhood search to find a local maximum is $O(n \log_2[f_{\max}/\bar{f}])$ where f_{\max} is the fitness of a global maximum. (Similar results hold, mutatis mutandis, for minimization problems.)

22.3.7 Graph Eigensystem

In the usual way, we can define eigenvalues and eigenvectors of the matrices associated with the graph Γ . The set of eigenvalues is called the *spectrum* of the graph. For an $n \times n$ matrix A the spectrum is

$$(\lambda_0 \lambda_1 \dots \lambda_{n-1}),$$

where λ_i is the i th eigenvalue, ranked in (weakly) descending order. Similarly, the spectrum of the Laplacian is

$$(\mu_0 \mu_1 \dots \mu_{n-1}),$$

where, again, μ_i is the i th eigenvalue, ranked this time in (weakly) ascending order. For a regular connected graph it can be shown that

$$\mu_i = n - \lambda_i \quad \forall i.$$

Further, from the corresponding eigenvectors the *eigenfunctions* $\{\varphi_i\}$ can be formed, and f can be expanded as

$$f(s) = \sum_i a_i \varphi_i(s).$$

This is sometimes called a Fourier expansion ([Stadler and Wagner 1998](#)).

Unfortunately, the size of these graphs rapidly becomes very large. However, graphs can often be partitioned in a way that makes it possible to reduce the scale of the problem. This enables the formation of a *collapsed matrix* \tilde{A} whose eigenvalues are the *distinct* eigenvalues of A , with multiplicities given by the cardinalities of the partitions. (Relevant mathematical details may be found in the books by [Biggs \(1993\)](#) and [Godsil \(1993\)](#).) If the diameter of such a graph is δ , the number of distinct eigenvalues is only $\delta + 1$, so a considerable reduction in size is possible—at least in principle.

Similarly, the Fourier expansion can be partitioned into a sum

$$f(s) = \sum_p \beta_p \tilde{\Phi}_p(s)$$

over the distinct eigenvalues of Δ . The corresponding values

$$|\beta_p|^2 = \sum |a_k|^2$$

(where the sum is over the coefficients that correspond to the p th distinct eigenvalue) form the *amplitude spectrum*, which expresses the relative importance of different components of the landscape.

22.3.8 Elementary Landscapes

Ideally, such mathematical characterizations could be used to aid our understanding of the important features of a landscape, and so help us to exploit them in designing search strategies. One step in this direction starts from the connection between Grover's observation and Stadler's rather more obscure results (Stadler 1995). Stadler designated landscapes where f is itself an eigenfunction of the "wave equation" as *elementary* landscapes.

In Whitley et al. (2008), it is shown that from a practical point of view, an elementary landscape arises when the objective function is *partially decomposable*. This occurs when the function f is formed as a sum of subfunctions that each involve only a subset of the relevant components of the objective function. Often, such a property is already used in implementing local search: evaluating the cost of a move implied by applying the operator ω does not necessarily entail evaluating f itself. (The TSP is the most obvious case in point: a 2-opt move disconnects the tour, but most of the inter-city distances are unchanged and do not need to be re-calculated.) In such cases, Whitley and his colleagues show, useful information can be gained from partial evaluation of neighborhoods.

Another interesting implication of this analysis is for landscapes where f is *not* partially decomposable. Problems such as flowshop and jobshop scheduling, for example, may have fundamentally different performance characteristics from those problems whose neighborhoods can induce elementary landscapes.

However, apart from the implications of Grover's equation, it is possible to carry out further analytical studies only for small graphs or graphs with a special structure, as illustrated for example, in Stadler (1995). For the important case of the Hamming landscape of a binary search space, analytical results for the graph spectrum show that the eigenvectors are thinly disguised versions of the familiar Walsh functions.³ For the case of recombinative operators the problem is considerably more complicated, and necessitates the use of *P-structures* (Stadler and Wagner 1998). The latter are essentially generalizations of graphs in which the mapping is from pairs of *parents* (x, y) to the set of possible strings that can be generated by their recombination. However, it can be shown that for some *recombination landscapes* (such as that arising from the use of uniform crossover) the eigenvectors are once more the Walsh functions. Whether this is also true in the case of one- or two-point crossover, for example, is not known, but Stadler and Wagner conjecture that it is. In view of the close relationship between the bit-flip and CX landscapes as demonstrated above, it would not be surprising if this is a general phenomenon. However, to obtain these results, some assumptions have to be made—such as a uniform distribution of parents—that are unlikely to be true in a specific finite realization of a genetic search.

³ For readers who are unfamiliar with Walsh functions, they are digital analogs of trigonometrical functions, forming an orthonormal set of rectangular waveforms. An introduction to their uses in the analysis of optimization methods can be found in Reeves and Rowe (2002).

In the case of the bit-flip landscape, the distinct eigenvalues correspond to sets of Walsh coefficients of different orders, and the amplitude spectrum is exactly the set of components of the *epistasis variance* associated with other attempts to measure problem difficulty (see [Reeves and Rowe 2002](#) for a review). For the cubic function of Sect. 22.2.1, the components of variance for the different orders of Walsh coefficients can be shown to be $(0.387, 0.512, 0.101, 0, 0)$ respectively; i.e. 61.3 % of the variation in the landscape is due to interactions of order two and three. This is consistent with the relatively poor performance of the bit-flip hill climber.

Of course, the eigenvalues and eigenvectors are exactly the same (up to a permutation) for the CX landscape of this function, and the set of values for the Walsh coefficients in the Fourier decomposition is also the same. However, the effect of the permutation inherent in the mapping from the bit flip landscape to the CX landscape is to re-label some of the vertices of the graph, and hence some of the Walsh coefficients. Thus some coefficients that previously referred to linear effects now refer to interactions, and vice versa. Taking the cubic function as example again, the components of variance or amplitude spectrum become $(0.771, 0.174, 0.044, 0.011, 0.000)$. We see that the linear effects now predominate (77.1 %), and this is consistent with the fact that the hill climber in the CX landscape always finds the optimum of this function.

Table 22.4 Illustration of the different groupings of the Walsh coefficients associated with the bit flip, CX and recombination landscapes

Binary					Binary				
Index	coding	Bit flip	CX	Recom	Index	coding	Bit flip	CX	Recom
0	0000	0	0	0	8	1000	1	2	1
1	0001	1	1	1	9	1001	2	3	4
2	0010	1	2	1	10	1010	2	4	3
3	0011	2	1	2	11	1011	3	3	4
4	0100	1	2	1	12	1100	2	2	2
5	0101	2	3	3	13	1101	3	3	4
6	0110	2	2	2	14	1110	3	2	3
7	0111	3	1	3	15	1111	4	1	4

22.3.9 Recombination Landscapes

If we look at the recombination landscapes derived from the P-structures of [Stadler and Wagner \(1998\)](#), we find that once again the Walsh coefficients are obtained, but labeled in yet another way. The coefficients in the bit flip and CX landscapes are grouped according to the number of 1s in their binary- and Gray-coded index representations respectively. However, in a recombination landscape—such as that generated by 1-point crossover—it is the *separation* between the outermost 1-bits that defines the groupings. Table 22.4 shows the groupings for a 4-bit problem.

