Jason Brownlee

Clever Algorithms

Modern Artificial Intelligence Recipes

Clever Algorithms: Modern Artificial Intelligence Recipes

License Summary

Preface

About the book

The need for this project was born of frustration while working towards my Ph.D. I was investigating optimization algorithms and was implementing a large number of them for a software platform called the Optimization Algorithm Toolkit (OAT)[1]. Each algorithm required considerable effort to locate the relevant source material (from books, papers, articles, and existing implementations), decipher and interpret the technique, then to finally attempt to piece together a functional implementation.

Taking a broader perspective, I realized that the communication of algorithmic techniques in the field of Artificial Intelligence was clearly a difficult and outstanding open problem. Generally, algorithm descriptions are:

- *Incomplete*: many techniques are ambiguously described, partially described, or not described at all.
- *Inconsistent*: a given technique may be described using a variety of formal and semi-formal methods that vary across different techniques, limiting the transferability of skills an audience requires (such as mathematics, pseudo code, program code, and narratives). An inconsistent representation for techniques mean that the skills used to understand and internalize one technique may not be transferable to realizing different techniques or even extensions of the same technique.
- *Distributed*: the description of data structures, operations, and parameterization of a given technique may span a collection of papers, articles, books, and source code published over a number of years, the access to which may be restricted and difficult to obtain.

For the practitioner, a badly described algorithm may be simply frustrating, where the gaps in available information are filled with intuition and 'best guess'. At the other end of the spectrum, a badly described algorithm may an example of bad science and the failure of the scientific method, where the inability to understand and implement a technique may prevent the replication of results, the application, or the investigation and extension of a technique.

[1] OAT located at http://optalgtoolkit.sourceforge.net

The software I produced provided a first step solution to this problem: a set of working algorithms implemented in a (somewhat) consistent way and downloaded from a single location (features likely provided of any library of artificial intelligence techniques). The next logical step needed to address this problem is to develop a methodology that anybody can follow. The strategy to address the open problem of poor technique communication is to present complete algorithm descriptions (rather than implementations) in a consistent manner, and in a centralized location. This book is the outcome of developing such a strategy that not only provides a methodology for standardized algorithm descriptions, but provides a large corpus of complete and consistent algorithm descriptions in a single centralized location.

The algorithms described in this work are practical, interesting, and fun, and the goal of this project was to promote these features by making algorithms from the field more accessible, usable, and understandable. This project was developed over a number years though a lot of writing, discussion, and revision. The content was developed and released publicly under a permissive license on the website http://www.CleverAlgorithms.com, where forerunning technical reports and the content of this book are freely available. I hope that this project has succeeded in some small way and that you too can enjoy applying, learning, and playing with Clever Algorithms.

About the author

Jason Brownlee has a Bachelors degree in Applied Science, a Masters in Information Technology and a Ph.D. in Computer Science from Swinburne University of Technology in Melbourne, Australia. The subject of Jason's Masters research was Niching Genetic Algorithms. Jason's Ph.D. work was in the area of Artificial Immune Systems and involved research into extending the state of Clonal Selection inspired machine learning algorithms and devising new techniques inspired by the structure and function of the acquired immune system. Jason has earned a living as a Consultant on numerous enterprise-level information technology projects in retail, energy, and information services sectors. Jason has also worked as a Software Engineer investigating the use of intelligent agent technology in geospatial and information services domains in the defense sector. Jason has a long standing passion for both practical software engineering and basic research into machine learning and has developed and released many reports, software plug-ins, and software tools. Jason also enjoys writing and maintains a blog located at http://www.neverreadpassively.com and can be followed on twitter at http://twitter.com/jbrownlee.

Acknowledgments

Jason Brownlee would like to sincerely thank Daniel Angus for early discussions that lead to the inception of this book project. Jason would like to thank Ying Liu for her unrelenting support and patience throughout the development of the project.

Contents

Part I

Background

Chapter 1

Introduction

Welcome to Clever Algorithms! This is a handbook of recipes for computational problem solving techniques from the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics. Clever Algorithms are interesting, practical, and fun to learn about and implement. Research scientists may be interested in browsing algorithm inspirations in search of an interesting system or process analogues to investigate. Developers and software engineers may compare various problem solving algorithms and technique-specific guidelines. Practitioners, students, and interested amateurs may implement state-of-the-art algorithms to address business or scientific needs, or simply play with the fascinating systems they represent.

This introduction chapter provides relevant background information on Artificial Intelligence and Algorithms. The core of the book provides a large corpus of algorithm described in a complete and consistent manner. The final chapter covers some advanced topics to consider once a number of algorithms have been mastered. This book has been designed as a reference text rather then being read cover-to-cover, where specific techniques are looked up, or where the algorithms across whole fields of study are browsed. This book is an algorithm handbook and a technique guidebook, and I hope you find something useful.

1.1 What is AI

1.1.1 Artificial Intelligence

The field of classical *Artificial Intelligence* (AI) coalesced in the 1950s drawing on an understanding of the brain from neuroscience, the new mathematics of information theory, control theory referred to as cybernetics, and the dawn of the digital computer. AI is a cross-disciplinary field of research generally concerned with developing and investigating systems that operate or act intelligently. It is generally considered a discipline in the field of computer science given the strong focus on computation.

Russell and Norvig provide a perspective that defines Artificial Intelligence in four categories: (1) systems that think like humans, (2) systems that act like humans, (3)

systems that think rationally, (4) systems that act rationally [195]. In their definition, acting like a human suggests that a system can do some specific things humans can do, this includes fields such as the Turing test, natural language processing, automated reasoning, knowledge representation, machine learning, computer vision, and robotics. Thinking like a human suggests systems that model the cognitive information processing properties of humans, for example a general problem solver and systems that build internal models of their world. Thinking rationally suggests laws of rationalism and structured thought, such as syllogisms and formal logic. Finally, acting rationally suggests systems that do rational things such as expected utility maximization and rational agents.

Luger and Stubblefield suggest that AI is a sub-field of computer science concerned with the automation of intelligence, and like other sub-fields of computer science has both theoretical concerns (*how and why do the systems work?*) and application concerns (*where and when can the systems be used?*) [152]. They suggest a strong empirical focus to research, because although there may be a strong desire for mathematical analysis, the systems themselves defy analysis given their complexity. The machines and software investigated in AI are not black boxes, rather analysis proceeds by observing the systems interactions with their environment, followed by an internal assessment of the system to relate its structure back to their behavior.

Artificial Intelligence is therefore concerned with investigating mechanisms that underlie intelligence and intelligence behavior. The traditional approach toward designing and investigating AI (the so-called 'good old fashioned' AI) has been to employ a symbolic basis for these mechanisms. A newer approach historically referred to as scruffy artificial intelligence or soft computing does not necessarily use a symbolic basis, instead patterning these mechanisms after biological or natural processes. This represents a modern paradigm shift in interest from symbolic knowledge representations, to inference strategies for adaptation and learning, and has been referred to as neat versus scruffy approaches to AI. The neat philosophy is concerned with formal symbolic models of intelligence that can explain *why* they work, whereas the scruffy philosophy is concerned with intelligent strategies that explain *how* they work [206].

Neat AI

The traditional stream of AI concerns a top down perspective of problem solving, generally involving symbolic representations and logic processes that most importantly can explain why they work. The successes of this prescriptive stream include a multitude of specialist approaches such as rule-based expert systems, automatic theorem provers, and operations research techniques that underly modern planning and scheduling software. Although traditional approaches have resulted in significant success they have their limits, most notably scalability. Increases in problem size result in an unmanageable increase in the complexity of such problems meaning that although traditional techniques can guarantee an optimal, precise, or true solution, the computational execution time or computing memory required can be intractable.

Scruffy AI

There have been a number of thrusts in the field of AI toward less crisp techniques that are able to locate approximate, imprecise, or partially-true solutions to problems with a reasonable cost of resources. Such approaches are typically *descriptive* rather than *prescriptive*, describing a process for achieving a solution (how), but not explaining why they work (like the neater approaches).

Scruffy AI approaches are defined as relatively simple procedures that result in complex emergent and self-organizing behavior that can defy traditional reductionist analyses, the effects of which can be exploited for quickly locating approximate solutions to intractable problems. A common characteristic of such techniques is the incorporation of randomness in their processes resulting in robust probabilistic and stochastic decision making contrasted to the sometimes more fragile determinism of the crisp approaches. Another important common attribute is the adoption of an inductive rather than deductive approach to problem solving, generalizing solutions or decisions from sets of specific observations made by the system.

1.1.2 Natural Computation

An important perspective on scruffy Artificial Intelligence is the motivation and inspiration for the core information processing strategy of a given technique. Computers can only do what they are instructed, therefore a consideration is to distill information processing from other fields of study, such as the physical world and biology. The study of biologically motivated computation is called Biologically Inspired Computing [46], and is one of three related fields of Natural Computing [81, 82, 176]. Natural Computing is an interdisciplinary field concerned with the relationship of computation and biology, which in addition to Biologically Inspired Computing is also comprised of Computationally Motivated Biology and Computing with Biology [177, 154].

Biologically Inspired Computation

Biologically Inspired Computation is computation inspired by biological metaphor, also referred to as *Biomimicry*, and *Biomemetics* in other engineering disciplines [38, 25]. The intent of this field is to devise mathematical and engineering tools to generate solutions to computation problems. The field involves using procedures for finding solutions abstracted from the natural world for addressing computationally phrased problems.

Computationally Motivated Biology

Computationally Motivated Biology involves investigating biology using computers. The intent of this area is to use information sciences and simulation to model biological systems in digital computers with the aim to replicate and better understand behaviors in biological systems. The field facilitates the ability to better understand life-as-it-is and investigate life-as-it-could-be. Typically, work in this sub-field is not concerned with the construction of mathematical and engineering tools, rather it is focused on simulating

natural phenomena. Common examples include Artificial Life, Fractal Geometry (L-systems, Iterative Function Systems, Particle Systems, Brownian motion), and Cellular Automata. A related field is that of Computational Biology generally concerned with modeling biological systems and the application of statistical methods such as in the sub-field of Bioinformatics.

Computation with Biology

Computation with Biology is the investigation of substrates other than silicon in which to implement computation [1]. Common examples include molecular or DNA Computing and Quantum Computing.

1.1.3 Computational Intelligence

Computational Intelligence is a modern name for the sub-field of AI concerned with sub-symbolic (also called messy, scruffy, and soft) techniques. Computational Intelligence describes techniques that focus on *strategy* and *outcome*. The field broadly covers sub-disciplines that focus on adaptive and intelligence systems, not limited to: Evolutionary Computation, Swarm Intelligence (Particle Swarm and Ant Colony Optimization), Fuzzy Systems, Artificial Immune Systems, and Artificial Neural Networks [57, 178]. This section provides a brief summary of the each of the five primary areas of study.

Evolutionary Computation

A paradigm that is concerned with the investigation of systems inspired by the neo-Darwinian theory of evolution by means of natural selection. Popular evolutionary algorithms include the Genetic Algorithm, Evolution Strategy, Genetic and Evolutionary Programming, and Differential Evolution [6, 7]. The evolutionary process is considered an adaptive strategy and is typically applied to search and optimization domains [104, 120].

Swarm Intelligence

A paradigm that considers collective intelligence as a behavior that emerges through the interaction and cooperation of large numbers of lesser intelligent agents. The paradigm consists of two dominant sub-fields (1) Ant Colony Optimization that investigates probabilistic algorithms inspired by the stigmergy and foraging behavior of ants [30, 54], and (2) Particle Swarm Optimization that investigates probabilistic algorithms inspired by the flocking and foraging behavior of birds and fish [204]. Like evolutionary computation, swarm intelligence-based techniques are considered adaptive strategies and are typically applied to search and optimization domains.

Artificial Neural Networks

Neural Networks are a paradigm that is concerned with the investigation of architectures and learning strategies inspired by the modeling of neurons in the brain [28]. Learning strategies are typically divided into supervised and unsupervised which manage environmental feedback in different ways. Neural network learning processes are considered adaptive learning and are typically applied to function approximation and pattern recognition domains.

Fuzzy Intelligence

Fuzzy Intelligence is a paradigm that is concerned with the investigation of fuzzy logic, which is a form of logic that is not constrained to true and false like propositional logic, but rather functions which define approximate truth or degrees of truth [246]. Fuzzy logic and fuzzy systems are a logic system used as a reasoning strategy and are typically applied to expert system and control system domains.

Artificial Immune Systems

A collection of approaches inspired by the structure and function of the acquired immune system of vertebrates. Popular approaches include clonal selection, negative selection, dendritic cell algorithm, and immune network algorithms. The immune-inspired adaptive processes vary in strategy and show similarities to the fields of Evolutionary Computation and Artificial Neural Networks, and are typically used for optimization and pattern recognition domains [47].

1.1.4 Metaheuristics

Another popular name for the strategy-outcome perspective of scruffy AI is *Metaheuristics*. A heuristic is an algorithm that locates 'good enough' solutions to a problem without concern for whether the solution can be proven to be correct or optimal [159]. Heuristic methods trade-off concerns such as precision, quality, and accuracy in favor of computational effort (space and time efficiency). The Greedy search procedure that only takes cost-improving steps is an example of heuristic method.

Like heuristics, Metaheuristic may be considered a general algorithmic framework that can be applied to different optimization problems with relative few modifications to make them adapted to a specific problem [99, 222]. The difference is that Metaheuristics are intended to extend the capabilities of heuristics by combining one or more heuristic methods (referred to as procedures) using a higher-level strategy (hence 'meta'). A procedure in a metaheuristic is considered black-box in that little (if any) prior knowledge is known about it by the meta-heuristic, and as such it may be replaced with a different procedure. Procedures may be as simple as the manipulation of a representation, or as complex as another complete metaheuristic. Some examples of metaheuristics include iterated local search, tabu search, the genetic algorithm, ant colony optimization, and simulated annealing.

Blum and Roli outline nine properties of metaheuristics [29], as follows:

- Metaheuristics are strategies that "guide" the search process.
- The goal is to efficiently explore the search space in order to find (near-)optimal solutions.
- Techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes.
- Metaheuristic algorithms are approximate and usually non-deterministic.
- They may incorporate mechanisms to avoid getting trapped in confined areas of the search space.
- The basic concepts of metaheuristics permit an abstract level description.
- Metaheuristics are not problem-specific.
- Metaheuristics may make use of domain-specific knowledge in the form of heuristics that are controlled by the upper level strategy.
- Todays more advanced metaheuristics use search experience (embodied in some form of memory) to guide the search.

Hyperheuristics are yet another extension that focuses on heuristics that modify their parameters (online or offline) to improve the efficacy of solution, or the efficiency of the computation. Hyperheuristics provide high-level strategies that may employ machine learning and adapt their search behavior by modifying the application of the sub-procedures or even which procedures are used (operating on the space of heuristics which in turn operate within the problem domain) [35, 36].

1.1.5 Clever Algorithms

This book is concerned with Clever Algorithms which are algorithms drawn from many sub-fields of Artificial Intelligence not limited to the scruff fields of Biologically Inspired Computation, Computational Intelligence and Metaheuristics. The term *Clever Algorithms* is intended to unify a collection of interesting and useful computational tools under a consistent and accessible banner. An alternative name (*Inspired Algorithms*) was considered, although ultimately rejected given that not all of the algorithms to be described in the project have an inspiration (specifically a biological or physical inspiration) for their computational strategy. The set of algorithms described in this book may generally be referred to as 'unconventional optimization algorithms' (for example, see [40]), as optimization is the main form of computation provided by the listed approaches. A technically more appropriate name for these approaches is Stochastic Global Optimization (for example, see [234] and [153]).

1.2 Problems

Algorithms from the fields of Computational Intelligence, Biologically Inspired Computing, and Metaheuristics are applied to difficult problems, to which more traditional approaches may not be suited. Michalewicz and Fogel propose five reasons why problems may be difficult [159] (page 11):

- The number of possible solutions in the search space is so large as to forbid an exhaustive search for the best answer.
- The problem is so complicated that just to facilitate any answer at all, we have to use such simplified models of the problem that any result is essentially useless.
- The evaluation function that describes the quality of any proposed solution is noisy or varies with time, thereby requiring not just a single solution but an entire series of solutions.
- The possible solutions are so heavily constrained that constructing even one feasible answer is difficult, let alone searching for an optimal solution.
- The person solving the problem is inadequately prepared or imagines some psychological barrier that prevents them from discovering a solution.

This section introduces two problem formalisms that embody many of the most difficult problems faced by Artificial and Computational Intelligence. They are: Function Optimization and Function Approximation. Each class of problem is described in terms of its general properties, a formalism, and a set of specialized sub-problems. These problem classes provide a tangible framing of the algorithmic techniques described throughout the work.

1.2.1 Function Optimization

Real-world optimization problems and generalizations thereof can be drawn from most fields of science, engineering, and information technology (for a sample see [2, 225]). Importantly, optimization problems have had a long tradition in the fields of Artificial Intelligence in motivating basic research into new problem solving techniques, and for investigating and verifying systemic behavior against benchmark problem instances.

Problem Description

Mathematically, optimization is defined as the search for a combination of parameters commonly referred to as decision variables ($x = \{x_1, x_2, x_3, \ldots x_n\}$) which minimize or maximize some ordinal quantity (c) (typically a scalar called a score or cost) assigned by an objective function or cost function (f), under a set of constraints ($g = \{g_1, g_2, g_3, \ldots g_n\}$). For example, a general minimization case would be as follows: $f(x\prime) \leq f(x), \forall x_i \in x$. Constraints may provide boundaries on decision variables (for

example in a real-value hypercube $\Re^n$), or may generally define regions of feasibility and in-feasibility in the decision variable space. In applied mathematics the field may be referred to as Mathematical Programming. More generally the field may be referred to as Global or Function Optimization given the focus on the objective function (for more general information on optimization refer to [123]).

Sub-Fields of Study

The study of optimization is comprised of many specialized sub-fields, based on an overlapping taxonomy that focuses on the principle concerns in the general formalism. For example, with regard to the decision variables, one may consider univariate and multivariate optimization problems. The type of decision variables promotes specialities for continuous, discrete, and permutations of variables. Dependencies between decision variables under a cost function define the fields of Linear Programming, Quadratic Programming, and Nonlinear Programming. A large class of optimization problems can be reduced to discrete sets and are considered in the field of Combinatorial Optimization, to which many theoretical properties are known, most importantly that many interesting and relevant problems cannot be solved by an approach with polynomial time complexity (so-called NP-complete, for example see [174]).

The topography of the response surface for the decision variables under the cost function may be convex, which is a class of functions to which many important theoretical findings have been made, not limited to the fact that location of the local optimal configuration also means the global optimal configuration of decisional variables has been located [32]. Many interesting and real-world optimization problems produce cost surfaces that are non-convex or so called multi-modal[1] (rather than uni-modal) suggesting that there are multiple peaks and valleys. Further, many real-world optimization problems with continuous decision variables cannot be differentiated given their complexity or limited information availability meaning that derivative-based gradient decent methods that are well understood are not applicable, requiring the use of so-called 'direct search' (sample or pattern-based) methods [150]. Real-world objective function evaluation may be noisy, discontinuous, dynamic, and the constraints of real-world problem solving may require an approximate solution in limited time or using resources, motivating the need for heuristic approaches.

1.2.2 Function Approximation

The phrasing of real-world problems in the Function Approximation formalism are among the most computationally difficult considered in the broader field of Artificial Intelligence for reasons including: incomplete information, high-dimensionality, noise in the sample observations, and non-linearities in the target function. This section considers the Function Approximation Formalism and related specialization's as a general

[1] Taken from statistics referring to the centers of mass in distributions, although in optimization it refers to 'regions of interest' in the search space, in particular valleys in minimization, and peaks in maximization cost surfaces.

motivating problem to contrast and compare with Function Optimization.

Problem Description

Function Approximation is the problem of finding a function (f) that approximates a target function (g), where typically the approximated function is selected based on a sample of observations (x, also referred to as the training set) taken from the unknown target function. In machine learning, the function approximation formalism is used to describe general problem types commonly referred to as pattern recognition, such as classification, clustering, and curve fitting (called a decision or discrimination function). Such general problem types are described in terms of approximating an unknown Probability Density Function (PDF), which underlies the relationships in the problem space, and is represented in the sample data. This 'function approximation' perspective of such problems is commonly referred to as statistical machine learning and/or density estimation [85, 28].

Sub-Fields of Study

The function approximation formalism can be used to phrase some of the hardest problems faced by Computer Science, and Artificial Intelligence in particular, such as natural language processing and computer vision. The general process focuses on (i) the collection and preparation of the observations from the target function, (ii) the selection and/or preparation of a model of the target function, and (ii) the application and ongoing refinement of the prepared model. Some important problem-based sub-fields include:

- *Feature Selection* where a feature is considered an aggregation of one-or-more attributes, where only those features that have meaning in the context of the target function are necessary to the modeling function [144, 106].

- *Classification* where observations are inherently organized into labelled groups (classes) and a supervised process models an underlying discrimination function to classify unobserved samples.

- *Clustering* where observations may be organized into groups based on underlying common features, although the groups are unlabeled requiring a process to model an underlying discrimination function without corrective feedback.

- *Curve or Surface Fitting* where a model is prepared that provides a 'best-fit' (called a regression) for a set of observations that may be used for interpolation over known observations and extrapolation for observations outside what has been modelled.

The field of Function Optimization is related to Function Approximation, as many-sub-problems of Function Approximation may be defined as optimization problems. Many of the technique paradigms used for function approximation are differentiated based on the representation and the optimization process used to minimize error or

maximize effectiveness on a given approximation problem. The difficulty of Function Approximation problems centre around (i) the nature of the unknown relationships between attributes and features, (ii) the number (dimensionality) of of attributes and features, and (iii) general concerns of noise in such relationships and the dynamic availability of samples from the target function. Additional difficulties include the incorporation of prior knowledge (such as imbalance in samples, incomplete information and the variable reliability of data), and problems of invariant features (such as transformation, translation, rotation, scaling and skewing of features).

1.3 Unconventional Optimization

Not all algorithms described in this book are for optimization, although, those that are may be referred to as 'unconventional' to differentiate them from the more traditional approaches. Examples of traditional approaches include (but are not not limited) to mathematical optimization algorithms (such as Newton's method and Gradient descent that uses derivatives to locate a local minimum) and direct search methods (such as the Simplex method and the Nelder-Mead method that use a search pattern to locate optima). Unconventional optimization algorithms are designed for the more difficult problem instances, the attributes of which were introduced in Section 1.2.1. This section introduces some common attributes of this class of algorithm.

1.3.1 Black Box Algorithms

Black Box optimization algorithms are those that exploit little, if any, information from a problem domain in order to devise a solution. They are generalized problem solving procedures that may be applied to a range of problems with very little modification [55]. Domain specific knowledge refers to making use of known relationships between solution representations and the objective cost function. Generally speaking, the less domain specific information incorporated into a technique, the more flexible the technique, although the less efficient it will be for a given problem. For example, 'random search' is the most general black box approach and is also the most flexible requiring only the generation of random solutions for a given problem. Random search also has a worst case behavior, that is worse than enumerating an entire search domain given the freedom it has to resample. In practice, the more prior knowledge available about a problem, the more information that should be exploited by a technique in order to efficiently locate a solution for the problem, heuristically or otherwise. Therefore, black box methods are those methods suitable for those problems where little information from the problem domain is available to be used by a problem solving approach.

1.3.2 No Free Lunch

The *No Free Lunch Theorem* of search and optimization by Wolpert and Macready proposes that all black box optimization algorithms are the same for searching for the extremum of a cost function when averaged over all possible functions [242, 241]. The

theorem has caused a lot of pessimism and misunderstanding, particularly in relation to the evaluation and comparison of Metaheuristic and Computational Intelligence algorithms.

The implication of the theorem is that searching for the 'best' general-purpose black box optimization algorithm is irresponsible as no such procedure is theoretically possible. The theory applies to stochastic and deterministic optimization algorithms as well as to algorithms that learn and adjust their search strategy over time. It is invariant to the performance measure used and the representation selected. The theorem is an important contribution to computer science, although its implications are theoretical. The original paper was produced at a time when grandiose generalizations were being made as to algorithm, representation, or configuration superiority. The practical impact of the theory is to encourage practitioners to bound claims of applicability for search and optimization algorithms. Wolpert and Macready encouraged effort be put into devising practical problem classes and into the matching of suitable algorithms to problem classes. Further, they compelled practitioners to exploit domain knowledge in optimization algorithm application, which is now an axiom in the field.

1.3.3 Stochastic Optimization

Stochastic optimization algorithms those that use randomness to elicit non-deterministic behaviors, contrasted to purely deterministic procedures. Most algorithms from the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics may be considered to belong the field of Stochastic Optimization. Algorithms that exploit randomness, are not random in behavior, rather they sample a problem space in a biased manner, focusing on areas of interest and neglecting less interesting areas [210]. A class of techniques that focus on the stochastic sampling of a domain are called Markov Chain Monte Carlo (MCMC) algorithms that provide good average performance, quickly, and generally offer a low chance of the worst case performance. Such approaches are suited to problems with many coupled degrees of freedom, for example large, high-dimensional spaces. MCMC approaches involve stochastically sampling from a target distribution function similar to Monte Carlo simulation methods using a process that resembles a biased Markov chain.

- *Monte Carlo* methods are used for selecting a statistical sample to approximate a given target probability density function and are traditionally used in statistical physics. Samples are drawn sequentially and the process may include criteria for rejecting samples and biasing the sampling locations within high-dimensional spaces.

- *Markov Chain* processes provide a probabilistic model for state transitions or moves within a discrete domain called a walk or a chain of steps. A Markov system is only dependent on the current position in the domain in order to probabilistically determine the next step in the walk.

MCMC techniques combine these two approaches to solve integration and optimization problems in large dimensional spaces by generating samples while exploring the space using a Markov chain process, rather than sequentially or independently [3]. The step generation is configured to bias sampling in more important regions of the domain. Three examples of MCMC techniques include the Metropolis-Hastings algorithm, Simulated annealing for global optimization, and the Gibbs sampler which are commonly employed in the fields of physics, chemistry, statistics, and economics.

1.3.4 Inductive Learning

Many unconventional optimization algorithms employ a process that includes the iterative improvement of candidate solutions against an objective cost function. This process of adaptation is generally a method by which the process obtains characteristics that improve the system's (candidate solution) relative performance in an environment (cost function). This adaptive behavior is commonly achieved through a 'selectionist process' of repetition of the steps: generation, test, and selection. The use of non-deterministic processes mean that the sampling of the domain (the generation step) is typically non-parametric, although guided by past experience.

The method of acquiring information is called inductive learning or learning from example, where the approach uses the implicit assumption that specific examples are representative of the broader information content of the environment, specifically with regard to anticipated need. Many unconventional optimization approaches maintain a single candidate solution, a population of samples, or a compression thereof that provides both an instantaneous representation of all of the information acquired by the process, and the basis for generating and making future decisions.

This method of simultaneously acquiring and improving information from the domain and the optimization of decision making (where to direct future effort) is called the k-armed bandit (two-armed and multi-armed bandit) problem from the field of statistical decision making known as game theory [192, 26]. This formalism considers the capability of a strategy to allocate available resources proportional to the future payoff the strategy is expected to receive. The classic example is the 2-armed bandit problem used by Goldberg to describe the behavior of the genetic algorithm [104]. The example involves an agent that learns which one of the two slot machines provides more return by pulling the handle of each (sampling the domain) and biasing future handle pulls proportional to the expected utility, based on the probabilistic experience with the past distribution of the payoff. The formalism may also be used to understand the properties of inductive learning demonstrated by the adaptive behavior of most unconventional optimization algorithms.

The stochastic iterative process of generate and test can be computationally wasteful, potentially re-searching areas of the problem space already searched, and requiring many trials or samples in order to achieve a 'good enough' solution. The limited use of prior knowledge from the domain (black box) coupled with the stochastic sampling process mean that the adapted solutions are created without top-down insight or instruction can sometimes be interesting, innovative, and even competitive with decades of human

expertise [140].

1.4 Book Organization

The remainder of this book is organized into two parts: *Algorithms* that describes a large number of techniques in a complete and a consistent manner presented in a rough algorithm groups, and *Extensions* that reviews more advanced topics suitable for when a number of algorithms have been mastered.

1.4.1 Algorithms

Algorithms are presented in six groups or kingdoms distilled from the broader fields of study each in their own chapter, as follows:

- *Stochastic Algorithms* that focus on the introduction of randomness into heuristic methods (Chapter 2).
- *Physical Algorithms* that focus on methods inspired by physical and social systems (Chapter 3).
- *Evolutionary Algorithms* that focus on methods in spired by evolution by means of natural selection (Chapter 4).
- *Probabilistic Algorithms* that focus on methods that build models and estimate distributions in search domains (Chapter 5).
- *Swarm Algorithms* that focus on methods that exploit the properties of collective intelligence (Chapter 6).
- *Immune Algorithms* that focus on methods inspired by the adaptive immune system of mammals (Chapter 7).

A given algorithm is more than just a procedure or code listing. Each approach is an island of research and the meta-information that define the context of a technique are just as important to understanding and application as abstract recipes and concrete implementations. A standardized algorithm description was adopted to provide a consistent presentation of algorithms with a mixture of softer narrative descriptions, programmatic descriptions both abstract and concrete, and most importantly useful sources for finding out more information about the technique.

The standardized algorithm description template covers the following subjects:

- *Name*: The algorithm name defines the canonical name used to refer to the technique, in addition to common aliases, abbreviations, and acronyms. The name is used as the heading of an algorithm descriptions.

- *Taxonomy*: The algorithm taxonomy defines where a techniques fits into the field, both the specific subfields of Computational Intelligence and Biologically Inspired Computation as well as the broader field of Artificial Intelligence. The taxonomy also provides a context for determining the relationships between algorithms.

- *Inspiration*: (optional) The inspiration describes the specific system or process that provoked the inception of the algorithm. The inspiring system may non-exclusively be natural, biological, physical, or social. The description of the inspiring system may include relevant domain specific theory, observation, nomenclature, and most important must include those salient attributes of the system that are somehow abstractly or conceptually manifest in the technique.

- *Metaphor*: (optional) The metaphor is a description of the technique in the context of the inspiring system or a different suitable system. The features of the technique are made apparent through an analogous description of the features of the inspiring system. The explanation through analogy is not expected to be literal, rather the method is used as an allegorical communication tool. The inspiring system is not explicitly described, this is the role of the 'inspiration' topic, which represents a loose dependency for this topic.

- *Strategy*: The strategy is an abstract description of the computational model. The strategy describes the information processing actions a technique shall take in order to achieve an objective. The strategy provides a logical separation between a computational realization (procedure) and a analogous system (metaphor). A given problem solving strategy may be realized as one of a number specific algorithms or problem solving systems.

- *Procedure*: The algorithmic procedure summarizes the specifics of realizing a strategy as a systemized and parameterized computation. It outlines how the algorithm is organized in terms of the computation, data structures, and representations.

- *Heuristics*: The heuristics section describes the commonsense, best practice, and demonstrated rules for applying and configuring a parameterized algorithm. The heuristics relate to the technical details of the techniques procedure and data structures for general classes of application (neither specific implementations nor specific problem instances).