Several things can be seen from this table: firstly, the linear Walsh coefficients (and hence the linear component of epistasis variance) are the same in both the bit flip and the recombination landscapes. Secondly (as already explained), the coefficients in the CX landscape are simply a re-labeling of those in the bit flip landscape. Thirdly, the coefficients in the recombination landscape do not form a natural grouping in terms of interactions, and consequently the different components of variance for the recombination landscape do not have a simple interpretation as due to interactions of a particular order.

22.3.10 Summary

This section has set out some of the basic mathematics necessary for the analysis of landscapes. As has probably become obvious, the details can require an extensive mathematical knowledge. Furthermore, the full analysis of a particular landscape (i.e. its eigensystem) may need the gathering of a large amount of empirical information, perhaps equivalent to a complete knowledge of the fitness function at all points of the search space! Landscape analysis in such cases can be no more than *a posteriori* justification (or lack of it!) for the choice of a particular neighborhood. Further discussion on some of these points may be found in [Reeves and Rowe \(2002\)](#).

While it is undeniably useful that we can construct mathematical techniques to help us neatly summarize certain facts about a landscape, we must also recognize that there are other features—possibly very important ones—that are not captured by these methods. In the simple example of the cubic function we have seen that the search strategy adopted can make a big difference to the likelihood of a hill climber finding the global optimum.

Mathematical analysis holds out some tantalizing prospects of future progress, but for the moment we turn to a consideration of the results of experimental work on landscapes.

22.4 Statistical Measures

If mathematical analysis of a landscape is a difficult task, then it is natural to ask if there is something we can learn about the nature of a landscape, simply from the process of searching it. Several ideas have been suggested.

22.4.1 Autocorrelation

One of the earliest attempts to obtain some statistical measure of a landscape was by Weinberger, who showed that certain quantities obtained in the course of a random walk can be useful indicators ([Weinberger 1990](#)). If the fitness of the point visited at time t is denoted by f_t , we can estimate the *autocorrelation function* (usually abbreviated to *acf*) of the landscape during a random walk of length T as

$$r_j = \frac{\sum_{t=1}^{T-j} (f_t - \bar{f})(f_{t+j} - \bar{f})}{\sum_{t=1}^T (f_t - \bar{f})^2}.$$

Here \bar{f} is of course the mean fitness of the T points visited, and j is known as the *lag*. The concept of autocorrelation is of course an important one in time series analysis, but its interpretation in the context of landscapes is interesting.

For *smooth* landscapes, and at small lags (i.e. for points that are close together), the *acf* is likely to be close to 1 since neighbors are likely to have similar fitness values. However, as the lag increases the correlations will diminish. “Rugged” landscapes are informally those where close points can nevertheless have completely unrelated fitnesses, and so the *acf* will be close to zero at all lags. Landscapes for which the *acf* has significant negative values are conceptually possible, but they would have to be rather odd.

A related quantity is the *correlation length* of the landscape, usually denoted by τ . Classical time series analysis ([Box and Jenkins 1970](#)) can be used to show that the standard error of the estimate r_j is approximately $1/\sqrt{T}$, so that there is only approximately 5 % probability that $|r_j|$ could exceed $2/\sqrt{T}$ by chance. Values of r_j less than this value can be assumed to be zero. The correlation length τ is then the last j for which r_j is non-zero:

$$\tau = j : |r_{j+1}| < 2/\sqrt{T} \wedge \{|r_k| > 2/\sqrt{T} \quad \forall k \leq j\}.$$

The *acf* and the correlation length are useful indicative measures of the ruggedness of a landscape, but they are rather crude statistics, and it is difficult to attach a great deal of meaning to their values for particular instances.

22.4.2 Number of Optima

Although it is not the full story, the number of local optima is widely acknowledged as a very important factor in how easy or difficult it is to find a global optimum of a landscape, and is clearly much more directly relevant for a particular instance than the correlation measures. Recently, some attempts have been made ([Reeves 2001; Eremeev and Reeves 2002, 2003; Reeves and Eremeev 2004](#)) to obtain direct estimates of the number of optima using statistical principles.

It is assumed that a heuristic search method can be restarted many times using different initial solutions. Given the landscape framework we have discussed above, by randomly generating initial solutions, we will sample many basins of attraction. Of course, this will be evident by the number of *different* final solutions $\{x^o\}$ that are found. Suppose this number is k , and the number of restarts is $r(\geq k)$. Various statistical models may be used in order to estimate the number of optima v .

22.4.2.1 Waiting-Time Model

We can ask for the distribution of the waiting-time to find all optima. If r exceeds k substantially, this fact can be used to estimate the probability that all optima have been found. This would also imply, a fortiori, that the global optimum had been found, and thus provides us with an objective confidence level concerning the quality of the best solution obtained.

22.4.2.2 Counting Model

In the event—unfortunately, a common one—that k is not much smaller than r , it is unlikely that we have seen many of the optima. However, a counting model can be used to estimate the value of v , in a similar way to those used by ecologists to estimate the size of an unknown animal population. This can be quite illuminating in showing the differences between landscapes generated by different neighborhood operators.

22.4.2.3 Non-parametric Estimates

Fairly restrictive assumptions are needed in order to obtain tractable statistical models of landscapes. Where these estimates can be checked against actuality (by enumerating all points on a landscape), it appears that the effect of these assumptions is to produce negatively biased estimates—i.e. the estimate of v is consistently smaller than the true value. Removing the assumptions by creating more general models would probably be impossible, so some *non-parametric* approaches have been explored, and found to provide useful estimates of v , although the problem of negative bias remains, albeit at a lower level. All these models are summarized in [Reeves and Eremeev \(2004\)](#).

22.5 Empirical Studies

Besides explicit statistical models of landscape features, several empirical studies have been aimed at providing some idea of the “big picture”. Although multi-dimensional fitness landscapes have few similarities with “real” 3D landscapes,

certain empirical findings can be interpreted sensibly in terms of characteristics of real landscapes, which provides us with some insights into ways we can approach hard optimization problems.

One of the most interesting observed properties of fitness landscapes has been seen in many different studies: it is a feature of Kauffman's well-known NK landscapes (Kauffman 1993),⁴ and it also appears in many examples of combinatorial optimization problems, such as the traveling salesman problem (Boese et al. 1994; Lin 1965), graph partitioning (Merz and Freisleben 1998), and flowshop scheduling (Reeves 1999).

In the first place, such studies have repeatedly found that, on average, local optima are very much closer to the global optimum than are randomly chosen points, and closer to each other than random points would be. That is, the distribution of local optima is not *isotropic*; rather, they tend to be clustered in a *central massif* (or—if we are minimizing—a *big valley*). This can be demonstrated graphically by

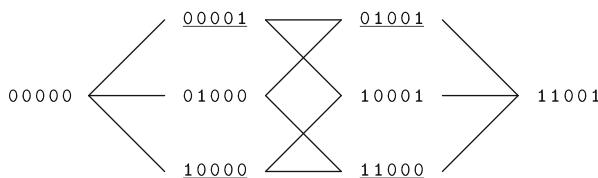


Fig. 22.1 The diagram shows the set of paths that could be traced between the parents 00000 and 11001. Only those intermediate vectors indicated by *underlines* can be generated by one-point crossover, but all can be generated by uniform crossover

plotting a scatter graph of fitness against distance to the global optimum. Secondly, if the basins of attraction of each local optimum are explored, size is quite highly correlated with quality: the better the local optimum, the larger is its basin of attraction. (If true, this also impinges on the estimation problem we discussed in the previous section: although there is a negative bias in the estimate of v , the *big valley* phenomenon implies that it is only the small basins and low-quality optima that we are missing.)