- *Code Listing*: The code listing description provides a minimal but functional version of the technique implemented with a programming language. The code description can be typed into an computer and provide a working execution of the technique. The technique implementation also includes a minimal problem instance to which it is applied, and both the problem and algorithm implementations are complete enough to demonstrate the techniques procedure. The description is presented as a programming source code listing with a terse introductory summary.

- *References*: The references section includes a listing of both primary sources of information about the technique as well as useful introductory sources for novices to gain a deeper understanding of the theory and application of the technique. The description consists of hand-selected reference material including books, peer reviewed conference papers, journal articles, and potentially websites.

Source code examples are included in the algorithm descriptions, and the Ruby Programming Language was selected for use throughout the book. Ruby was selected because it supports the procedural programming paradigm that was adopted to ensure that examples can be easily ported to object-oriented and other paradigms. Additionally, Ruby is interpreted meaning the code can be directly executed without an introduced compilation step, and it is free to download and use from the website[2]. Finally, Ruby is concise, expressive, and supports meta-programming features that improve the readability of code examples. All of the source code for the algorithms presented in this book is available from the books website at http://www.CleverAlgorithms.com.

1.4.2 Extensions

There are some some advanced topics that cannot be meaningfully considered until one has a firm grasp of a number of algorithms, and these are discussed at the back of the book. The Advanced Topics chapter addresses topics such as: the use of alternative programming paradigms when implementing clever algorithms, methodologies used when devising entirely new approaches, strategies to consider when testing clever algorithms, visualizing the behavior and results of algorithms, and finally comparing algorithms based on the results they produce using statistical methods. Like the background information provided in this chapter, the extensions provide a gentle introduction and starting point into some advanced topics and plenty of references for seeking a deeper understanding.

1.5 How to Read this Book

This book is a reference text that provides a large compendium of algorithm descriptions. It is a trusted handbook of practical computational recipes to consulted when one is confronted with difficult function optimization and approximation problems. It is also an encompassing guidebook of modern heuristic methods that may be browsed for inspiration, exploration, and general interest.

The audience for this work may be interested with the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics and may count themselves as belonging to one of the following broader groups:

- *Scientists*: Research scientists concerned with theoretically or empirically investigating algorithms, addressing questions such as: *What is the motivating system*

[2]Ruby can be downloaded for free from http://www.ruby-lang.org

and strategy for a given technique? What are some algorithms that may be used in a comparison within a given subfield or across subfields?

- *Engineers*: Programmers and developers concerned with implementing, applying, or maintaining algorithms, addressing questions such as: *What is the algorithm procedure for a given technique? What are the best practice heuristics for employing a given technique?*

- *Students*: Undergraduate and graduate students interested in learning about techniques, addressing questions such as: *What are some interesting algorithms to study? How to implement a given approach?*

- *Amateurs*: Practitioners interested in knowing more about algorithms, addressing questions such as: *What classes of techniques exist and what algorithms do they provide? How to conceptualize the computation of a technique?*

1.6 Further Reading

This book is not an introduction to Artificial Intelligence or related sub-fields, nor is it a field guide for a specific class of algorithms. This section provides some pointers to selected books and articles for those readers seeking a deeper understanding of the fields of study to which the Clever Algorithms described in this book belong.

1.6.1 Artificial Intelligence

Artificial Intelligence is large field of study and many excellent texts have been written to introduce the subject. Russell and Novig's "*Artificial Intelligence: A Modern Approach*" is an excellent introductory text providing a broad and deep review of what the field has to offer and is useful for students and practitioners alike [195]. Luger and Stubblefield's "*Artificial Intelligence: Structures and Strategies for Complex Problem Solving*" is also an excellent reference text, providing a more empirical approach to the field.

1.6.2 Computational Intelligence

Introductory books for the field of Computational Intelligence generally focus on a handful of specific sub-fields and their techniques. Engelbrecht's "*Computational Intelligence: An Introduction*" provides a modern and detailed introduction to the field covering classic subjects such as Evolutionary Computation and Artificial Neural Networks, as well as more recent techniques such as Swarm Intelligence and Artificial Immune Systems [57]. Pedrycz's slightly more dated "*Computational Intelligence: An Introduction*" also provides a solid coverage of the core of the field with some deeper insights into fuzzy logic and fuzzy systems [178].

1.6.3 Biologically Inspired Computation

Computational methods inspired by natural and biologically systems represent a large fraction of the algorithms described in this book. The collection of articles published in de Castro and Von Zuben's "*Recent Developments in Biologically Inspired Computing*" provide a good overview of the state of the field, and the introductory chapter on need for such methods does an excellent job to motivate the field of study [38]. Forbes's "*Imitation of Life: How Biology Is Inspiring Computing*" set's the scene for Natural Computing and the interrelated disciplines, of which Biologically Inspired Computing is but one useful example [81]. Finally, Benyus's "*Biomimicry: Innovation Inspired by Nature*" provides a good introduction into the broader related field of a new frontier in science and technology that involves building systems inspired by an understanding of biological systems [25].

1.6.4 Metaheuristics

The field of Metaheuristics was initially constrained to heuristics for applying classical optimization procedures, although has expanded to encompass a broader and diverse set of techniques. Michalewicz and Fogel's "*How to Solve It: Modern Heuristics*" provides a practical tour of heuristic methods with a consistent set of worked examples [159]. Glover and Kochenberger's "*Handbook of Metaheuristics*" provides a solid introduction into a broad collection of techniques and their capabilities [99].

1.6.5 The Ruby Programming Language

The Ruby Programming Language is a multi-paradigm dynamic language that appeared in approximately 1995. It's meta-programming capabilities coupled with concise and readable syntax have made it a popular language of choice for web development, scripting, and application development. The classic reference text for the language is Thomas, Fowler, and Hunt's "*Programming Ruby: The Pragmatic Programmers' Guide*" referred to as the 'pickaxe book' because of the picture of the pickaxe on the cover [223]. An updated edition is available that covers version 1.9 (compared to 1.8 in the cited version) that will work just as well for use as a reference for the examples in this book. Flanagan and Matsumoto's "*The Ruby Programming Language*" also provides a seminal reference text with contributions from Yukihiro Matsumoto, the author of the language [70].

Part II

Algorithms

Chapter 2

Stochastic Algorithms

2.1 Overview

This chapter describes Stochastic Algorithms. The majority of the algorithms to be described in the Clever Algorithms project are comprised of probabilistic and stochastic processes. What differentiates the 'stochastic algorithms' in this chapter from the remaining algorithms is the specific lack of i) an inspiring system, and ii) a metaphorical explanation. Both 'inspiration' and 'metaphor' refer to the descriptive elements in the standardized algorithm description.

These described algorithms are predominately global optimization algorithms and metaheuristics that manage the application of an embedded neighborhood exploring (local) search procedure. As such, with the exception of 'Stochastic Hill Climbing' and 'Random Search' the algorithms may be considered extensions of the multi-start search (also known as multi-restart search). The set of algorithms provide various different strategies by which 'better' and varied starting points can be generated and issued to a neighborhood searching technique for refinement, a process that is repeated with potentially improving or unexplored areas to search.

2.2 Random Search

Random Search, RS, Blind Random Search, Blind Search, Pure Random Search, PRS

2.2.1 Taxonomy

Random search belongs to the fields of Stochastic Optimization and Global Optimization. Random search is a direct search method as it does not require derivatives to search a continuous domain. This base approach is related to techniques that provide small improvements such as Directed Random Search, and Adaptive Random Search (Section 2.3).

2.2.2 Strategy

The strategy of Random Search is to sample solutions from across the entire search space using a uniform probability distribution. Each future sample is independent of the samples that come before it.

2.2.3 Procedure

Algorithm 1 provides a pseudo-code listing of the Random Search Algorithm for minimizing a cost function.

Algorithm 1: Pseudo Code Listing for the Random Search Algorithm.

```
Input: NumIterations, ProblemSize, SearchSpace
Output: Best
Best ← 0;
foreach iter_i ∈ NumIterations do
    candidate_i = RandomSolution(ProblemSize, SearchSpace);
    if Cost(candidate_i) < Cost(Best) then
        Best ← candidate_i;
    end
end
return Best;
```

2.2.4 Heuristics

- Random search is minimal in that it only requires a candidate solution construction routine and a candidate solution evaluation routine, both of which may be calibrated using the approach.
- The worst case performance for Random Search for locating the optima is worse than an Enumeration of the search domain, given that Random Search has no memory and can blindly resample.

- Random Search can return a reasonable approximation of the optimal solution within a reasonable time under low problem dimensionality, although the approach does not scale well with problem size (such as the number of dimensions).
- Care must be taken with some problem domains to ensure that random candidate solution construction is unbiased
- The results of a Random Search can be used to seed another search technique, like a local search technique (such as the Hill Climbing algorithm) that can be used to locate the best solution in the neighborhood of the 'good' candidate solution.

2.2.5 Code Listing

Listing 2.1 provides an example of the Random Search Algorithm implemented in the Ruby Programming Language. In the example, the algorithm runs for a fixed number of iterations and returns the best candidate solution discovered. The example problem is an instance of a continuous function optimization that seeks $minf(x)$ where $f = \sum_{i=1}^{n} x_i^2$, $-5.0 \leq x_i \leq 5.0$ and $n = 2$. The optimal solution for this basin function is $(v_0, \ldots, v_{n-1}) = 0.0$.

```ruby
def cost(candidate_vector)
  return candidate_vector.inject(0) {|sum, x| sum + (x ** 2.0)}
end

def random_solution(problem_size, search_space)
  return Array.new(problem_size) do |i|
    search_space[i][0] + ((search_space[i][1] - search_space[i][0]) * rand())
  end
end

def search(max_iterations, problem_size, search_space)
  best = nil
  max_iterations.times do |iter|
    candidate = {}
    candidate[:vector] = random_solution(problem_size, search_space)
    candidate[:cost] = cost(candidate[:vector])
    best = candidate if best.nil? or candidate[:cost] < best[:cost]
    puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
  end
  return best
end

max_iterations = 100
problem_size = 2
search_space = Array.new(problem_size) {|i| [-5, +5]}

best = search(max_iterations, problem_size, search_space)
puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.1: Random Search Algorithm in the Ruby Programming Language

2.2.6 References

Primary Sources

There is no seminal specification of the Random Search algorithm, rather there are discussions of the general approach and related random search methods from the 1950's through to the 1970's. This was around the time that pattern and direct search methods were actively researched. Brooks is credited with the so-called 'pure random search' [33]. Two seminal reviews of 'random search methods' of the time include: Karnopp [133] and prhaps Kul'chitskii [145].

Learn More

For overviews of into Random Search Methods see Zhigljavsky [247], Solis and Wets [209], and also White [235] who provides an excellent review article. Spall provides a detailed overview of the field of Stochastic Optimization, including the Random Search method [210] (for example, see Chapter 2). For a shorter introduction by Spall, see [211] (specifically Section 6.2). Also see Zabinsky for another detailed review of the broader field [245].

2.3 Adaptive Random Search

Adaptive Random Search, ARS, Adaptive Step Size Random Search, ASSRS, Variable Step-Size Random Search.

2.3.1 Taxonomy

The Adaptive Random Search algorithm belongs to the general set of approaches known as Stochastic Optimization and Global Optimization. It is a direct search method in that it does not require derivatives to navigate the search space. Adaptive Random Search is an extension of the Random Search (Section 2.2) and Localized Random Search algorithms.

2.3.2 Strategy

The Adaptive Random Search algorithm was designed to address the limitations of the fixed step size in the Localized Random Search algorithm. The strategy for Adaptive Random Search is to continually approximate the optimal step size required to reach the global optimum in the search space. This is achieved by trialling and adopting smaller or larger step sizes only if they result in an improvement in the search performance.

The Strategy of the Adaptive Step Size Random Search algorithm (the specific technique reviewed) is to trial a larger step in each iteration and adopt the larger step if it results in an improved result. Very large step sizes are trialled in the same manner although with a much lower frequency. This strategy of preferring large moves is intended to allow the technique to escape local optimal. Smaller step sizes are adopted if no improvement is made for an extended period.

2.3.3 Procedure

Algorithm 2 provides a pseudo-code listing of the Adaptive Random Search Algorithm for minimizing a cost function based on the specification for 'Adaptive Step-Size Random Search' by Schummer and Steiglitz [199].

2.3.4 Heuristics

- Adaptive Random Search was designed for continuous function optimization problem domains.
- Candidates with equal cost should be considered improvements to allow the algorithm to make progress across plateaus in the response surface.
- Adaptive Random Search may adapt the search direction in addition to the step size.
- The step size may be adapted for all parameters, or for each parameter individually.

Algorithm 2: Pseudo Code Listing for the Adaptive Random Search Algorithm.

Input: $Iter_{max}$, ProblemSize, SearchSpace, $StepSize_{init}$, $StepSizeF_{small}$, $StepSizeF_{large}$, $StepSizeIter_{large}$, $NoChng_{max}$
Output: Current

$nochng_{count} \leftarrow 0$;
$step_{size} \leftarrow$ `InitializeStepSize`(SearchSpace, $StepSize_{init}$);
Current $\leftarrow$ `RandomSolution`(ProblemSize, SearchSpace);
foreach $iter_i \in Iter_{max}$ **do**
  $candidate_1 \leftarrow$ `TakeStep`(SearchSpace, Current, $step_{size}$);
  $largestep_{size} \leftarrow 0$;
  if $iter_i \bmod StepSizeIter_{large}$ **then**
    $largestep_{size} \leftarrow step_{size} \times StepSizeF_{large}$;
  end
  else
    $largestep_{size} \leftarrow step_{size} \times StepSizeF_{small}$;
  end
  $candidate_2 \leftarrow$ `TakeStep`(SearchSpace, Current, $largestep_{size}$);
  if `Cost`($candidate_1$)$\leq$`Cost`(Current) *or* `Cost`($candidate_2$)$\leq$`Cost`(Current) **then**
    if `Cost`($candidate_2$)$<$`Cost`($candidate_1$) **then**
      Current $\leftarrow candidate_2$;
      $step_{size} \leftarrow largestep_{size}$;
    end
    else
      Current $\leftarrow candidate_1$;
    end
    $nochng_{count} \leftarrow 0$;
  end
  else
    $nochng_{count} \leftarrow nochng_{count} + 1$;
    if $nochng_{count} > NoChng_{max}$ **then**
      $nochng_{count} \leftarrow 0$;
      $step_{size} \leftarrow \frac{step_{size}}{StepSizeF_{small}}$
    end
  end
end
return Current;

2.3.5 Code Listing

Listing 2.2 provides an example of the Adaptive Random Search Algorithm implemented in the Ruby Programming Language, based on the specification for 'Adaptive Step-Size Random Search' by Schummer and Steiglitz [199]. In the example, the algorithm runs for a fixed number of iterations and returns the best candidate solution discovered. The example problem is an instance of a continuous function optimization that seeks $minf(x)$ where $f = \sum_{i=1}^{n} x_i^2$, $-5.0 < x_i < 5.0$ and $n = 2$. The optimal solution for this basin function is $(v_0, \ldots, v_{n-1}) = 0.0$.

```
def cost(candidate_vector)
  return candidate_vector.inject(0) {|sum, x| sum + (x ** 2.0)}
end

def random_solution(problem_size, search_space)
  return Array.new(problem_size) do |i|
    search_space[i][0] + ((search_space[i][1] - search_space[i][0]) * rand())
  end
end

def take_step(problem_size, search_space, current, step_size)
  step = []
  problem_size.times do |i|
    max, min = current[i]+step_size, current[i]-step_size
    max = search_space[i][1] if max > search_space[i][1]
    min = search_space[i][0] if min < search_space[i][0]
    step << min + ((max - min) * rand)
  end
  return step
end

def large_step_size(iteration, step_size, small_factor, large_factor, factor_multiple)
  if iteration.modulo(factor_multiple)
    return step_size * large_factor
  end
  return step_size * small_factor
end

def search(max_iterations, problem_size, search_space, init_factor, small_factor,
    large_factor, factor_multiple, max_no_improvements)
  step_size = (search_space[0][1]-search_space[0][0]) * init_factor
  current, count = {}, 0
  current[:vector] = random_solution(problem_size, search_space)
  current[:cost] = cost(current[:vector])
  max_iterations.times do |iter|
    step, bigger_step = {}, {}
    step[:vector] = take_step(problem_size, search_space, current[:vector], step_size)
    step[:cost] = cost(step[:vector])
    bigger_step_size = large_step_size(iter, step_size, small_factor, large_factor,
        factor_multiple)
    bigger_step[:vector] = take_step(problem_size, search_space, current[:vector],
        bigger_step_size)
    bigger_step[:cost] = cost(bigger_step[:vector])
```

```
    if step[:cost] <= current[:cost] or bigger_step[:cost] <= current[:cost]
      if bigger_step[:cost] < step[:cost]
        step_size, current = bigger_step_size, bigger_step
      else
        current = step
      end
      count = 0
    else
      count += 1
      count, stepSize = 0, (step_size/small_factor) if count >= max_no_improvements
    end
    puts " > iteration #{(iter+1)}, best=#{current[:cost]}"
  end
  return current
end

max_iterations = 1000
problem_size = 2
search_space = Array.new(problem_size) {|i| [-5, +5]}
init_factor = 0.05
small_factor = 1.3
large_factor = 3.0
factor_multiple = 10
max_no_improvements = 30

best = search(max_iterations, problem_size, search_space, init_factor, small_factor,
    large_factor, factor_multiple, max_no_improvements)
puts "Done. Best Solution: cost=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.2: Adaptive Random Search Algorithm in the Ruby Programming Language

2.3.6 References

Primary Sources

Many works in the 1960s and 1970s experimented with variable step sizes for Random Search methods. Schummer and Steiglitz are commonly credited the adaptive step size procedure, which they called 'Adaptive Step-Size Random Search' [199]. Their approach only modifies the step size based on an approximation of the optimal step size required to reach the global optima. Kregting and White review adaptive random search methods and propose an approach called 'Adaptive Directional Random Search' that modifies both the algorithms step size and direction in response to the cost function [143].

Learn More

White reviews extensions to Rastrigin's 'Creeping Random Search' [188] (fixed step size) that use probabilistic step sizes drawn stochastically from uniform and probabilistic distributions [235]. White also reviews works that propose dynamic control strategies for the step size, such as Karnopp [133] who proposes increases and decreases to the step size based on performance over very small numbers of trials. Schrack and Choit review

random search methods that modify their step size in order to approximate optimal moves while searching, including the property of reversal [198]. Masri, et al. describe an adaptive random search strategy that alternates between periods of fixed and variable step sizes [158].

2.4 Stochastic Hill Climbing

Stochastic Hill Climbing, SHC, Random Hill Climbing, RHC, Random Mutation Hill Climbing, RMHC.

2.4.1 Taxonomy

The Stochastic Hill Climbing algorithm is a Stochastic Optimization algorithm and is a Local Optimization algorithm (contrasted to Global Optimization). It is a direct search technique, as it does not require derivatives of the search space. Stochastic Hill Climbing is an extension of deterministic hill climbing algorithms such as Simple Hill Climbing (first-best neighbor), Steepest-Ascent Hill Climbing (best neighbor), and a parent of approaches such as Parallel Hill Climbing and Random-Restart Hill Climbing.

2.4.2 Strategy

The strategy of the Stochastic Hill Climbing algorithm is iterate the process of randomly selecting a neighbor for a candidate solution and only accept it if it results in an improvement. The strategy was proposed to address the limitations of deterministic hill climbing techniques that were likely to get stuck in local optima due to their greedy acceptance of neighboring moves.

2.4.3 Procedure

Algorithm 3 provides a pseudo-code listing of the Stochastic Hill Climbing algorithm for minimizing a cost function, specifically the Random Mutation Hill Climbing algorithm described by Forrest and Mitchell applied to a maximization optimization problem [83].

Algorithm 3: Pseudo Code Listing for the Stochastic Hill Climbing algorithm.

```
Input: Iter_max, ProblemSize
Output: Current
Current ← RandomSolution(ProblemSize);
foreach iter_i ∈ Iter_max do
    Candidate ← RandomNeighbor(Current);
    if Cost(Candidate) ≥ Cost(Current) then
        Current ← Candidate;
    end
end
return Current;
```

2.4.4 Heuristics

- Stochastic Hill Climbing was designed to be used in discrete domains with explicit neighbors such as combinatorial optimization (compared to continuous function

optimization).

- The algorithm's strategy may be applied to continuous domains by making use of a step-size to define candidate-solution neighbors (such as Localized Random Search and Fixed Step-Size Random Search).
- Stochastic Hill Climbing is a local search technique (compared to global search) and may be used to refine a result after the execution of a global search algorithm.
- Even though the technique uses a stochastic process, it can still get stuck in local optima.
- Neighbors with better or equal cost should be accepted, allowing the technique to navigate across plateaus in the response surface.
- The algorithm can be restarted and repeated a number of times after it converges to provide an improved result (called Multiple Restart Hill Climbing).
- The procedure can be applied to multiple candidate solutions concurrently, allowing multiple algorithm runs to be performed at the same time (called Parallel Hill Climbing).

2.4.5 Code Listing

Listing 2.3 provides an example of the Stochastic Hill Climbing algorithm implemented in the Ruby Programming Language, specifically the Random Mutation Hill Climbing algorithm described by Forrest and Mitchell [83]. The algorithm is executed for a fixed number of iterations and is applied to a binary string optimization problem called 'One Max'. The objective of this maximization problem is to prepare a string of all '1' bits, where the cost function only reports the number of bits in a given string.

```
1  def cost(bitstring)
2    return bitstring.inject(0) {|sum,x| sum = sum + ((x=='1') ? 1 : 0)}
3  end
4
5  def random_solution(problem_size)
6    return Array.new(problem_size){|i| (rand<0.5) ? "1" : "0"}
7  end
8
9  def random_neighbor(bitstring)
10   mutant = Array.new(bitstring)
11   pos = rand(bitstring.length)
12   mutant[pos] = (mutant[pos]=='1') ? '0' : '1'
13   return mutant
14 end
15
16 def search(max_iterations, problem_size)
17   candidate = {}
18   candidate[:vector] = random_solution(problem_size)
19   candidate[:cost] = cost(candidate[:vector])
```

```
20   max_iterations.times do |iter|
21     neighbor = {}
22     neighbor[:vector] = random_neighbor(candidate[:vector])
23     neighbor[:cost] = cost(neighbor[:vector])
24     candidate = neighbor if neighbor[:cost] >= candidate[:cost]
25     puts " > iteration #{(iter+1)}, best=#{candidate[:cost]}"
26     break if candidate[:cost] == problem_size
27   end
28   return candidate
29 end
30
31 max_iterations = 1000
32 problem_size = 64
33
34 best = search(max_iterations, problem_size)
35 puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].join}"
```

Listing 2.3: Stochastic Hill Climbing algorithm in the Ruby Programming Language

2.4.6 References

Primary Sources

Perhaps the most popular implementation of the Stochastic Hill Climbing algorithm is by Forrest and Mitchell, who proposed the Random Mutation Hill Climbing (RMHC) algorithm (with communication from Richard Palmer) in a study that investigated the behavior of the genetic algorithm on a deceptive class of (discrete) bit-string optimization problem called 'royal road' functions [83]. The RMHC was compared to two other hill climbing algorithms in addition to the genetic algorithm, specifically: the Steepest-Ascent Hill Climber, and the Next-Ascent Hill Climber. This study was then followed up by Mitchell and Holland [164]

Jules and Wattenberg were also early to consider stochastic hill climbing as an approach to compare to the genetic algorithm [132]. Skalak applied the RMHC algorithm to a single long bit-string that represented a number of prototype vectors for use in classification [205].

Learn More

The Stochastic Hill Climbing algorithm is related to the genetic algorithm without crossover. Simplified version's of the approach are investigated for bit-string based optimization problems with the population size of the genetic algorithm reduced to one. The general technique has been investigated under the names Iterated Hillclimbing [167], ES(1+1,m,hc) [168], Random Bit Climber [44], and (1+1)-Genetic Algorithm [5]. This main difference between RMHC and ES(1+1) is that the latter uses a fixed probability of a mutation for each discrete element of a solution (meaning the neighborhood size is probabilistic), whereas RMHC will only stochastically modify one element.

2.5 Iterated Local Search

Iterated Local Search, ILS.

2.5.1 Taxonomy

Iterated Local Search is a Metaheuristic and a Global Optimization technique. It is an extension of Mutli Start Search and may be considered a parent of many two-phase search approaches such as Greedy Randomized Adaptive Search Procedure (Section 2.8) and Variable Neighborhood Search (Section 2.7).

2.5.2 Strategy

The objective of Iterated Local Search is to improve upon stochastic Mutli-Restart Search by sampling in the broader neighborhood of candidate solutions and using a Local Search technique to refine solutions to their local optima. Iterated Local Search explores a sequence of solutions created as perturbations of the current best solution, the result of which is refined using an embedded heuristic.

2.5.3 Procedure

Algorithm 4 provides a pseudo-code listing of the Iterated Local Search algorithm for minimizing a cost function.

Algorithm 4: Pseudo Code for the Iterated Local Search algorithm.

```
   Input:
   Output: S_best
 1 S_best ← ConstructInitialSolution();
 2 S_best ← LocalSearch();
 3 SearchHistory ← S_best;
 4 while ¬ StopCondition() do
 5 |   S_candidate ← Perturbation(S_best, SearchHistory);
 6 |   S_candidate ← LocalSearch(S_candidate);
 7 |   SearchHistory ← S_candidate;
 8 |   if AcceptanceCriterion(S_best, S_candidate, SearchHistory) then
 9 |   |   S_best ← S_candidate;
10 |   end
11 end
12 return S_best;
```

2.5.4 Heuristics

- Iterated Local Search was designed for and has been predominately applied to discrete domains, such as combinatorial optimization problems.

- The perturbation of the current best solution should be in a neighborhood beyond the reach of the embedded heuristic and should not be easily undone.
- Perturbations that are too small make the algorithm too greedy, perturbations that are too large make the algorithm too stochastic.
- The embedded heuristic is most commonly a problem-specific local search technique.
- The starting point for the search may be a randomly constructed candidate solution, or constructed using a problem-specific heuristic (such as nearest neighbor).
- Perturbations can be made deterministically, although stochastic and probabilistic (adaptive based on history) are the most common.
- The procedure may store as much or as little history as needed to be used during perturbation and acceptance criteria. No history represents a random walk in a larger neighborhood of the best solution and is the most common implementation of the approach.
- The simplest and most common acceptance criteria is an improvement in the cost of constructed candidate solutions.

2.5.5 Code Listing

Listing 2.4 provides an example of the Iterated Local Search algorithm implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units.

The Iterated Local Search runs for a fixed number of iterations. The implementation is based on a common algorithm configuration for the TSP, where a 'double-bridge move' (4-opt) is used as the perturbation technique, and a stochastic 2-opt is used as the embedded Local Search heuristic. The double-bridge move involves partitioning a permutation into 4 pieces (a,b,c,d) and putting it back together in a specific and jumbled ordering (a,d,c,b).

```
1  def euc_2d(c1, c2)
2    Math::sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
3  end
4
5  def cost(permutation, cities)
6    distance =0
7    permutation.each_with_index do |c1, i|
8      c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
9      distance += euc_2d(cities[c1], cities[c2])
10   end
11   return distance
12 end
```

```
13
14 def random_permutation(cities)
15   all = Array.new(cities.length) {|i| i}
16   return Array.new(all.length) {|i| all.delete_at(rand(all.length))}
17 end
18
19 def stochastic_two_opt(permutation)
20   perm = Array.new(permutation)
21   c1, c2 = rand(perm.length), rand(perm.length)
22   c2 = rand(perm.length) while c1 == c2
23   c1, c2 = c2, c1 if c2 < c1
24   perm[c1...c2] = perm[c1...c2].reverse
25   return perm
26 end
27
28 def local_search(best, cities, max_no_improvements)
29   count = 0
30   begin
31     candidate = {}
32     candidate[:vector] = stochastic_two_opt(best[:vector])
33     candidate[:cost] = cost(candidate[:vector], cities)
34     if candidate[:cost] < best[:cost]
35       count, best = 0, candidate
36     else
37       count += 1
38     end
39   end until count >= max_no_improvements
40   return best
41 end
42
43 def double_bridge_move(perm)
44   pos1 = 1 + rand(perm.length / 4)
45   pos2 = pos1 + 1 + rand(perm.length / 4)
46   pos3 = pos2 + 1 + rand(perm.length / 4)
47   return perm[0...pos1] + perm[pos3..perm.length] + perm[pos2...pos3] +
         perm[pos1...pos2]
48 end
49
50 def perturbation(cities, best)
51   candidate = {}
52   candidate[:vector] = double_bridge_move(best[:vector])
53   candidate[:cost] = cost(candidate[:vector], cities)
54   return candidate
55 end
56
57 def search(cities, max_iterations, max_no_improvements)
58   best = {}
59   best[:vector] = random_permutation(cities)
60   best[:cost] = cost(best[:vector], cities)
61   best = local_search(best, cities, max_no_improvements)
62   max_iterations.times do |iter|
63     candidate = perturbation(cities, best)
64     candidate = local_search(candidate, cities, max_no_improvements)
```

```
    best = candidate if candidate[:cost] < best[:cost]
    puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
  end
  return best
end

max_iterations = 100
max_no_improvements = 50
berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],[880,660],[25,230],
 [525,1000],[580,1175],[650,1130],[1605,620],[1220,580],[1465,200],[1530,5],
 [845,680],[725,370],[145,665],[415,635],[510,875],[560,365],[300,465],
 [520,585],[480,415],[835,625],[975,580],[1215,245],[1320,315],[1250,400],
 [660,180],[410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
 [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],[95,260],
 [875,920],[700,500],[555,815],[830,485],[1170,65],[830,610],[605,625],
 [595,360],[1340,725],[1740,245]]

best = search(berlin52, max_iterations, max_no_improvements)
puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.4: Iterated Local Search algorithm in the Ruby Programming Language

2.5.6 References

Primary Sources

The definition and framework for Iterated Local Search was described by Stützle in his PhD dissertation [221]. Specifically he proposed constrains on what constitutes an Iterated Local Search algorithm as i) a single chain of candidate solutions, and ii) the method used to improve candidate solutions occurs within a reduced space by a black-box heuristic. Stützle does not take credit for the approach, instead highlighting specific instances of Iterated Local Search from the literature, such as 'iterated descent' [23], 'large-step Markov chains' [157], 'iterated Lin-Kernighan' [125], 'chained local optimization' [156], as well as [24] that introduces the principle, and [126] that summarized it (list taken from [187])

Learn More

Two early technical reports by Stützle that present applications of Iterated Local Search include a report on the Quadratic Assignment Problem [219], and another on the permutation flow shop problem [218]. Stützle and Hoos also published an early paper studying Iterated Local Search for to the TSP [220]. Lourenco, Martin, and Stützle provide a concise presentation of the technique, related techniques and the framework, much as it is presented in Stützle's dissertation [151]. The same author's also preset an authoritative summary of the approach and its applications as a book chapter [187].

2.6 Guided Local Search

Guided Local Search, GLS.

2.6.1 Taxonomy

The Guided Local Search algorithm is a Metaheuristic and a Global Optimization algorithm that makes use of an embedded Local Search algorithm. It is an extension to Local Search algorithms such as Hill Climbing (Section 2.4) and is similar in strategy to the Tabu Search algorithm (Section 2.10) and the Iterated Local Search algorithm (Section 2.5).

2.6.2 Strategy

The strategy for the Guided Local Search algorithm is to use penalties to encourage a Local Search technique to escape local optima and discover the global optima. A Local Search algorithm is run until it gets stuck in a local optima. The features from the local optima are evaluated and penalized, the results of which are used in an augmented cost function employed by the Local Search procedure. The Local Search is repeated a number of times using the last local optima discovered and the augmented cost function that guides exploration away from solutions with features present in discovered local optima.

2.6.3 Procedure

Algorithm 5 provides a pseudo-code listing of the Guided Local Search algorithm for minimization. The Local Search algorithm used by the Guided Local Search algorithm uses an augmented cost function in the form $h(s) = g(s) + \lambda \cdot \sum_{i=1}^{M} f_i$, where $h(s)$ is the augmented cost function, $g(s)$ is the problem cost function,λ is the 'regularization parameter' (a coefficient for scaling the penalties), s is a locally optimal solution of M features, and f_i is the i'th feature in locally optimal solution. The augmented cost function is only used by the local search procedure, the Guided Local Search algorithm uses the problem specific cost function without augmentation.

Penalties are only updated for those features in a locally optimal solution that maximize utility, updated by adding 1 to the penalty for the future (a counter). The utility for a feature is calculated as $U_{feature} = \frac{C_{feature}}{1+P_{feature}}$, where $U_{feature}$ is the utility for penalizing a feature (maximizing), $C_{feature}$ is the cost of the feature, and $P_{feature}$ is the current penalty for the feature.

2.6.4 Heuristics

- The Guided Local Search procedure is independent of the Local Search procedure embedded within it. A suitable domain-specific search procedure should be identified and employed.

Algorithm 5: Pseudo Code Listing for the Guided Local Search algorithm.

Input: $Iter_{max}$, λ
Output: S_{best}
$f_{penalties} \leftarrow 0$;
$S_{best} \leftarrow$ `RandomSolution()`;
foreach $Iter_i \in Iter_{max}$ **do**
  $S_{curr} \leftarrow$ `LocalSearch(`S_{best}, λ, $f_{penalties}$`)`;
  $f_{utilities} \leftarrow$ `CalculateFeatureUtilities(`S_{curr}, $f_{penalties}$`)`;
  $f_{penalties} \leftarrow$ `UpdateFeaturePenalties(`S_{curr}, $f_{penalties}$, $f_{utilities}$`)`;
  if `Cost(`S_{curr}`)` $\leq$ `Cost(`S_{best}`)` **then**
    $S_{best} \leftarrow S_{curr}$;
  end
end
return S_{best};

- The Guided Local Search procedure may need to be executed for thousands to hundreds-of-thousands of iterations, each iteration of which assumes a run of a Local Search algorithm to convergence.
- The algorithm was designed for discrete optimization problems where a solution is comprised of independently assessable 'features' such as Combinatorial Optimization, although it has been applied to continuous function optimization modeled as binary strings.
- The λ parameter is a scaling factor for feature penalization that must be in the same proportion to the candidate solution costs from the specific problem instance to which the algorithm is being applied. As such, the value for λ must be meaningful when used within the augmented cost function (such as when it is added to a candidate solution cost in minimization and subtracted from a cost in the case of a maximization problem).

2.6.5 Code Listing

Listing 2.5 provides an example of the Guided Local Search algorithm implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units.

The implementation of the algorithm for the TSP was based on the configuration specified by Voudouris in [226]. A TSP-specific local search algorithm is used called 2-opt that selects two points in a permutation and reconnects the tour, potentially untwisting the tour at the selected points. The stopping condition for 2-opt was configured to be a fixed number of non-improving moves.

The equation for setting λ for TSP instances is $\lambda = \alpha \cdot \frac{cost(optima)}{N}$, where N is the number of cities, $cost(optima)$ is the cost of a local optimum found by a local search, and $\alpha \in (0, 1]$ (around 0.3 for TSP and 2-opt). The cost of a local optima was fixed to the approximated value of 15000 for the Berlin52 instance. The utility function for features (edges) in the TSP is $U_{edge} = \frac{D_{edge}}{1+P_{edge}}$, where U_{edge} is the utility for penalizing an edge (maximizing), D_{edge} is the cost of the edge (distance between cities) and P_{edge} is the current penalty for the edge.