Of course, there is no guarantee that this property holds in any particular case, but it provides an explanation for the success of *perturbation* methods (Johnson 1990; Martin et al. 1992; Zweig 1995) which currently appear to be the best available for the traveling salesman problem. It is also tacitly assumed by such methods as simulated annealing and tabu search, which would lose a great deal of their potency if local optima were isotropically distributed.

⁴ In Kauffman's notation, N is the length of a binary string, and K is the maximum number of genes that are allowed to interact with any other; e.g. if $K = 1$, each gene can interact with just one other. There are several different ways in which the sets of interacting genes can be chosen, but essentially they turn out to make little difference.

22.5.1 Practical Applications

These studies have suggested a starting point for the development of new heuristic search algorithms, such as the *adaptive multi-start* algorithm of Boese et al. (1994). As a more recent example, we shall consider the *path tracing* algorithms introduced in Reeves and Yamada (1998) and Yamada and Reeves (1998), which can be motivated either as a use of the idea of a landscape, or in terms of extending the boundaries of evolutionary algorithms.

If we consider the case of crossover of vectors in $\{0, 1\}^\ell$, it is easily seen that any *child* produced from two *parents* will lie on a path that leads from one parent to another. Figure 22.1 demonstrates this fact.

In an earlier paper (Reeves 1994), such points were described as intermediate vectors. In other search spaces, the distance measure may be more complicated, but the principle is still relevant. Crossover is re-interpreted as finding a point lying *between* two parents in some landscape in which we hope the big valley conjecture is true. This *path-tracing crossover* was implemented for both the makespan and the flowsum versions of the flowshop sequencing problem; Fig. 22.2 shows in a two-dimensional diagram the idea behind it, while full details can be found in Reeves and Yamada (1998, 1999).

In this way, the concept of recombination can be fully integrated with traditional neighborhood search methods, and the results obtained for flowshop instances (see Reeves and Yamada 1998 and Yamada and Reeves 1998 for details) have been gratifyingly good. For the makespan problem, embedded path tracing helped the GA to achieve results of outstandingly high quality: several new best solutions were discovered for well known benchmarks. For the flowsum version, optimal solutions are not known, but the path-tracing GA consistently produced better solutions than other proposed techniques.

This idea has also recently been applied to multi-constrained knapsack problems (Levenhagen et al. 2001), where the need was confirmed for a *big valley* structure in order to benefit from this approach.

Very recently, a detailed examination of TSP instances (Hains et al. 2011) has shown that the big valley structure breaks down around high-quality local optima,

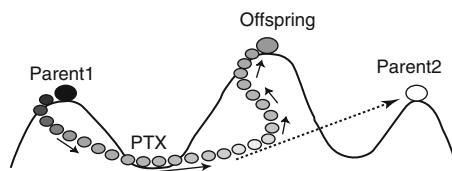


Fig. 22.2 Path tracing crossover combined with local search: a path is traced from one parent in the direction of the other. In the *middle* of the path, solutions may be found that are not in the basins of attractions of the parents. A local search can then exploit this new starting point by climbing to the *top* of a hill (or the *bottom* of a valley, if it is a minimization problem)—a new local optimum. The acronym PTX signifies *path-tracing crossover*

with multiple *funnel* structures appearing. Further investigation leads to new ideas about search operators for such instances, although whether this extends to other COPs is not yet known.

22.6 Promising Areas for Future Application

Finally, we should remark that several interesting future research questions are suggested. On the theoretical side, a deeper knowledge of the connections between algebra and graph theory may provide further useful analytical results. For example, it would be useful to have analytical results for all the common operators in permutation spaces analogous to those derived for the simpler case of binary strings. As mentioned in Sect. 22.3.4, real fitness landscapes for a number of combinatorial optimization problems have to cope with extensive plateaux. Measuring the extent and effects of such formations also needs further study, as does the characterization of basins of attraction. Some promising ideas based on the notion of a *barrier tree* have been put forward by Stadler and colleagues in Flamm et al. (2002), which are developed further in Hallam and Prugel-Bennet (2005).

Building on such notions, it would be helpful if we could provide a formal definition of what it means for a *big valley* structure to exist, and how it could be related to mathematical constructs associated with neighborhood structures. Does the big valley exist almost everywhere? If not, can we define classes of problems and neighborhood structures for which it does not occur? Further empirical analysis, such as that described by Levenhagen et al. (2001), Watson et al. (2002), and Hains et al. (2011), should be of considerable assistance in suggesting fruitful avenues to explore.

More generally, it is clear that crude correlation measures can only be a general guide to the nature of a landscape instance, and we need to find better ways of characterizing landscapes from empirical measurements. Some suggestions have been made in Reeves (2004) for further work in this direction.

In the area of implementation, it is important to see if we can further refine the path tracing methodology and its integration into heuristic search methods such as evolutionary algorithms. Also, the methodological developments pioneered in Reeves (2001), Eremeev and Reeves (2002, 2003), and Reeves and Eremeev (2004) for deducing properties of an instance of a landscape from the results of heuristic search offer the possibility of making principled probability statements about the quality of solutions obtained.

22.7 Tricks of the Trade

Mathematical analysis of landscapes is generally possible only for small problems, and then can only really be useful as an *a posteriori* validation (or questioning) of the decisions already made. However, empirical analysis is relatively easy and may provide some useful insights.

Correlation analysis can be a helpful indicator of the type of landscape with which we are dealing. Typically this proceeds by making a random walk on the landscape for several thousand steps and collecting data on fitness. The resultant *time series* can be analyzed with standard statistical tools. The drawback of this approach is that even when it is complete, knowing how smooth or rugged the landscape is from the perspective of a random walk does not help very much in deciding which heuristic search method to adopt. Further, much computation has been carried out yet the search for an optimum has not even started!

For those wishing to make use of empirical landscape analysis as part of a general research program, it should be realized that much of the necessary information is inherently generated in the course of applying a heuristic search method to a combinatorial optimization problem. Of course, if a single run is all that is used, nothing much can be gleaned, but if independent restarts or a Metropolis-type search are used, it becomes possible to collect statistics and make use of them.

The existence of a *big valley* is usually an encouraging feature, and requires little checking. Assuming the global optimum is unknown it will not be possible to do a complete analysis, but useful information can be gained by computing the distance of each local optimum from the best local optimum, and plotting this against their corresponding differences in fitness. A strong correlation is indicative of a *big valley*, and motivates the application of metaheuristics that perform intensive searches in the region of *good* local optima.

If every local optimum ever found is distinct, not much more can be done, but if it is noticed that specific local optima are being detected multiple times, it becomes possible to provide indications of solution quality, using statistical estimation tools based on the waiting-time or counting models mentioned above. For low values of the ratio k/r (see above), it may even be possible to provide a (probabilistic) guarantee that the global optimum has indeed been found.

22.8 Conclusion

This chapter has reviewed and discussed in some detail the basic mathematical theory and methods associated with the concept of a fitness landscape. While these methods can be very useful in enhancing our understanding of evolutionary algorithms, it has been emphasized that they cannot provide a complete explanation for the performance of a specific algorithm on their own—even in the case of very simple functions. Secondly, and more briefly, some empirically determined properties of many search landscapes have been described, and one approach whereby such properties can be exploited has been outlined.