```
def euc_2d(c1, c2)
  Math::sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
end

def random_permutation(cities)
  perm = Array.new(cities.length){|i|i}
  for i in 0...perm.length
    r = rand(perm.length-i) + i
    perm[r], perm[i] = perm[i], perm[r]
  end
  return perm
end

def two_opt(permutation)
  perm = Array.new(permutation)
  c1, c2 = rand(perm.length), rand(perm.length)
  c2 = rand(perm.length) while c1 == c2
  c1, c2 = c2, c1 if c2 < c1
  perm[c1...c2] = perm[c1...c2].reverse
  return perm
end

def augmented_cost(permutation, penalties, cities, lambda)
  distance, augmented = 0, 0
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
    c1, c2 = c2, c1 if c2 < c1
    d = euc_2d(cities[c1], cities[c2])
    distance += d
    augmented += d + (lambda * (permutation[c1][c2]))
  end
  return distance, augmented
end

def local_search(current, cities, penalties, max_no_improvements, lambda)
  current[:cost], current[:acost] = augmented_cost(current[:vector], penalties, cities,
      lambda)
  count = 0
  begin
    perm = {}
    perm[:vector] = two_opt(current[:vector])
    perm[:cost], perm[:acost] = augmented_cost(perm[:vector], penalties, cities, lambda)
    if perm[:acost] < current[:acost]
      count, current = 0, perm
```

```
    else
      count += 1
    end
  end until count >= max_no_improvements
  return current
end

def calculate_feature_utilities(penalties, cities, permutation)
  utilities = Array.new(permutation.length,0)
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
    c1, c2 = c2, c1 if c2 < c1
    utilities[i] = euc_2d(cities[c1], cities[c2]) / (1.0 + penalties[c1][c2])
  end
  return utilities
end

def update_penalties!(penalties, cities, permutation, utilities)
  max = utilities.max()
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
    c1, c2 = c2, c1 if c2 < c1
    penalties[c1][c2] += 1 if utilities[i] == max
  end
  return penalties
end

def search(max_iterations, cities, max_no_improvements, lambda)
  best, current = nil, {}
  current[:vector] = random_permutation(cities)
  penalties = Array.new(cities.length){Array.new(cities.length,0)}
  max_iterations.times do |iter|
    current = local_search(current, cities, penalties, max_no_improvements, lambda)
    utilities = calculate_feature_utilities(penalties, cities, current[:vector])
    update_penalties!(penalties, cities, current[:vector], utilities)
    if(best.nil? or current[:cost] < best[:cost])
      best = current
    end
    puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
  end
  return best
end

max_iterations = 100
max_no_improvements = 15
alpha = 0.3
local_search_optima = 15000.0
berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],[880,660],[25,230],
 [525,1000],[580,1175],[650,1130],[1605,620],[1220,580],[1465,200],[1530,5],
 [845,680],[725,370],[145,665],[415,635],[510,875],[560,365],[300,465],
 [520,585],[480,415],[835,625],[975,580],[1215,245],[1320,315],[1250,400],
 [660,180],[410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
 [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],[95,260],
```

```
 [875,920],[700,500],[555,815],[830,485],[1170,65],[830,610],[605,625],
 [595,360],[1340,725],[1740,245]]

lambda = alpha * (local_search_optima/berlin52.length.to_f)
best = search(max_iterations, berlin52, max_no_improvements, lambda)
puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.5: Guided Local Search algorithm in the Ruby Programming Language

2.6.6 References

Primary Sources

Guided Local Search emerged from an approach called GENET, which is a connectionist approach to constraint satisfaction [233, 224]. Guided Local Search was presented by Voudouris and Tsang in a series of technical reports (that were later published) that described the technique and provided example applications of it to constraint satisfaction [229], combinatorial optimization [232, 231], and function optimization [230]. The seminal work on the technique was Voudouris' PhD dissertation [226].

Learn More

Voudouris and Tsang provide a high-level introduction to the technique [228], and a contemporary summary of the approach in Glover and Kochenberger's 'Handbook of metaheuristics' [227] that includes a review of the technique, application areas, and demonstration applications on a diverse set of problem instances. Mills, et al. elaborated on the approach, devising an 'Extended Guided Local Search' (EGLS) technique that added 'aspiration criteria' and random moves to the procedure [161], work which culminated in Mills' PhD dissertation [160]. Lau and Tsang further extended the approach by integrating it with a Genetic Algorithm, called the 'Guided Genetic Algorithm' (GGA) [149], that also culminated in a PhD dissertation by Lau [148].

2.7 Variable Neighborhood Search

Variable Neighborhood Search, VNS.

2.7.1 Taxonomy

Variable Neighborhood Search is a Metaheuristic and a Global Optimization technique that manages a Local Search technique. It is related to the Iterative Local Search algorithm (Section 2.5).

2.7.2 Strategy

The strategy for the Variable Neighborhood Search involves iterative exploration of larger and larger neighborhoods for a given local optima until an improvement is located after which time the search across expanding neighborhoods is repeated. The strategy is motivated by three principles: i) a local minimum for one neighborhood structure may not be a local minimum for a different neighborhood structure, ii) a global minimum is a local minimum for all possible neighborhood structures, and iii) local minimum are relatively close to global minimum for many problem classes.

2.7.3 Procedure

Algorithm 6 provides a pseudo-code listing of the Variable Neighborhood Search algorithm for minimizing a cost function. The pseudo code shows that the systematic search of expanding neighborhoods for a local optimum is abandoned when an global improvement is achieved (shown with the `Break` jump).

2.7.4 Heuristics

- Approximation methods (such as stochastic hill climbing) are suggested for use as the Local Search procedure for large problem instances in order to reduce the running time.
- Variable Neighborhood Search has been applied to a very wide array of combinatorial optimization problems as well as clustering and continuous function optimization problems.
- The embedded Local Search technique should be specialized to the problem type and instance to which the technique is being applied.
- The Variable Neighborhood Descent (VND) can be embedded in the Variable Neighborhood Search as a the Local Search procedure and has been shown to be most effective.

Algorithm 6: Pseudo Code Listing for the Variable Neighborhood Search algorithm.

Input: Neighborhoods
Output: S_{best}
$S_{best} \leftarrow$ `RandomSolution()`;
while $\neg$ `StopCondition()` **do**
  foreach $Neighborhood_i \in$ Neighborhoods **do**
    $Neighborhood_{curr} \leftarrow$ `CalculateNeighborhood(`S_{best}, $Neighborhood_i$`)`;
    $S_{candidate} \leftarrow$ `RandomSolutionInNeighborhood(`$Neighborhood_{curr}$`)`;
    $S_{candidate} \leftarrow$ `LocalSearch(`$S_{candidate}$`)`;
    if `Cost(`$S_{candidate}$`)` $<$ `Cost(`S_{best}`)` **then**
      $S_{best} \leftarrow S_{candidate}$;
      `Break`;
    end
  end
end
return S_{best};

2.7.5 Code Listing

Listing 2.6 provides an example of the Variable Neighborhood Search algorithm implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units.

The Variable Neighborhood Search uses a stochastic 2-opt procedure as the embedded local search. The procedure deletes two edges and reverses the sequence in-between the deleted edges, potentially removing 'twists' in the tour. The neighborhood structure used in the search is the number of times the 2-opt procedure is performed on a permutation, between 1 and 20 times. The stopping condition for the local search procedure is a maximum number of iterations without improvement. The same stop condition is employed by the higher-order Variable Neighborhood Search procedure, although with a lower boundary on the number of non-improving iterations.

```
1  def euc_2d(c1, c2)
2    Math::sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
3  end
4
5  def random_permutation(cities)
6    perm = Array.new(cities.length){|i|i}
7    for i in 0...perm.length
8      r = rand(perm.length-i) + i
9      perm[r], perm[i] = perm[i], perm[r]
10   end
11   return perm
```

```
12 end
13
14 def stochastic_two_opt!(perm)
15   c1, c2 = rand(perm.length), rand(perm.length)
16   c2 = rand(perm.length) while c1 == c2
17   c1, c2 = c2, c1 if c2 < c1
18   perm[c1...c2] = perm[c1...c2].reverse
19   return perm
20 end
21
22 def cost(permutation, cities)
23   distance =0
24   permutation.each_with_index do |c1, i|
25     c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
26     distance += euc_2d(cities[c1], cities[c2])
27   end
28   return distance
29 end
30
31 def local_search(best, cities, max_no_improvements, neighborhood)
32   count = 0
33   begin
34     candidate = {}
35     candidate[:vector] = Array.new(best[:vector])
36     neighborhood.times{stochastic_two_opt!(candidate[:vector])}
37     candidate[:cost] = cost(candidate[:vector], cities)
38     if candidate[:cost] < best[:cost]
39       count, best = 0, candidate
40     else
41       count += 1
42     end
43   end until count >= max_no_improvements
44   return best
45 end
46
47 def search(cities, neighborhoods, max_no_improvements, max_no_improvements_ls)
48   best = {}
49   best[:vector] = random_permutation(cities)
50   best[:cost] = cost(best[:vector], cities)
51   iter, count = 0, 0
52   begin
53     neighborhoods.each do |neighborhood|
54       candidate = {}
55       candidate[:vector] = Array.new(best[:vector])
56       neighborhood.times{stochastic_two_opt!(candidate[:vector])}
57       candidate[:cost] = cost(candidate[:vector], cities)
58       candidate = local_search(candidate, cities, max_no_improvements_ls, neighborhood)
59       puts " > iteration #{(iter+1)}, neighborhood=#{neighborhood}, best=#{best[:cost]}"
60       iter += 1
61       if(candidate[:cost] < best[:cost])
62         best = candidate
63         count = 0
64         puts "New best, restarting neighborhood search."
```

```
      break
    else
      count += 1
    end
  end
 end until count >= max_no_improvements
 return best
end

max_no_mprovements = 50
local_search_no_improvements = 70
neighborhoods = 1...20
berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],[880,660],[25,230],
 [525,1000],[580,1175],[650,1130],[1605,620],[1220,580],[1465,200],[1530,5],
 [845,680],[725,370],[145,665],[415,635],[510,875],[560,365],[300,465],
 [520,585],[480,415],[835,625],[975,580],[1215,245],[1320,315],[1250,400],
 [660,180],[410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
 [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],[95,260],
 [875,920],[700,500],[555,815],[830,485],[1170,65],[830,610],[605,625],
 [595,360],[1340,725],[1740,245]]

best = search(berlin52, neighborhoods, max_no_mprovements, local_search_no_improvements)
puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.6: Variable Neighborhood Search algorithm in the Ruby Programming Language

2.7.6 References

2.7.7 Primary Sources

The seminal paper for describing Variable Neighborhood Search was by Mladenovic and Hansen in 1997 [166], although an early abstract by Mladenovic is sometimes cited [165]. The approach is explained in terms of three different variations on the general theme. Variable Neighborhood Descent (VND) refers to the use of a Local Search procedure and the deterministic (as opposed to stochastic or probabilistic) change of neighborhood size. Reduced Variable Neighborhood Search (RVNS) involves performing a stochastic random search within a neighborhood and no refinement via a local search technique. Basic Variable Neighborhood Search is the canonical approach described by Mladenovic and Hansen in the seminal paper.

2.7.8 Learn More

There are a large number of papers published on Variable Neighborhood Search, its applications and variations. Hansen and Mladenovic provide an overview of the approach that includes its recent history, extensions and a detailed review of the numerous areas of application [109]. For some additional useful overviews of the technique, its principles, and applications, see [107, 110, 108].

There are many extensions to Variable Neighborhood Search. Some popular examples include: Variable Neighborhood Decomposition Search (VNDS) that involves embedding a second heuristic or metaheuristic approach in VNS to replace the Local Search procedure [111], Skewed Variable Neighborhood Search (SVNS) that encourages exploration of neighborhoods far away from discovered local optima, and Parallel Variable Neighborhood Search (PVNS) that either parallelizes the local search of a neighborhood or parallelizes the searching of the neighborhoods themselves.

2.8 Greedy Randomized Adaptive Search

Greedy Randomized Adaptive Search Procedure, GRASP.

2.8.1 Taxonomy

The Greedy Randomized Adaptive Search Procedure is a Metaheuristic and Global Optimization algorithm, originally proposed for the Operations Research practitioners. The iterative application of an embedded Local Search technique relate the approach to Iterative Local Search (Section 2.5) and Multi-Start techniques.

2.8.2 Strategy

The objective of the Greedy Randomized Adaptive Search Procedure is to repeatedly sample stochastically greedy solutions, and then use a local search procedure to refine them to a local optima. The strategy of the procedure is centered on the stochastic and greedy step-wise construction mechanism that constrains the selection and order-of-inclusion of the components of a solution based on the value they are expected to provide.

2.8.3 Procedure

Algorithm 7 provides a pseudo-code listing of the Greedy Randomized Adaptive Search Procedure for minimizing a cost function.

Algorithm 7: Pseudo Code for the Greedy Randomized Adaptive Search Procedure.

```
   Input: α
   Output: S_best
1  S_best ← ConstructRandomSolution();
2  while ¬ StopCondition() do
3  |  S_candidate ← GreedyRandomizedConstruction(α);
4  |  S_candidate ← LocalSearch(S_candidate);
5  |  if Cost(S_candidate) < Cost(S_best) then
6  |  |  S_best ← S_candidate;
7  |  end
8  end
9  return S_best;
```

Algorithm 8 provides the pseudo-code the Greedy Randomized Construction function. The function involves the step-wise construction of a candidate solution using a stochastically greedy construction process. The functions works by building a Restricted Candidate List (RCL) that constraints the components of a solution (features) that may be selected from each cycle. The RCL may be constrained by an explicit size, or by

using a threshold ($\alpha \in [0, 1]$) on the cost of adding each feature to the current candidate solution.

Algorithm 8: Pseudo Code Listing for the Greedy Randomized Construction function.

```
   Input: α
   Output: S_candidate
 1 S_candidate ← 0;
 2 while S_candidate ≠ ProblemSize do
 3 |   Feature_costs ← 0;
 4 |   for Feature_i ∉ S_candidate do
 5 |   |   Feature_costs ← CostOfAddingFeatureToSolution(S_candidate, Feature_i);
 6 |   end
 7 |   RCL ← 0;
 8 |   Fcost_min ← MinCost(Feature_costs);
 9 |   Fcost_max ← MaxCost(Feature_costs);
10 |   for F_icost ∈ Feature_costs do
11 |   |   if F_icost ≤ Fcost_min + α · (Fcost_max − Fcost_min) then
12 |   |   |   RCL ← Feature_i;
13 |   |   end
14 |   end
15 |   S_candidate ← SelectRandomFeature(RCL);
16 end
17 return S_candidate;
```

2.8.4 Heuristics

- The α threshold defines the amount of greediness of the construction mechanism, where values close to 0 may be too greedy, and values close to 1 may be too generalized.
- As an alternative to using the α threshold, the RCL can be constrained to the top $n\%$ of candidate features that may be selected from each construction cycle.
- The technique was designed for discrete problem classes such as combinatorial optimization problems.

2.8.5 Code Listing

Listing 2.7 provides an example of the Greedy Randomized Adaptive Search Procedure implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units.

The stochastic and greedy step-wise construction of a tour involves evaluating candidate cities by the the cost they contribute as being the next city in the tour. The algorithm uses a stochastic 2-opt procedure for the Local Search with a fixed number of non-improving iterations as the stopping condition.

```
def euc_2d(c1, c2)
  Math::sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
end

def cost(permutation, cities)
  distance =0
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
    distance += euc_2d(cities[c1], cities[c2])
  end
  return distance
end

def stochastic_two_opt(permutation)
  perm = Array.new(permutation)
  c1, c2 = rand(perm.length), rand(perm.length)
  c2 = rand(perm.length) while c1 == c2
  c1, c2 = c2, c1 if c2 < c1
  perm[c1...c2] = perm[c1...c2].reverse
  return perm
end

def local_search(best, cities, max_no_improvements)
  count = 0
  begin
    candidate = {}
    candidate[:vector] = stochastic_two_opt(best[:vector])
    candidate[:cost] = cost(candidate[:vector], cities)
    if candidate[:cost] < best[:cost]
      count, best = 0, candidate
    else
      count += 1
    end
  end until count >= max_no_improvements
  return best
end

def construct_randomized_greedy_solution(cities, alpha)
  candidate = {}
  candidate[:vector] = [rand(cities.length)]
  allCities = Array.new(cities.length) {|i| i}
  while candidate[:vector].length < cities.length
    candidates = allCities - candidate[:vector]
    costs = Array.new(candidates.length){|i| euc_2d(cities[candidate[:vector].last],
        cities[i])}
    rcl, max, min = [], costs.max, costs.min
    costs.each_with_index { |c,i| rcl<<candidates[i] if c <= (min + alpha*(max-min)) }
    candidate[:vector] << rcl[rand(rcl.length)]
```

```
  end
  candidate[:cost] = cost(candidate[:vector], cities)
  return candidate
end

def search(cities, max_iterations, max_no_improvements, alpha)
  best = nil
  max_iterations.times do |iter|
    candidate = construct_randomized_greedy_solution(cities, alpha);
    candidate = local_search(candidate, cities, max_no_improvements)
    best = candidate if best.nil? or candidate[:cost] < best[:cost]
    puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
  end
  return best
end

max_iterations = 50
max_no_improvements = 100
greediness_factor = 0.3
berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],[880,660],[25,230],
 [525,1000],[580,1175],[650,1130],[1605,620],[1220,580],[1465,200],[1530,5],
 [845,680],[725,370],[145,665],[415,635],[510,875],[560,365],[300,465],
 [520,585],[480,415],[835,625],[975,580],[1215,245],[1320,315],[1250,400],
 [660,180],[410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
 [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],[95,260],
 [875,920],[700,500],[555,815],[830,485],[1170,65],[830,610],[605,625],
 [595,360],[1340,725],[1740,245]]

best = search(berlin52, max_iterations, max_no_improvements, greediness_factor)
puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.7: Greedy Randomized Adaptive Search Procedure algorithm in the Ruby Programming Language

2.8.6 References

Primary Sources

The seminal paper that introduces the general approach of stochastic and greedy stepwise construction of candidate solutions is by Feo and Resende [60]. The general approach was inspired by greedy heuristics by Hart and Shogan [112]. The seminal review paper that is cited with the preliminary paper is by Feo and Resende [58], and provides a coherent description of the GRASP technique, an example, and review of early applications. An early application was by Feo, Venkatraman and Bard for a machine scheduling problem [62]. Other early applications to scheduling problems include technical reports [59] (later published as [12]) and [61] (also later published as [63]).

Learn More

There are a vast number of review, application, and extension papers for GRASP. Pitsoulis and Resende provide an extensive contemporary overview of the field as a review chapter [180], as does Resende and Ribeiro that includes a clear presentation of the use of the α threshold parameter instead of a fixed size for the RCL [191]. Festa and Resende provide an annotated bibliography as a review chapter that provides some needed insight into large amount of study that has gone into the approach [68]. There are numerous extensions to GRASP, not limited to the popular Reactive GRASP for adapting α [183], the use of long term memory to allow the technique to learn from candidate solutions discovered in previous iterations, and parallel implementations of the procedure such as 'Parallel GRASP' [175].

2.9 Scatter Search

Scatter Search, SS.

2.9.1 Taxonomy

Scatter search is a Metaheuristic and a Global Optimization algorithm. It is also sometimes associated with the field of Evolutionary Computation given the use of a population and recombination in the structure of the technique. Scatter Search is a sibling of Tabu Search (Section 2.10), developed by the same author and based on similar origins.

2.9.2 Strategy

The objective of Scatter Search is to maintain a set of diverse and high-quality candidate solutions. The principle of the approach is that useful information about the global optima is stored in a diverse and elite set of solutions (the reference set) and that recombining samples from the set can exploit this information. The strategy involves an iterative process, where a population of diverse and high-quality candidate solutions that are partitioned into subsets and linearly recombined to create weighted centroids of sample-based neighborhoods. The results of recombination are refined using an embedded heuristic and assessed in the context of the reference set as to whether or not they are retained.

2.9.3 Procedure

Algorithm 9 provides a pseudo-code listing of the Scatter Search algorithm for minimizing a cost function. The procedure is based on the abstract form presented by Glover as a template for the general class of technique [94], with influences from an application of the technique to function optimization by Glover [94].

2.9.4 Heuristics

- Scatter search is suitable for both discrete domains such as combinatorial optimization as well as continuous domains such as non-linear programming (continuous function optimization).
- Small set sizes are preferred for the `ReferenceSet`, such as 10 or 20 members.
- Subset sizes can be 2,3,4 or more members that are all recombined to produce viable candidate solutions within the neighborhood of the members of the subset.
- Each subset should comprise at least one member added to the set in the previous algorithm iteration.
- The Local Search procedure should be a problem-specific improvement heuristic.

Algorithm 9: Pseudo Code for the Scatter Search algorithm.

Input: $DiverseSet_{size}$, $ReferenceSet_{size}$
Output: ReferenceSet

```
InitialSet ← ConstructInitialSolution(DiverseSet_size);
RefinedSet ← 0;
for S_i ∈ InitialSet do
    RefinedSet ← LocalSearch(S_i);
end
ReferenceSet ← SelectInitialReferenceSet(ReferenceSet_size);
while ¬ StopCondition() do
    Subsets ← SelectSubset(ReferenceSet);
    CandidateSet ← 0;
    for Subset_i ∈ Subsets do
        RecombinedCandidates ← RecombineMembers(Subset_i);
        for S_i ∈ RecombinedCandidates do
            CandidateSet ← LocalSearch(S_i);
        end
    end
    ReferenceSet ← Select(ReferenceSet, CandidateSet, ReferenceSet_size);
end
return ReferenceSet;
```

- The selection of members for the `ReferenceSet` at the end of each iteration favors solutions with higher quality and may also promote diversity.
- The `ReferenceSet` may be updated at the end of an iteration, or dynamically as candidates are created (a so-called steady-state population in some evolutionary computation literature).
- A lack of changes to the `ReferenceSet` may be used as a signal to stop the current search, and potentially restart the search with a newly initialized `ReferenceSet`.

2.9.5 Code Listing

Listing 2.8 provides an example of the Scatter Search algorithm implemented in the Ruby Programming Language. The example problem is an instance of a continuous function optimization that seeks $minf(x)$ where $f = \sum_{i=1}^{n} x_i^2$, $-5.0 \leq x_i \leq 5.0$ and $n = 3$. The optimal solution for this basin function is $(v_1, \ldots, v_n) = 0.0$.

The algorithm is an implementation of Scatter Search as described in an application of the technique to unconstrained non-linear optimization by Glover [97]. The seeds for initial solutions are generated as random vectors, as opposed to stratified samples. The example was further simplified by not including a restart strategy, and the exclusion of diversity maintenance in the `ReferenceSet`. A stochastic local search algorithm is used