As our understanding of the nature of fitness landscapes and how to exploit them develops, this promises to become an important area of research into the theory and application of heuristic search.

Sources of Additional Information

- For technical and theoretical analysis, there are many papers associated with Peter Stadler and his co-workers. The paper of [Reidys and Stadler \(2002\)](#) is perhaps the most readily accessible and recent treatment of theoretical properties of landscapes, although the seminal work is still [Stadler \(1995\)](#). Many of these papers can be found on the University of Leipzig website⁵ and also at the Santa Fe Institute.⁶
- Several papers give a general low-tech introduction to landscapes—for example [Reeves \(1999, 2000\)](#)—as does the chapter in [Reeves and Rowe \(2002\)](#). For ideas relating to elementary landscapes, see [Barnes et al. \(2003\)](#), [Dimova et al. \(2005\)](#), and [Whitley et al. \(2008\)](#).
- For correlation analysis, [Weinberger \(1990\)](#) is still a major source of information, supplemented by more recent work in papers by Stadler and co-workers (see the Vienna website); another useful reference is [Hordijk \(1996\)](#).
- For work relating to the *big valley* and its exploitation, there are several important papers: [Boese et al. \(1994\)](#), [Reeves and Yamada \(1998\)](#), [Merz and Freisleben \(1998\)](#), and [Reeves \(1999\)](#); a chapter by Reeves and Yamada in [Corne et al. \(1999\)](#) is also an accessible introduction.
- The statistical approach to estimation of landscape properties is described in a series of papers ([Eremeev and Reeves 2002, 2003](#); [Reeves and Eremeev 2004](#)), and its extension to the use of the Metropolis algorithm is considered in [Reeves and Aupetit-Bélaidouni \(2004\)](#).

References

- Barnes JW, Dimova B, Dokov SP, Solomon A (2003) The theory of elementary landscapes. *Appl Math Lett* 16:37–343
- Biggs NL (1993) Algebraic graph theory. Cambridge University Press, Cambridge
- Boese KD, Kahng AB, Muddu S (1994) A new adaptive multi-start technique for combinatorial global optimizations. *Oper Res Lett* 16:101–113
- Box GEP, Jenkins GM (1970) Time series analysis, forecasting and control. Holden Day, San Francisco
- Corne DA, Dorigo M, Glover F (eds) (1999) New methods in optimization. McGraw-Hill, London
- Dawkins R (1996) Climbing mount improbable. Viking, London
- Dimova B, Barnes JW, Popova E (2005) Arbitrary elementary landscapes and AR(1) processes. *Appl Math Lett* 18:287–292
- Dobzhansky T (1951) Genetics and the origin of species. Columbia University Press, New York

⁵ www.bioinf.uni-leipzig.de/~studla/Publications/index.html

⁶ www.santafe.edu/research/publications/

- Eigen M (1993) Viral quasispecies. *Sci Am* 269:32–39
- Eigen M, McCaskill J, Schuster P (1989) The molecular quasi-species. *Adv Chem Phys* 75:149–263
- Eldredge N, Cracraft J (1980) Phylogenetic patterns and the evolutionary process. Columbia University Press, New York
- Eremeev AV, Reeves CR (2002) Non-parametric estimation of properties of combinatorial landscapes. In: Gottlieb J, Raidl G (eds) Applications of evolutionary computing. LNCS 2279. Springer, Berlin, pp 31–40
- Eremeev AV, Reeves CR (2003) On confidence intervals for the number of local optima. In: Raidl G et al (eds) Applications of evolutionary computing. LNCS 2611. Springer, Berlin, pp 224–235
- Flamm C, Hofacker IL, Stadler PF, Wolfinger MT (2002) Barrier trees of degenerate landscapes. *Z Phys Chem* 216:155–173
- Frank J, Cheeseman P, Stutz J (1997) When gravity fails: local search topology. *J Artif Intell Res* 7:249–281
- Futuyma DJ (1998) Evolutionary biology. Sinauer Associates, Sunderland
- Godsil CD (1993) Algebraic combinatorics. Chapman and Hall, London
- Grover LK (1992) Local search and the local structure of *NP*-complete problems. *Oper Res Lett* 12:235–243
- Hains DR, Whitley LD, Howe A (2011) Revisiting the big valley search space structure in the TSP. *J Oper Res Soc* 62:305–312
- Haldane JBS (1931) A mathematical theory of natural selection, Part VI: metastable populations. *Proc Camb Phil Soc* 27:137–142
- Hallam J, Prugel-Bennett A (2005) Large barrier trees for studying search. *IEEE Trans Evol Comput* 9:385–397
- Hordijk W (1996) A measure of landscapes. *Evol Comput* 4:335–360
- Johnson DS (1990) Local optimization and the traveling salesman problem. In: Goos G, Hartmanis J (eds) Automata, languages and programming. LNCS 443. Springer, Berlin, pp 446–461
- Jones TC (1995) Evolutionary algorithms, fitness landscapes and search. Doctoral dissertation, University of New Mexico, Albuquerque
- Kaplan J (2008) The end of the adaptive landscape metaphor? *Biol Phil* 23:625–638
- Kauffman S (1993) The origins of order: self-organization and selection in evolution. Oxford University Press, New York
- Levenhagen J, Bortfeldt A, Gehring H (2001) Path tracing in genetic algorithms applied to the multiconstrained knapsack problem. In: Boers EJW et al (eds) Applications of evolutionary computing. Springer, Berlin, pp 40–49
- Lin S (1965) Computer solutions of the traveling salesman problem. *Bell Syst Tech J* 44:2245–2269
- Martin O, Otto SW, Felten EW (1992) Large step Markov chains for the TSP incorporating local search heuristics. *Oper Res Lett* 11:219–224
- Merz P, Freisleben B (1998) Memetic algorithms and the fitness landscape of the graph bi-partitioning problem. In: Eiben AE et al (eds) Proceedings of the PPSN V, Amsterdam. Springer, Berlin, pp 765–774