as the embedded heuristic that uses a stochastic step size in the range of half a percent of the search space.

```
def cost(candidate_vector)
  return candidate_vector.inject(0) {|sum, x| sum + (x ** 2.0)}
end

def random_solution(problem_size, search_space)
  return Array.new(problem_size) do |i|
    search_space[i][0] + ((search_space[i][1] - search_space[i][0]) * rand())
  end
end

def take_step(current, search_space, step_size)
  step = []
  current.length.times do |i|
    max, min = current[i]+step_size, current[i]-step_size
    max = search_space[i][1] if max > search_space[i][1]
    min = search_space[i][0] if min < search_space[i][0]
    step << min + ((max - min) * rand)
  end
  return step
end

def local_search(best, search_space, max_no_improvements, step_size)
  count = 0
  begin
    candidate = {}
    candidate[:vector] = take_step(best[:vector], search_space, step_size)
    candidate[:cost] = cost(candidate[:vector])
    if candidate[:cost] < best[:cost]
      count, best = 0, candidate
    else
      count += 1
    end
  end until count >= max_no_improvements
  return best
end

def construct_initial_set(problem_size, search_space, div_set_size,
    max_no_improvements, step_size)
  diverse_set = []
  begin
    candidate = {}
    candidate[:vector] = random_solution(problem_size, search_space)
    candidate[:cost] = cost(candidate[:vector])
    candidate = local_search(candidate, search_space, max_no_improvements, step_size)
    diverse_set << candidate if !diverse_set.any? {|x| x[:vector]==candidate[:vector]}
  end until diverse_set.length == div_set_size
  return diverse_set
end

def euclidean(v1, v2)
  sum = 0.0
```

```
51    v1.each_with_index {|v, i| sum += (v**2.0 - v2[i]**2.0) }
52    sum = sum + (0.0-sum) if sum < 0.0
53    return Math.sqrt(sum)
54  end
55
56  def distance(vector1, reference_set)
57    sum = 0.0
58    reference_set.each do |s|
59      sum += euclidean(vector1, s[:vector])
60    end
61    return sum
62  end
63
64  def diversify(diverse_set, numElite, ref_set_size)
65    diverse_set.sort!{|x,y| x[:cost] <=> y[:cost]}
66    reference_set = Array.new(numElite){|i| diverse_set[i]}
67    remainder = diverse_set - reference_set
68    remainder.sort!{|x,y| distance(y[:vector], reference_set) <=> distance(x[:vector],
          reference_set)}
69    reference_set = reference_set + remainder[0..(ref_set_size-reference_set.length)]
70    return reference_set, reference_set[0]
71  end
72
73  def select_subsets(reference_set)
74    additions = reference_set.select{|c| c[:new]}
75    remainder = reference_set - additions
76    remainder = additions if remainder.empty?
77    subsets = []
78    additions.each{|a| remainder.each{|r| subsets << [a,r] if a!=r}}
79    return subsets
80  end
81
82  def recombine(subset, problem_size, search_space)
83    a, b = subset
84    d = rand(euclidean(a[:vector], b[:vector]))/2.0
85    children = []
86    subset.each do |p|
87      step = (rand<0.5) ? +d : -d
88      child = {}
89      child[:vector] = Array.new(problem_size){|i| p[:vector][i]+step}
90      child[:vector].each_with_index {|m,i| child[:vector][i]=search_space[i][0] if
            m<search_space[i][0]}
91      child[:vector].each_with_index {|m,i| child[:vector][i]=search_space[i][1] if
            m>search_space[i][1]}
92      child[:cost] = cost(child[:vector])
93      children << child
94    end
95    return children
96  end
97
98  def search(problem_size, search_space, max_iterations, ref_set_size, div_set_size,
        max_no_improvements, step_size, max_elite)
99    diverse_set = construct_initial_set(problem_size, search_space, div_set_size,
```

```
      max_no_improvements, step_size)
  reference_set, best = diversify(diverse_set, max_elite, ref_set_size)
  reference_set.each{|c| c[:new] = true}
  max_iterations.times do |iter|
    wasChange = false
    subsets = select_subsets(reference_set)
    reference_set.each{|c| c[:new] = false}
    subsets.each do |subset|
      candidates = recombine(subset, problem_size, search_space)
      improved = Array.new(candidates.length) {|i| local_search(candidates[i],
          search_space, max_no_improvements, step_size)}
      improved.each do |c|
        if !reference_set.any? {|x| x[:vector]==c[:vector]}
          c[:new] = true
          reference_set.sort!{|x,y| x[:cost] <=> y[:cost]}
          if c[:cost]<reference_set.last[:cost]
            reference_set.delete(reference_set.last)
            reference_set << c
            wasChange = true
          end
        end
      end
    end
    reference_set.sort!{|x,y| x[:cost] <=> y[:cost]}
    best = reference_set[0] if reference_set[0][:cost] < best[:cost]
    puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
    break if !wasChange
  end
  return best
end

num_iterations = 100
problem_size = 3
search_space = Array.new(problem_size) {|i| [-5, +5]}
step_size = (search_space[0][1]-search_space[0][0])*0.005
max_no_improvements = 30
ref_set_size = 10
diverse_set_size = 20
no_elite = 5

best = search(problem_size, search_space, num_iterations, ref_set_size,
    diverse_set_size, max_no_improvements, step_size, no_elite)
puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.8: Scatter Search algorithm in the Ruby Programming Language

2.9.6 References

2.9.7 Primary Sources

A form of the Scatter Search algorithm was proposed by Glover for integer programming [88], based on Glover's earlier work on surrogate constraints. The approach remained

idle until it was revisited by Glover and combined with Tabu Search [93]. The modern canonical reference of the approach was proposed by Glover who provides a abstract template of the procedure that may be specialized for a given application domain [94].

2.9.8 Learn More

The primary reference for the approach is the book by Laguna and Martí that reviews the principles of the approach in detail and presents tutorials on applications of the approach on standard problems using the C programming language [146]. There are many review articles and chapters on Scatter Search that may be used to supplement an understanding of the approach, such as a detailed review chapter by Glover [95], a review of the fundamentals of the approach and its relationship to an abstraction called 'path linking' by Glover, Laguna, and Martí [96], and a modern overview of the technique by Martí, Lagunab, and Glover [155].

2.10 Tabu Search

Tabu Search, TS, Taboo Search.

2.10.1 Taxonomy

Tabu Search is a Global Optimization algorithm and a Metaheuristic or Meta-strategy for controlling an embedded heuristic technique. Tabu Search is a parent for a large family of derivative approaches that introduce memory structures in Metaheuristics, such as Reactive Tabu Search (Section 2.11) and Parallel Tabu Search.

2.10.2 Strategy

The objective for the Tabu Search algorithm is to constrain an embedded heuristic from returning to recently visited areas of the search space, referred to as cycling. The strategy of the approach is to maintain a short term memory of the specific changes of recent moves within the search space and preventing future moves from undoing those changes. Additional intermediate-term memory structures may be introduced to bias moves toward promising areas of the search space, as well as longer-term memory structures that promote a general diversity in the search across the search space.

2.10.3 Procedure

Algorithm 10 provides a pseudo-code listing of the Tabu Search algorithm for minimizing a cost function. The listing shows the simple Tabu Search algorithm with short term memory, without intermediate and long term memory management.

2.10.4 Heuristics

- Tabu search was designed to manage an embedded hill climbing heuristic, although may be adapted to manage any neighborhood exploration heuristic.
- Tabu search was designed for, and has predominately been applied to discrete domains such as combinatorial optimization problems.
- Candidates for neighboring moves can be generated deterministically for the entire neighborhood or the neighborhood can be stochastically sampled to a fixed size, trading off efficiency for accuracy.
- Intermediate-term memory structures can be introduced (complementing the short-term memory) to focus the search on promising areas of the search space (intensification), called aspiration criteria.
- Long-term memory structures can be introduced (complementing the short-term memory) to encourage useful exploration of the broader search space, called diversification. Strategies may include generating solutions with rarely used components and biasing generation away from the most commonly used solution components.

Algorithm 10: Pseudo Code for the Tabu Search algorithm.

Input: $TabuList_{size}$
Output: S_{best}
$S_{best} \leftarrow$ `ConstructInitialSolution()`;
TabuList $\leftarrow 0$;
while $\neg$ `StopCondition()` **do**
  CandidateList $\leftarrow 0$;
  for $S_{candidate} \in Sbest_{neighborhood}$ **do**
    if $\neg$ `ContainsAnyFeatures(`$S_{candidate}$, TabuList`)` **then**
      CandidateList $\leftarrow S_{candidate}$;
    end
  end
  $S_{candidate} \leftarrow$ `LocateBestCandidate(CandidateList)`;
  if `Cost(`$S_{candidate}$`)` $\leq$ `Cost(`S_{best}`)` **then**
    $S_{best} \leftarrow S_{candidate}$;
    TabuList $\leftarrow$ `FeatureDifferences(`$S_{candidate}$, S_{best}`)`;
    while TabuList $> TabuList_{size}$ **do**
      `DeleteFeature(TabuList)`;
    end
  end
end
return S_{best};

2.10.5 Code Listing

Listing 2.9 provides an example of the Tabu Search algorithm implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units.

The algorithm is an implementation of the simple Tabu Search with a short term memory structure that executes for a fixed number of iterations. The starting point for the search is prepared using a random permutation that is refined using a stochastic 2-opt Local Search procedure. The stochastic 2-opt procedure is used as the embedded hill climbing heuristic with a fixed sized candidate list. The two edges that are deleted in each 2-opt move are stored on the tabu list. This general approach is similar to that used by Knox in his work on Tabu Search for symmetrical TSP [135] and Fiechter for the Parallel Tabu Search for the TSP [69].

```
1 def euc_2d(c1, c2)
2   Math::sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
3 end
4
5 def cost(permutation, cities)
```

```
  distance =0
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
    distance += euc_2d(cities[c1], cities[c2])
  end
  return distance
end

def random_permutation(cities)
  all = Array.new(cities.length) {|i| i}
  return Array.new(all.length) {|i| all.delete_at(rand(all.length))}
end

def stochastic_two_opt(permutation)
  perm = Array.new(permutation)
  c1, c2 = rand(perm.length), rand(perm.length)
  c2 = rand(perm.length) while c1 == c2
  c1, c2 = c2, c1 if c2 < c1
  perm[c1...c2] = perm[c1...c2].reverse
  return perm, [[permutation[c1-1], permutation[c1]], [permutation[c2-1],
      permutation[c2]]]
end

def generate_initial_solution(cities, max_no_improvements)
  best = {}
  best[:vector] = random_permutation(cities)
  best[:cost] = cost(best[:vector], cities)
  count = 0
  begin
    candidate = {}
    candidate[:vector] = stochastic_two_opt(best[:vector])[0]
    candidate[:cost] = cost(candidate[:vector], cities)
    if candidate[:cost] <= best[:cost]
      count, best = 0, candidate
    else
      count += 1
    end
  end until count >= max_no_improvements
  return best
end

def is_tabu?(permutation, tabu_list)
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
    tabu_list.each do |forbidden_edge|
      return true if forbidden_edge == [c1, c2]
    end
  end
  return false
end

def generate_candidate(best, tabu_list, cities)
  permutation, edges = nil, nil
```

```
58    begin
59      permutation, edges = stochastic_two_opt(best[:vector])
60    end while is_tabu?(permutation, tabu_list)
61    candidate = {}
62    candidate[:vector] = permutation
63    candidate[:cost] = cost(candidate[:vector], cities)
64    return candidate, edges
65  end
66
67  def search(cities, tabu_list_size, candidate_list_size, max_iterations,
        max_no_improvements)
68    best = generate_initial_solution(cities, max_no_improvements)
69    current = best
70    tabu_list = Array.new(tabu_list_size)
71    max_iterations.times do |iter|
72      candidates = Array.new(candidate_list_size) {|i| generate_candidate(current,
          tabu_list, cities)}
73      candidates.sort! {|x,y| x.first[:cost] <=> y.first[:cost]}
74      best_candidate = candidates.first[0]
75      best_candidate_edges = candidates.first[1]
76      if best_candidate[:cost] < current[:cost]
77        current = best_candidate
78        best = best_candidate if best_candidate[:cost] < best[:cost]
79        best_candidate_edges.each {|edge| tabu_list.push(edge)}
80        tabu_list.pop while tabu_list.length > tabu_list_size
81      end
82      puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
83    end
84    return best
85  end
86
87  max_iterations = 100
88  max_no_improvements = 50
89  tabu_list_size = 15
90  max_candidates = 50
91  berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],[880,660],[25,230],
92   [525,1000],[580,1175],[650,1130],[1605,620],[1220,580],[1465,200],[1530,5],
93   [845,680],[725,370],[145,665],[415,635],[510,875],[560,365],[300,465],
94   [520,585],[480,415],[835,625],[975,580],[1215,245],[1320,315],[1250,400],
95   [660,180],[410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
96   [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],[95,260],
97   [875,920],[700,500],[555,815],[830,485],[1170,65],[830,610],[605,625],
98   [595,360],[1340,725],[1740,245]]
99
100 best = search(berlin52, tabu_list_size, max_candidates, max_iterations,
        max_no_improvements)
101 puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.9: Tabu Search algorithm in the Ruby Programming Language

2.10.6 References

2.10.7 Primary Sources

Tabu Search was introduced by Glover applied to scheduling employees to duty rosters [98] and a more general overview in the context of the TSP [89], based on his previous work on surrogate constraints on integer programming problems [88]. Glover provided a seminal overview of the algorithm in a two-part journal article, the first part of which introduced the algorithm, and reviewed then-recent applications [90], and the second which focused on advanced topics and open areas of research [91].

2.10.8 Learn More

Glover provides a high-level introduction to Tabu Search in the form of a practical tutorial [92], as does Glover and Taillard in a user guide format [100]. The best source of information for Tabu Search is the book dedicated to the approach by Glover and Laguna that covers the principles of the technique in detail as well as an in-depth review of applications [101]. The approach appeared in Science, that considered a modification for its application to continuous function optimization problems [42]. Finally, Gendreau provides an excellent contemporary review of the algorithm, highlighting best practices and application heuristics collected from across the field of study [87].

2.11 Reactive Tabu Search

Reactive Tabu Search, RTS, R-TABU, Reactive Taboo Search.

2.11.1 Taxonomy

Reactive Tabu Search is a Metaheuristic and a Global Optimization algorithm. It is an extension of Tabu Search (Section 2.10) and the basis for a field of reactive techniques called Reactive Local Search and more broadly the field of Reactive Search Optimization.

2.11.2 Strategy

The objective of Tabu Search is to avoid cycles while applying a local search technique. The Reactive Tabu Search addresses this objective by explicitly monitoring the search and reacting to the occurrence of cycles and their repetition by adapting the tabu tenure (tabu list size). The strategy of the broader field of Reactive Search Optimization is to automate the process by which a practitioner configures a search procedure by monitoring its online behavior and to use machine learning techniques to adapt a techniques configuration.

2.11.3 Procedure

Algorithm 11 provides a pseudo-code listing of the Reactive Tabu Search algorithm for minimizing a cost function. The pseudo code is based on a the version of the Reactive Tabu Search described by Battiti and Tecchiolli in [20] with supplements like the `IsTabu` function from [18]. The procedure has been modified for brevity to exude the diversification procedure (escape move). Algorithm 12 describes the memory based reaction that manipulates the size of the `ProhibitionPeriod` in response to identified cycles in the ongoing search. Algorithm 13 describes the selection of the best move from a list of candidate moves in the neighborhood of a given solution. The function permits prohibited moves in the case where a prohibited move is better than the best know solution and the selected admissible move (called aspiration). Algorithm 14 determines whether a given neighborhood move is tabu based on the current `ProhibitionPeriod`, and is employed by sub-functions of the Algorithm 13 function.

2.11.4 Heuristics

- Reactive Tabu Search is an extension of Tabu Search and as such should exploit the best practices used for the parent algorithm.
- Reactive Tabu Search was designed for discrete domains such as combinatorial optimization, although has been applied to continuos function optimization.
- Reactive Tabu Search was proposed to use efficient memory data structures such as hash tables.

Algorithm 11: Pseudo Code for the Reactive Tabu Search algorithm.

Input: $Iteration_{max}$, Increase, Decrease, ProblemSize
Output: S_{best}

$S_{curr} \leftarrow$ `ConstructInitialSolution()`;
$S_{best} \leftarrow S_{curr}$;
TabuList $\leftarrow 0$;
ProhibitionPeriod $\leftarrow 1$;
foreach $Iteration_i \in Iteration_{max}$ **do**
  `MemoryBasedReaction`(Increase, Decrease, ProblemSize);
  CandidateList $\leftarrow$ `GenerateCandidateNeighborhood`(S_{curr});
  $S_{curr} \leftarrow$ `BestMove`(CandidateList);
  TabuList $\leftarrow Scurr_{feature}$;
  if `Cost`(S_{curr}) $\leq$ `Cost`(S_{best}) **then**
    $S_{best} \leftarrow S_{curr}$;
  end
end
return S_{best};

Algorithm 12: Pseudo Code for the `MemoryBasedReaction` function in the Reactive Tabu Search algorithm.

Input: Increase, Decrease, ProblemSize
Output:

if `HaveVisitedSolutionBefore`(S_{curr}, VisitedSolutions) **then**
  $Scurr_t \leftarrow$ `RetrieveLastTimeVisited`(VisitedSolutions, S_{curr});
  RepetitionInterval $\leftarrow Iteration_i - Scurr_t$;
  $Scurr_t \leftarrow Iteration_i$;
  if RepetitionInterval $< 2 *$ ProblemSize **then**
    $RepetitionInterval_{avg} \leftarrow$
    $0.1 *$ RepetitionInterval $+ 0.9 * RepetitionInterval_{avg}$;
    ProhibitionPeriod $\leftarrow$ ProhibitionPeriod $*$ Increase;
    $ProhibitionPeriod_t \leftarrow Iteration_i$;
  end
else
  VisitedSolutions $\leftarrow S_{curr}$;
  $Scurr_t \leftarrow Iteration_i$;
end
if $Iteration_i$-$ProhibitionPeriod_t > RepetitionInterval_{avg}$ **then**
  ProhibitionPeriod $\leftarrow$ `Max`(*1*, ProhibitionPeriod $*$ Decrease);
  $ProhibitionPeriod_t \leftarrow Iteration_i$;
end

Algorithm 13: Pseudo Code for the `BestMove` function in the Reactive Tabu Search algorithm.

Input: ProblemSize
Output: S_{curr}
$CandidateList_{admissible} \leftarrow$ `GetAdmissibleMoves(CandidateList)`;
$CandidateList_{tabu} \leftarrow$ CandidateList $- CandidateList_{admissible}$;
if `Size(`$CandidateList_{admissible}$`)` < 2 **then**
  ProhibitionPeriod $\leftarrow$ ProblemSize $- 2$;
  $ProhibitionPeriod_t \leftarrow Iteration_i$;
end
$S_{curr} \leftarrow$ `GetBest(`$CandidateList_{admissible}$`)`;
$Sbest_{tabu} \leftarrow$ `GetBest(`$CandidateList_{tabu}$`)`;
if `Cost(`$Sbest_{tabu}$`)` $<$ `Cost(`S_{best}`)` $\wedge$ `Cost(`$Sbest_{tabu}$`)` $<$ `Cost(`S_{curr}`)` **then**
  $S_{curr} \leftarrow Sbest_{tabu}$;
end
return S_{curr};

Algorithm 14: Pseudo Code for the `IsTabu` function in the Reactive Tabu Search algorithm.