- Reeves CR (1994) Genetic algorithms and neighbourhood search. In: Fogarty TC (ed) Evolutionary computing: AISB workshop selected papers. Springer, Berlin, pp 115–130
- Reeves CR (1999) Landscapes, operators and heuristic search. *Ann Oper Res* 86:473–490
- Reeves CR (2000) Fitness landscapes and evolutionary algorithms. In: Fonlupt C et al (eds) 4th European conference on artificial evolution. LNCS 1829. Springer, Berlin, pp 3–20
- Reeves CR (2001) Direct statistical estimation of GA landscape features. In: Martin WN, Spears WM (eds) Foundations of genetic algorithms 6. Morgan Kaufmann, San Francisco, pp 91–107
- Reeves CR (2004) Partitioning landscapes. Available online at <http://www.dagstuhl.de/04081/Talks/>
- Reeves CR, Aupetit-Bélaidouni MM (2004) Estimating the number of solutions for SAT problems. In: Yao X et al (eds) Proceedings of the PPSN VIII, Birmingham. LNCS 3242. Springer, Berlin, pp 101–110
- Reeves CR, Eremeev AV (2004) Statistical analysis of local search landscapes. *J Oper Res Soc* 55:687–693
- Reeves CR, Rowe JE (2002) Genetic algorithms—principles and perspectives. Kluwer, Norwell
- Reeves CR, Yamada T (1998) Genetic algorithms, path relinking and the flowshop sequencing problem. *Evol Comput* 6:45–60
- Reeves CR, Yamada T (1999) Goal-oriented path tracing methods. In: Corne DA, Dorigo M, Glover F (eds) New methods in optimization. McGraw-Hill, London
- Reidys CM, Stadler PF (2002) Combinatorial landscapes. *SIAM Rev* 44:3–54
- Ridley M (1993) Evolution. Blackwell, Oxford
- Simpson GG (1953) The major features of evolution. Columbia University Press, New York
- Skipper R (2004) The heuristic role of Sewell Wright's 1932 adaptive landscape diagram. *Philos Sci* 71:1176–1188
- Stadler PF (1995) Towards a theory of landscapes. In: Lopéz-Peña R et al (eds) Complex systems and binary networks. Springer, Berlin, pp 77–163
- Stadler PF, Wagner GP (1998) Algebraic theory of recombination spaces. *Evol Comput* 5:241–275
- Waterman MS (1995) Introduction to computational biology. Chapman and Hall, London
- Watson J-P, Barbulescu L, Whitley LD, Howe AE (2002) Contrasting structured and random permutation flow-shop scheduling problems: search-space topology and algorithm performance. *INFORMS J Comput* 14:98–123
- Weinberger ED (1990) Correlated and uncorrelated landscapes and how to tell the difference. *Biol Cybern* 63:325–336
- Whitley LD, Sutton AM, Howe AE (2008) Understanding elementary landscapes. In: Proceedings of the GECCO 2008, Atlanta. ACM, New York, pp 585–592

- Wright S (1932) The roles of mutation, inbreeding, crossbreeding and selection in evolution. In: Jones D (ed) Proceedings of the 6th international congress on genetics, Ithaca, vol 1, pp 356–366
- Wright S (1967) Surfaces of selective value. Proc Nat Acad Sci 102:81–84
- Wright S (1988) Surfaces of selective value revisited. Am Nat 131:115–123
- Yamada T, Reeves CR (1998) Solving the C_{sum} permutation flowshop scheduling problem by genetic local search. In: Proceedings of the CEC 1998. IEEE, Piscataway, pp 230–234
- Zweig G (1995) An effective tour construction and improvement procedure for the traveling salesman problem. Oper Res 43:1049–1057

Index

- acceleration coefficients, 229
acceptance probability, 269, 272
ACO, *see* ant algorithms
adaptive
 heuristics, 221, 224
 immune system, 186–208
ADF, *see* automatically defined functions
adjacency matrix, 331, 690
affinity measure, 194–196
agents, 393
Alice, 387
alldifferent constraint, 373, 383
alleles, 93, 97, 99
amplitude spectrum, 692
ancillary tests, 329
ant algorithms, 214–227
 candidate lists, 226
 changing α , β values, 226
 lookahead, 226
 lower bounds, 226
 moving direction, 226
 repelling pheromone, 226
 stagnation recovery, 226
antibody, 188–201
antigen, 188–194, 196–198, 204, 206
aperiodic, 272
approximate evaluation, 107
approximate rule, 563
approximation algorithms, 639–679
approximation scheme, 327, 641
approximation strategies, 643–668
APX, 641, 670–671
arc consistency, 371, 375, 377
architecture, 370, 393, 492, 495, 498, 509
architecture-altering operations, 147, 150, 152,
 157
artificial immune systems, 186–211
auto-immune attacks, 191
B-cells, 188–190
cellular, 188
cellular immunity, 190
encoding, 193
immune network theory, 191
immunological memory, 191
initialization, 193
innate, 187
self cells, 191
T-cells, 188–190
artificial intelligence, 3
aspiration criteria, 250, 257
association rules, 571, 600
asymmetric, 591
asymmetric distance measure, 688
asymmetric matrix, 691
attributes, 481, 485, 510, 557, 560–567, 571,
 580, 588, 597, 599, 600, 602
autocorrelation, 696
automatically defined functions, 156
auxiliary objectives, 254
axes of a search space, 681
B-cells, 196, 202
backpropagation algorithm, 490
backtrack search, 25, 28–31, 374–375, 377,
 387, 398, 672
 complete backtrack search, 672
 complete randomized, 672
 intelligent, 375
backtrack-free, 378
backtracking, 59
backward recursion, 38, 60
base formulation, 77
basin of attraction, 684, 689, 698, 700
batch mode of training, 490

- Bayesian learning, 478, 484–486, 511
 Bellman's principle, 36
 best improvement, 250, 317, 333, 684
 best-bound search, 87
 best-first strategy, 35, 59, 459
 best-of-generation, 154
 big-M model, 75, 79
 bin packing, 639, 650, 669
 binary variables, 72–76, 78–85
 bioinformatics, 370, 384
 bird flocking, 213, 227
 bloat, 160, 169, 628
 blossom algorithm, 55
 Boltzmann
 - constant, 268
 - distribution, 269, 273
 Boolean satisfiability, 453, 615
 boosting, 508
 bottom-up learning, 483
 bounding argument, 71
 branch and backtrack, 25
 branch and bound, 19, 24–35, 41, 51, 59, 68, 71, 72, 76, 80, 83, 85, 379
 - parameters, 86
 branch and price, 85, 86
 branching strategy, 25, 26, 28, 59
 breadth-first search, 25, 87
 Bron and Kerbosch algorithm, 29, 34
 Brown's algorithm, 29
 building blocks, 102–104, 107
 - exchange, 104
 - identification, 104
 C4.5, 482, 486
 candidate list, 226, 251, 291
 candidate solutions, 144
 cantilever plate design, 430
 Cartesian inner product of fuzzy sets, 526
 certain rules, 558
 chess, 376, 479
 chromosome, 93–101, 109, 496
 Cigol, 484
 classical set, 521
 classical techniques, 19–65
 classification, 199, 208, 478–482, 508, 519, 545, 559–574, 600, 602
 - tree, 481–482
 clique, 29, 31–35, 81
 clonal selection, 190–192, 196
 closure, 461
 CLP, 388
 co-domain, 454–457, 459, 463
 co-evolutionary learning, 492
 collapsed matrix, 692
 combinatorial optimization, 287, 340, 348, 683–685
 competent GAs, 95, 101, 104, 105
 competition complexity, 103
 complement A' , 525
 complementarity property, 565, 576
 complementary slackness, 46, 330
 complex systems theory, 519
 complexity theory, 451–476
 computer programs, 143
 computer security problems, 204
 condition attributes, 567
 condition profile, 572
 connection weights, 495
 constraining structures, 157
 constraint, 5, 67–89
 - databases, 392
 - explicit, 83
 - generation, 86
 - hard, 5
 - implicit, 83
 - logic programming, 387
 - matrix, 42
 - programming, 368–401
 - propagation, 380–385
 - satisfaction problems, 369–372, 393, 672
 - soft, 5
 constraints and graphics, 391
 construction method, 215
 constructive heuristic, 8, 109
 constructive method, 24
 continual improvement, 94
 continuous
 - diversification, 253
 - function, 228–229, 314, 481
 - optimization, 228–229, 255, 683
 control parameter, 146, 269–273, 276–280, 454
 convex
 - function, 326
 - hull, 385
 cooling schedule, 279
 cooperative swarms, 235
 - CPSO optimizer, 235
 - split swarm, 235
 crossover, 56, 57, 96–99, 108, 131, 143, 147, 150, 493, 495, 685, 687, 693–695
 - k -point, 97
 - cycle, 99
 - one-point, 96, 99, 693, 698
 - order-based, 98
 - partially matched, 99, 131
 - two-point, 96, 693
 - uniform order-based, 97

- crowded tournament selection, 423
crowding distance metric, 423
curse of dimensionality, 511
cuts, 34, 78–80, 330
cutting and packing, 4, 611
CX, *see* crossover
cyclic network, 46
- danger theory, 202–204
Darwinian
 evolution, 171
 selection, 150, 155
data mining, 192–194, 198, 207, 479, 571, 600
decision
 attributes, 567, 573–574, 602
 classes, 558, 563, 567, 602
 making, 437
 problem, 452, 520, 557, 672
 profile, 572
 rules, 582–585
 tree, 25–33, 452, 453, 478, 482, 486, 510, 511, 571
variable, 67, 93, 381, 683
variable space, 406
decomposition, 102, 163, 327, 544, 694
 principled efficiency-enhancement
 technique, 95
defuzzification unit, 533–535
 center of sums technique, 534
 centroid method, 540
 centroid technique, 533
 mean of maximum technique, 533
degree of certainty, 566
delete-all, 101
denial-of-service attacks, 192
density estimation, 479
depth-first, 25, 59, 87, 153, 374
design innovation, 432
deterministic, 6
 polynomial, 452
differential calculus, 683
direct encoding of neural networks, 496
direction of branching, 87
discrete
 function, 455
 binary PSO algorithm, 231
 function, 490
distance-to-target diagrams, 315
distinct eigenvalues, 692
distributed processing, 393
distribution of neighborhoods, 329
diversification, 252, 255–257, 292, 333
 generation method, 119, 122–125
dominance, 570, 585, 598, 600
cones, 575–581
relation, 413, 559, 573, 589, 592–594
 without degrees of preference, 594
dominance-based rough set approach, 559, 589
domination, 412
 definition, 412
doubtful region, 565, 578, 593, 602
DRSA, 559, 589
DSATUR, 31
dual variables, 46
duality, 330
 dual value, 84
 dual problem, 61
 dual simplex, 87
dynamic
 programming, 487
 cooling schedules, 280
 heuristics, 382
 programming, 19, 36–42, 51–53, 57, 58, 60, 488, 653
ECLiPSe, 387
elementary sets, 558
elitism strategy, 220
elitist solutions, 220–221, 255
EMO
 a posteriori approach, 437
 a priori approach, 437
 cantilever plate design, 430
 GPU computing, 439
 interactive approach, 437
 many objectives, 438
 non-evolutionary approaches, 438
 parallel implementation, 439
 post-optimality studies, 435
 redundant objectives, 440
 research challenges, 437
 spacecraft trajectory design, 427
 theoretical studies, 440
ensemble learning, 500
entropy, 482
enumeration, 24, 56, 60, 317, 456
Euclidean distance, 128
evaluation
 difficulty of a problem, 316
 function, 5
 performance of previous algorithms, 316
 performance of previous heuristics, 316
 relaxation, 107
 search algorithms, 470
evolution, 93, 109, 170, 268
 connection weights, 495–497
 of architectures, 496

- of learning rules, 498
- strategies, 100, 492
- evolutionary
 - fuzzy logic systems, 492
 - learning, 478
 - neural networks, 492, 507
 - programming, 492
- exact algorithm, 316
- exact methods, 10
- exchangeable
 - attribute, 562
 - criteria, 580, 588
- exhaustive search, 7
- expanded formulation, 77
- facility location, 68–71, 75, 77, 80, 87
- fail first, 375
- feasible solution, 4
- financial decision support system, 370
- finite element method, 430
- finite-state machines, 492
- firewall, 192
- first improvement strategy, 684
- fish schooling, 227
- fitness, 5
 - function, 107
 - inheritance, 107
 - landscapes, 680–705
 - an example, 683
 - empirical studies, 697
 - mathematical characterization, 686
 - practical applications, 699
 - measure, 145, 147, 148, 152
 - of a neural network, 495
 - proportionate selection, 95, 154
- fixed charge, 74, 247
- flow-augmenting chains, 43–49, 54
- Floyd’s shortest path algorithm, 52
- Ford–Fulkerson algorithm, 43–45, 50
- forward recursion, 37
- fully polynomial time approximation scheme, 641, 652, 668–670
- fuzzy
 - adaptive control schemes, 541–543
 - CSPs, 379
 - implication, 529
 - inference system, 531–535
 - defuzzification unit, 531–534
 - logic, 511, 520–521, 524
 - measures, 580
 - relation, 527
 - set composition, 528
 - set operations, 525
 - sets, 519–547
- similarity measures, 530–531
- systems
 - modeling, 543
 - stability, 544
- fuzzy reasoning, 518–556
- GA, *see* genetic algorithm
- gene
 - deletion, 143
 - duplication, 143
- generalization test, 480, 492
- generation probability, 272–274
- genes, 93, 98–99, 698
- genetic algorithm, 93–117, 131, 143, 193, 195, 198, 199, 201, 207, 231, 246, 376, 493, 684
 - relaxation, 255
- genetic programming, 142–185
 - developmental, 159
 - probabilistic, 159
 - theory, 167
- genotype, 199, 496
- global constraints, 383
- global optimum, 6, 698
- globally Pareto-optimal set, 416
- Gomory–Chvátal procedure, 78
- GP, *see* genetic programming
- gradient descent algorithm, 490
- granulation, 558
- granules of knowledge, 560, 593
- graph
 - algorithms, 53
 - bipartite, 55, 646
 - coloring, 29–33, 81, 324, 369, 639
 - complete, 32
 - eigensystem, 692
 - partitioning, 266, 698
 - problem, 42
 - representation, 690
 - theory, 28, 58, 255, 331, 454, 519, 700
- GRASP, 122, 129, 286–303
- Gray code, 466, 685
- greedy, 125
 - algorithms, 288
 - approximation algorithms, 644
 - construction, 123
 - heuristic, 319, 324
 - knapsack, 648
 - MAX-CUT, 647
 - MAX-SAT, 646
 - solution, 295
 - vertex cover, 645
- grow initialization method, 147

- H-means, 320
HAL, 387
Hamming
 distance, 128, 194, 330, 687
 landscape, 687, 693
Heaviside function, 489
Hebbian learning rule, 489
hedge, *see* transformation operator
heuristics, 8–9, 613
 generation, 616, 619, 622
hill climbing, 9
homogeneous, 42, 51, 272, 279
Hopfield networks, 491
human-competitive, 170
hyper-heuristics, 11, 610–638
 hybrid, 614
hypervolume, 439
- ideal point, 410
idiotypic networks, 200–201
inclusion property, 565, 576
incomplete search technique, 397
independent variables, 145
indicator function, 687
indirect encoding of neural networks, 497
indiscernibility relation, 558, 559, 564, 569, 573
indispensable
 attribute, 562, 563, 566
 criteria, 580, 588
inductive learning, 480–484, 568
inductive logic programming, 478, 482–484
inertia weight, 229
infeasible solutions, 4
inference methods, 371–372
inference rules, 520, 529
infix notation, 145
information
 gain, 481
 theory, 477, 481
 transfer, 219
inhomogeneous algorithm, 274
initial random population, 150, 151, 155
initial solution, 8
initialization, 94, 147, 196, 234, 329, 495
innovation, 94, 102
innovation, 431
 automated procedure, 431
integer programming, 66–92
integer quantities, 73
intensification, 216, 225, 252, 256–257, 292, 333
interchangeability, 372
- interior point, 87
intermediate vectors, 698
intermediate-term memory, 252
interpolation, 479
intersection, 381, 521, 525, 558, 560, 562, 567, 576, 578, 594
intractable, 105, 378, 639
invariant, 104, 331
inverse consistency, 372
inverse problem, 530
irreducible, 272
iterative improvement, 266, 269
- job shop scheduling, 207, 222, 244, 247, 253, 266, 651
- K-means, 320
kilter
 diagram, 46
 line, 46, 54
knapsack
 binary knapsack problem, 55
 bounded knapsack problem, 55
 maximum fraction, 649
 maximum integer, 649, 655
 minimum integer, 655
 problem, 29, 40, 60, 83, 128, 642, 648, 650
 unbounded, 37, 55
- Lagrangian relaxation, 34
Laplacian matrix, 691
learning
 algorithms, 471, 480–508
 Bayesian, 484–486
 bottom-up, 375, 483
 chess play learning, 479
 classifier systems, 207, 493
 decision-tree, 483
 element, 477
 methods, 502–507
 reinforcement, 486–488
 robot learning, 479
 sample, 570
 sequence of actions, 479
 top-down, 483
level of confidence, 568
limited discrepancy search, 375
linear propagation, 385
linear relaxation, 69–71, 76
linearly separable, 489
Lisp S-expressions, 144, 169
local optimum, 6

- local search, 9, 109, 120, 125, 127, 290, 341
 locally Pareto-optimal set, 416
 logic programming, 387
 logical constraints, 75
 long-term memory, 253
 look-ahead, 375
 lower approximation, 558, 561, 593
 lower bound, 26–31, 34, 41, 46–49, 53–55
- machine learning, 477–517
 makespan, 244, 699
 Markov chains, 271–274, 279
 matching function, 193, 197, 200
 matching problem, 55
 matheuristics, 11
 mating pool, 95
 MAX-Ak-SAT, 668
 max-closure, 378
 max-CSP problem, 379
 MAX-CUT problem, 128, 343, 647
 MAX-SAT, 646, 668, 670
 maximum
 diversity problem, 129
 flow problem, 43–45
 weighted independent set, 85
 McCulloch–Pitts neurons, 488
 MCDM, 437
 mean square error, 490, 496
 membership function, 521–528, 566
 Π , 523
 Gaussian, 524
 monotonically decreasing, 522
 monotonically increasing, 522
 trapezoid, 522
 triangular, 522
 memetic algorithm, 106, 109
 metaheuristics, 10–11
 Metropolis algorithm, 268, 702
 min-conflicts heuristic, 376
 minimal domain size, 374
 minimum
 cost flow problem, 45–50
 job scheduling, 651
 sum-of-squares clustering, 329
 vertex cover, 642, 645, 658
 mixed integer programming, 255
 modern heuristics, *see* metaheuristics
 modifier, *see* transformation operator
 MOEAs, 421
 Monte Carlo, 268, 332, 487
 MOOP, *see* multi-objective optimization problem
 multi-attribute, 557, 602
 multi-objective optimization, 403–443
 ϵ -constraint approach, 420
 classical approach, 409
 differences from single-objective optimization, 405
 ideal approach, 407
 preference-based approach, 409
 principles, 406
 weighted-sum approach, 419
 multi-objectivization, 439
 multicriteria, 557–559, 571, 585–590, 599, 600
 multigraded dominance, 592–594
 multilayer feedforward neural networks, 490
 multimodal landscapes, 689
 multistage programming, 36
 mutation, 57, 94, 99, 107, 143, 147, 150, 191, 198, 493, 495, 497
 probability, 100, 109
- nadir point, 411
 negative dominance cone, 574
 negative selection, 191, 196–197, 205
 neighbor, 266, 683
 neighborhood, 266
 graph, 267
 search, 313
 very large scale, 339–364
 structure, 247, 686–688
- network
 flow, 50, 54, 56, 391
 flow programming, 19, 42–51, 58, 60
 management, 370
 simplex algorithm, 50
 neural networks, 488–492, 494–497
 no free lunch, 450–476
 node potentials, 46
 non-dominated
 front, 417
 set, 414
 solutions, procedure to identify, 416
 sorting, 418
 non-parametric estimates, 697
 NP-complete, 51, 453
 NP-hard, 245, 265, 378, 452, 639, 641, 673
 NSGA-II, 422
- objective
 function, 5
 objective normalization, 412
 objective space, 406
 off-policy, 487–488
 on-policy, 487–488

- online, 488, 650
operational research, 4
operations research, 4
OPL, 88, 388
optimal solutions, 6, 613
optimization, *see* combinatorial optimization,
 50, 83, 86, 93, 199, 207, 226, 234, 237,
 390, 492, 646
 algorithm, 105, 266
ordinal selection, 96
out-of-kilter, 46–50, 53, 56
outranking, 559, 574, 590, 593
- P, 451
P-boundary, 564
P-dominated set, 574
P-lower approximation, 564, 576
P-rough set, 564, 565
P-upper approximation, 564, 576
p-median, 319, 321, 333
parallel, 188, 452, 488, 651
 algorithm, 393
 search strategy, 170, 255
parallelization, 61, 105
parameter
 calibration, 258
 optimization, 453–455
parental solutions, 94
Pareto, 137, 571
 optimal solutions, 403, 405, 410
partial order, 414
partial solutions, 24–31, 34, 57, 389
particle swarm optimization, 137, 214,
 227–236
 adaptive PSO, 231
 advanced features, 231
 controlling diversity, 233
 convergence enforcement, 232
 maximum velocity, 232
 neighborhood best velocity update, 231
 PSO algorithm, 228–229
 queen particle, 232
 simplified PSO, 232
 stagnation recovery, 234
partitioning, 24, 29, 35, 319, 324, 650, 669
path
 consistency, 372
 relinking, 121, 133, 292–298
 tracing, 699
pathogen, 188
pattern classification, 478
PCPs, *see* probabilistically checkable proofs
Pearson correlation coefficient, 195
penalty function, 5
perceptron learning, 488–490
perfect graph, 33
permutation, 31, 81, 131, 222, 244, 267, 454,
 688, 694
closure, 460–463
code, 97
matrix, 690
problems, 126, 128, 131, 216, 219, 226
personnel scheduling, 369
PESA, 425
phenotype, 199
pheromone, 214
 best-worst, 225
 matrix, 217–220
 update, 216, 225–227
 values, 217–219
plant location problem, 243–245, 247, 249,
 252–254
plug flow tubular reactor case study, 538
PMX, 99
polynomial time, 452
 algorithm, 644, 668–670
 approximation scheme, 641, 650
 verifier, 671
population, 10, 94–96, 143–161
 matrix, 219
 matrix update, 220
 population-based ACO, 219
 size, 152
positive dominance cone, 574, 575
predictive system, 481
prefix notation, 144
primal, 330
primal simplex, 87
primal-dual method, 659
principle of optimality, *see* Bellman's principle
prior knowledge, 480, 570–574, 600
probabilistic safety factor, 103
probabilistically checkable proofs, 670–671
probabilistically selected, 146, 151
probability distribution, 218, 272, 478, 666
probability space, 478
problem-specific repair mechanism, 97
production planning problem, 38–42
production scheduling, 370, 391, 546
propagation, 372, 377, 380–385, 391, 672
proportional-differential-like fuzzy controller,
 537
proportional-integral-like fuzzy controller, 536
protected division, 152, 155
pruning, 26, 28, 34, 375
PSO, *see* particle swarm optimization

- PTAS, *see* polynomial time approximation scheme
- Q*-learning, 487
- quality measure, 565
- quality of approximation, 561, 563, 565, 567, 579, 586, 593, 602
- queens problem, 376
- random
- 3SAT, 673
 - binary template, 98
 - bouncing, 234
 - constants, 145
 - crossover, 98
 - cut, 667
 - enumeration, 456, 473
 - initial weights, 497
 - jump, 333
 - MAX-SAT solution, 668
 - number, 95, 233, 269, 332, 456
 - restart, 375
 - sampling, 107, 251, 456
 - search, 154, 458, 473
 - selection, 98, 150, 320, 323
 - solution, 332
 - value, 229
 - variable, 641, 667
 - walk, 94, 246, 696, 701
- randomization, 666
- ranking selection, 94
- real-time decision problem, 255
- recombination, 94–96, 107, 492, 699
- landscapes, 694–695
 - operators, 96, 104, 106
 - sexual, 143, 150, 170
- recursive relationship, 37, 40, 51, 60
- reduced VNS, 321
- reduct, 567, 580, 586, 599
- reducts, 562
- redundant criteria, 580
- reflexive, 569–570, 574, 592, 594
- regression, 151, 479
- tree, 481
- reinforcement learning, 478, 486–488, 492, 495, 510–511
- relaxation, 34, 76, 107, 648, 672
- repair, 50, 373, 375–376, 386
- replacement, 94, 96, 101, 109, 456
- reproduction, 95, 143, 147, 150–152, 155, 496
- resource allocation, 370, 454
- restart diversification, 253
- robustness, 258
- rough approximation, 559, 564–568, 570, 573, 577, 585, 589, 594
- rough sets, 137, 138, 557–609
- certain, 567
 - certain knowledge, 559, 576, 583
 - certain rules, 563, 570
 - dominance-based approach, 559, 589
 - formal description, 563
 - fundamentals, 559–570
 - illustrative example, 597
 - indiscernibility-based approach, 563
 - indiscernibility-based approach, 558, 559, 589, 602
 - uncertain knowledge, 559
- roulette wheel, 94–95
- routing, 36, 42, 60, 329, 391, 672
- rule base design
- heuristic, 534
 - systematic, 534
- rules
- induction of, 567
 - running metric, 436
- Sarsa learning algorithm, 487
- scatter search, 119–139
- diversification generation, 119, 127, 135
 - improvement, 119, 127, 136
 - reference set update, 119, 120, 122, 127–129, 136
 - solution combination, 119–122, 131–134, 136
 - subset generation, 119–121, 129–131
- schema theorem, 96
- search space, 7, 247–248, 253
- selection, 95–96, 147
- selection-intensity models, 102
- self-adaptive systems, 492
- separation, 83, 265, 369, 386, 389, 508
- sequencing problems, 61, 76
- sequential
- algorithms, 650–652
 - job scheduling, 651
 - mode of training, 490
- shaking, 323, 329, 332
- short-term memory, 249
- shortest path, 26–29, 36, 42, 51, 57, 214–217, 645, 687
- SICStus, 387
- similarity, 192, 520, 568–570
- classes, 569
 - measure, 193–196, 545
- simplex, 42, 43, 46
- simulated annealing, 264–285, 289

- single machine total weighted tardiness problem, 221
skewed VNS, 326
social insect colonies, 213
spacecraft trajectory design, 427
SPEA2, 425
staff planning, 370
states, 36
steady state, 94, 101, 539
steepest descent, 317, 327
stochastic, 36, 315, 318, 373, 490
 element, 375
 noise, 104
 programming, 255
 search algorithm, 456
 universal selection, 94, 95
 variable, 271
stopping criterion, 216, 218
strategic oscillation, 253
subcomponent complexity, 103
subjective function, 93
sum-of-squares clustering, 319
superfluous attribute, 562, 566
supervised learning, 478, 492, 510
supply chain management, 370
surrogate objectives, 254
survival of the fittest, 94
swapping probability, 97
swarm intelligence, 212–242
symbolic regression, 151
synapse, 488, 489
syntax, *see* tree, syntax, 567, 570, 585, 589, 594, 599
- tabu
 list, 57, 361
 length, 249
 search, 133, 135, 243–263, 327
 multiple tabu list, 249
 probabilistic, 251
 recency memory, 252
 tenure, 249
takeover time models, 102
task scheduling, 370
Tchebycheff catastrophe, 314
temperature, 189, 268, 454, 524, 527
temporal difference learning, 487
terminal
 node, 25, 32, 481
 set, 145, 152
termination criterion, 146, 151, 152, 155, 251
test function, 230
thrashing behavior, 374
threshold function, 489
threshold methods, 246
time continuation, 107
time-independent, 272
timetabling, 29, 56, 106, 390, 619
top-down learning, 483
tournament selection, 94, 96, 154
tractability, 378–379, 452, 673
trade-off solutions, 404
transfer functions, 490
transformation operator, 520, 526, 545
transitive, 564, 570, 591, 594
transportation, 370, 382, 384, 390
 assignment, 53
 cost, 244
 problem, 53, 245, 247, 254
traveling salesman problem, 7, 29, 57, 93,
 97, 99, 207, 215–221, 266, 328, 340,
 345–347, 457, 698
 minimum, 639, 670
tree, 25–27, 481
 rooted point-labeled program, 147
 syntax, 144, 149, 150
truncation selection, 96
TSP, *see* traveling salesman problem
Turing, 143, 170, 488
 machine, 452–453
two-dimensional cutting problems, 40–41
- uncertain knowledge, 511, 559
uniform probability, 150
unimodal landscapes, 689
unimodular, 42
union \cup , 525
unsupervised learning, 478, 492, 510
update, 218
upper approximation, 558, 593, 602
upper bound, 26–30, 32–35, 43–49, 53–55, 74,
 229, 233, 375, 530
utility service optimization, 370
utopian point, 411
- variable
 generation, 84–86
 neighborhood search, 290, 313–337
 reduced, 313, 320–323, 332
 skewed, 313, 325–327
 variable neighborhood descent, 126, 313
 VNS within exact algorithm, 330
 precision rough set approach, 578
vehicle routing, 255, 348
Visual CHIP, 388
VNDS, 313, 315, 327

- VNS, *see* variable neighborhood search, 322, 327
Vogel's approximation method, 54
waiting-time model, 697
weak preference relation, 574
weighted maximum satisfiability, 326
weighted-sum approach, 419
 Y -reduct, 567
zero-argument functions, 145
Zykov's algorithm, 31