Input:
Output: Tabu
Tabu $\leftarrow$ FALSE;
$Scurr^t_{feature} \leftarrow$ `RetrieveTimeFeatureLastUsed(`$Scurr_{feature}$`)`;
if $Scurr^t_{feature} \geq Iteration_{curr}$ $-$ProhibitionPeriod **then**
  Tabu $\leftarrow$ TRUE;
end
return Tabu;

- Reactive Tabu Search was proposed to use an long-term memory to diversify the search after a threshold of cycle repetitions has been reached.
- The `increase` parameter should be greater than one (such as 1.1 or 1.3) and the `decrease` parameter should be less than one (such as 0.9 or 0.8).

2.11.5 Code Listing

Listing 2.10 provides an example of the Reactive Tabu Search algorithm implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units.

The procedure is based on the code listing described by Battiti and Tecchiolli in [20]

with supplements like the `IsTabu` function from [18]. The implementation does not use efficient memory data structures such as hash tables. The algorithm is initialized with a stochastic 2-opt local search, and the neighborhood is generated as a fixed candidate list of stochastic 2-opt moves. The edges selected for changing in the 2-opt move are stored as features in the tabu list. The example does not implement the escape procedure for search diversification.

```
def euc_2d(c1, c2)
  Math::sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
end

def cost(permutation, cities)
  distance =0
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
    distance += euc_2d(cities[c1], cities[c2])
  end
  return distance
end

def random_permutation(cities)
  all = Array.new(cities.length) {|i| i}
  return Array.new(all.length) {|i| all.delete_at(rand(all.length))}
end

def stochastic_two_opt(permutation)
  perm = Array.new(permutation)
  c1, c2 = rand(perm.length), rand(perm.length)
  c2 = rand(perm.length) while c1 == c2
  c1, c2 = c2, c1 if c2 < c1
  perm[c1...c2] = perm[c1...c2].reverse
  return perm, [[permutation[c1-1], permutation[c1]], [permutation[c2-1],
      permutation[c2]]]
end

def generate_initial_solution(cities, max_no_improvements)
  best = {}
  best[:vector] = random_permutation(cities)
  best[:cost] = cost(best[:vector], cities)
  count = 0
  begin
    candidate = {}
    candidate[:vector] = stochastic_two_opt(best[:vector])[0]
    candidate[:cost] = cost(candidate[:vector], cities)
    if candidate[:cost] <= best[:cost]
      count, best = 0, candidate
    else
      count += 1
    end
  end until count >= max_no_improvements
  return best
end

```

```
46 def is_tabu?(edge, tabu_list, iteration, prohibition_period)
47   tabu_list.each do |entry|
48     if entry[:edge] == edge
49       if entry[:iteration] >= iteration-prohibition_period
50         return true
51       else
52         return false
53       end
54     end
55   end
56   return false
57 end
58 
59 def make_tabu(tabu_list, edge, iteration)
60   tabu_list.each do |entry|
61     if entry[:edge] == edge
62       entry[:iteration] = iteration
63       return entry
64     end
65   end
66   entry = {}
67   entry[:edge] = edge
68   entry[:iteration] = iteration
69   tabu_list.push(entry)
70   return entry
71 end
72 
73 def to_edge_list(permutation)
74   list = []
75   permutation.each_with_index do |c1, i|
76     c2 = (i==permutation.length-1) ? permutation[0] : permutation[i+1]
77     c1, c2 = c2, c1 if c1 > c2
78     list << [c1, c2]
79   end
80   return list
81 end
82 
83 def equivalent_permutations(edgelist1, edgelist2)
84   edgelist1.each do |edge|
85     return false if !edgelist2.include?(edge)
86   end
87   return true
88 end
89 
90 def generate_candidate(best, cities)
91   candidate = {}
92   candidate[:vector], edges = stochastic_two_opt(best[:vector])
93   candidate[:cost] = cost(candidate[:vector], cities)
94   return candidate, edges
95 end
96 
97 def get_candidate_entry(visited_list, permutation)
98   edgeList = to_edge_list(permutation)
```

```
99    visited_list.each do |entry|
100     return entry if equivalent_permutations(edgeList, entry[:edgelist])
101   end
102   return nil
103 end
104
105 def store_permutation(visited_list, permutation, iteration)
106   entry = {}
107   entry[:edgelist] = to_edge_list(permutation)
108   entry[:iteration] = iteration
109   entry[:visits] = 1
110   visited_list.push(entry)
111   return entry
112 end
113
114 def sort_neighbourhood(candidates, tabu_list, prohibition_period, iteration)
115   tabu, admissable = [], []
116   candidates.each do |a|
117     if is_tabu?(a[1][0], tabu_list, iteration, prohibition_period) or
118       is_tabu?(a[1][1], tabu_list, iteration, prohibition_period)
119       tabu << a
120     else
121       admissable << a
122     end
123   end
124   return tabu, admissable
125 end
126
127 def search(cities, max_no_improvements, max_candidates, max_iterations, increase,
      decrease)
128   current = generate_initial_solution(cities, max_no_improvements)
129   best = current
130   tabu_list, prohibition_period = [], 1
131   visited_list, avg_length, last_change = [], 1, 0
132   max_iterations.times do |iter|
133     candidate_entry = get_candidate_entry(visited_list, current[:vector])
134     if !candidate_entry.nil?
135       repetition_interval = iter - candidate_entry[:iteration]
136       candidate_entry[:iteration] = iter
137       candidate_entry[:visits] += 1
138       if repetition_interval < 2*(cities.length-1)
139         avg_length = 0.1*(iter-candidate_entry[:iteration]) + 0.9*avg_length
140         prohibition_period = (prohibition_period.to_f * increase)
141         last_change = iter
142       end
143     else
144       store_permutation(visited_list, current[:vector], iter)
145     end
146     if iter-last_change > avg_length
147       prohibition_period = [prohibition_period*decrease,1].max
148       last_change = iter
149     end
150     candidates = Array.new(max_candidates) {|i| generate_candidate(current, cities)}
```

```
151     candidates.sort! {|x,y| x.first[:cost] <=> y.first[:cost]}
152     tabu, admissible = sort_neighbourhood(candidates, tabu_list, prohibition_period,
            iter)
153     if admissible.length < 2
154       prohibition_period = cities.length-2
155       last_change = iter
156     end
157     current, best_move_edges = (admissible.empty?) ? tabu.first : admissible.first
158     if !tabu.empty? and tabu.first[0][:cost]<best[:cost] and
            tabu.first[0][:cost]<current[:cost]
159       current, best_move_edges = tabu.first
160     end
161     best_move_edges.each {|edge| make_tabu(tabu_list, edge, iter)}
162     best = candidates.first[0] if candidates.first[0][:cost] < best[:cost]
163     puts " > iteration #{(iter+1)}, tenure=#{prohibition_period.round},
            best=#{best[:cost]}"
164   end
165   return best
166 end
167
168 max_iterations = 300
169 max_no_improvements = 50
170 max_candidates = 50
171 increase = 1.3
172 decrease = 0.9
173 berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],[880,660],[25,230],
174  [525,1000],[580,1175],[650,1130],[1605,620],[1220,580],[1465,200],[1530,5],
175  [845,680],[725,370],[145,665],[415,635],[510,875],[560,365],[300,465],
176  [520,585],[480,415],[835,625],[975,580],[1215,245],[1320,315],[1250,400],
177  [660,180],[410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
178  [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],[95,260],
179  [875,920],[700,500],[555,815],[830,485],[1170,65],[830,610],[605,625],
180  [595,360],[1340,725],[1740,245]]
181
182 best = search(berlin52, max_no_improvements, max_candidates, max_iterations, increase,
        decrease)
183 puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 2.10: Reactive Tabu Search algorithm in the Ruby Programming Language

2.11.6 References

2.11.7 Primary Sources

Reactive Tabu Search was proposed by Battiti and Tecchiolli as an extension to Tabu Search that included an adaptive tabu list size in addition to a diversification mechanism [18]. The technique also used efficient memory structures that were based on an earlier work by Battiti and Tecchiolli that considered a parallel tabu search [17]. Some early application papers by Battiti and Tecchiolli include a comparison to Simulated Annealing applied to the Quadratic Assignment Problem [19], benchmarked on instances of the knapsack problem and N-K models and compared with Repeated Local Minima Search,

Simulated Annealing, and Genetic Algorithms [20], and training neural networks on an array of problem instances [21].

2.11.8 Learn More

Reactive Tabu Search was abstracted to a form called Reactive Local Search that considers adaptive methods that learn suitable parameters for heuristics that manage an embedded local search technique [15, 16]. Under this abstraction, the Reactive Tabu Search algorithm is single example of the Reactive Local Search principle applied to the Tabu Search. This framework was further extended to the use of any adaptive machine learning techniques to adapt the parameters of an algorithm by reacting to algorithm outcomes online while solving a problem, called Reactive Search [22]. The best reference for this general framework is the book on Reactive Search Optimization by Battiti, Brunato, and Mascia [14]. Additionally, the review chapter by Battiti and Brunato provides a contemporary description [13].

Chapter 3

Physical Algorithms

3.1 Overview

todo

3.2 Simulated Annealing

The heading and alternate headings for the algorithm description.

3.2.1 Taxonomy

A small tree diagram showing related fields and algorithms.

3.2.2 Inspiration

A textual description of the inspiring system.

3.2.3 Metaphor

A textual description of the algorithm by analogy.

3.2.4 Strategy

A textual description of the information processing strategy.

3.2.5 Procedure

A pseudo code description of the algorithms procedure.

3.2.6 Heuristics

A bullet-point listing of best practice usage.

3.2.7 Code Listing

A code listing and a terse description of the listing.

3.2.8 References

An bullet-point annotated reference list of primary sources and useful resources.

3.3 Adaptive Simulated Annealing

The heading and alternate headings for the algorithm description.

3.3.1 Taxonomy

A small tree diagram showing related fields and algorithms.

3.3.2 Inspiration

A textual description of the inspiring system.

3.3.3 Metaphor

A textual description of the algorithm by analogy.

3.3.4 Strategy

A textual description of the information processing strategy.

3.3.5 Procedure

A pseudo code description of the algorithms procedure.

3.3.6 Heuristics

A bullet-point listing of best practice usage.

3.3.7 Code Listing

A code listing and a terse description of the listing.

3.3.8 References

An bullet-point annotated reference list of primary sources and useful resources.

3.4 Memetic Algorithm

The heading and alternate headings for the algorithm description.

3.4.1 Taxonomy

A small tree diagram showing related fields and algorithms.

3.4.2 Inspiration

A textual description of the inspiring system.

3.4.3 Metaphor

A textual description of the algorithm by analogy.

3.4.4 Strategy

A textual description of the information processing strategy.

3.4.5 Procedure

A pseudo code description of the algorithms procedure.

3.4.6 Heuristics

A bullet-point listing of best practice usage.

3.4.7 Code Listing

A code listing and a terse description of the listing.

3.4.8 References

An bullet-point annotated reference list of primary sources and useful resources.

3.5 Extremal Optimization

The heading and alternate headings for the algorithm description.

3.5.1 Taxonomy

A small tree diagram showing related fields and algorithms.

3.5.2 Inspiration

A textual description of the inspiring system.

3.5.3 Metaphor

A textual description of the algorithm by analogy.

3.5.4 Strategy

A textual description of the information processing strategy.

3.5.5 Procedure

A pseudo code description of the algorithms procedure.

3.5.6 Heuristics

A bullet-point listing of best practice usage.

3.5.7 Code Listing

A code listing and a terse description of the listing.

3.5.8 References

An bullet-point annotated reference list of primary sources and useful resources.

3.6 Cultural Algorithm

The heading and alternate headings for the algorithm description.

3.6.1 Taxonomy

A small tree diagram showing related fields and algorithms.

3.6.2 Inspiration

A textual description of the inspiring system.

3.6.3 Metaphor

A textual description of the algorithm by analogy.

3.6.4 Strategy

A textual description of the information processing strategy.

3.6.5 Procedure

A pseudo code description of the algorithms procedure.

3.6.6 Heuristics

A bullet-point listing of best practice usage.

3.6.7 Code Listing

A code listing and a terse description of the listing.

3.6.8 References

An bullet-point annotated reference list of primary sources and useful resources.

3.7 Summary

todo

Chapter 4

Evolutionary Algorithms

4.1 Overview

This chapter describes Stochastic Algorithms.

Evolution

Evolutionary Algorithms belong to the Evolutionary Computation field of study concerned with computational methods inspired by the process and mechanisms of biological evolution. The process of evolution by means of natural selection (descent with modification) was proposed by Darwin to account for the variety of life and its suitability (adaptive fit) for its environment. The mechanisms of evolution describe how evolution actually takes place through the modification and propagation of genetic material (proteins). Evolutionary algorithms are concerned with investigating computations that resemble simplified versions of the processes and mechanisms of evolution toward achieving the effects of these processes and mechanisms, namely the development of adaptive systems. Additional subject areas that fall within the realm of Evolutionary Computation are algorithms that seek to exploit the properties from the related fields of Population Genetics, Population Ecology, Coevolutionary Biology, and Developmental Biology.

References

Evolutionary Algorithms share properties of adaptation through an iterative process of trial and error that accumulates and amplifies beneficial variation. Candidate solutions represent members of a virtual population striving to survive in an environment defined by a problem specific objective function. In each case, the evolutionary process refines the adaptive fit of the population of candidate solutions in the environment, typically using surrogates for the mechanisms of evolution such as genetic recombination and genetic mutation.

There are many excellent texts on the theory of evolution, although Darwin's original source can be an interesting and surprisingly enjoyable read [43]. Huxley's book defined

the modern synthesis in evolutionary biology that combined Darwin's natural selection with Mendel's genetic mechanisms [124], although any good textbook on evolution would suffice (such as Futuyma's 'Evolution' [86]). Popular science books on evolution are an easy place to start, such as Dawkins' 'The Selfish Gene' that presents on a gene-centric perspective on evolution [45], and Dennett's 'Darwin's Dangerous Idea' that considers on the algorithmic properties of the process [53].

Goldberg's classic text is still a valuable resource for the Genetic Algorithm [104], and Holland's text is interesting for those looking to learn about the research into adaptive systems that became the Genetic Algorithm [120]. Additionally, the seminal work by Koza should be considered for those interested in Genetic Programming [137], and Schwefel's seminal work should be considered for those with an interest in Evolution Strategies [202]. For an indepth review of the history of research into the use of simulated evolutionary processed for problem solving, see Fogel [75] For a rounded and modern review of the field of Evolutionary Computation Bäck, Fogel, and Michalewicz's two volumes of 'Evolutionary Computation' are an excellent resource covering the major techniques, theory, and application specific concerns [6, 7]. For some additional modern books on the unified field of Evolutionary Computation and Evolutionary Algorithms, see De Jong [128], a recent edition of Fogel [74], and Eiben and Smith [56].

4.2 Genetic Algorithm

Genetic Algorithm, GA, Simple Genetic Algorithm, SGA, Canonical Genetic Algorithm, CGA.

4.2.1 Taxonomy

The Genetic Algorithm is an Adaptive Strategy and a Global Optimization technique. It is an Evolutionary Algorithm and belongs to the broader study of Evolutionary Computation. The Genetic Algorithm is a sibling of other Evolutionary Algorithms such as Genetic Programming, Evolution Strategies, Genetic Programming, and Learning Classifier Systems. The Genetic Algorithm is a parent of a large number of variant techniques and sub-fields too numerous to list.

4.2.2 Inspiration

The Genetic Algorithm is inspired by population genetics (including heredity and gene frequencies), and evolution at the population level, as well as the Mendelian understanding of the structure (such as chromosomes, genes, alleles) and mechanisms (such as recombination and mutation). This is the so-called new or modern synthesis of evolutionary biology.

4.2.3 Metaphor

Individuals of a population contribute their genetic material (called the genotype) proportional to their suitability of their expressed genome (called their phenotype) to their environment. The next generation is created through a process of mating that involves recombination of two individuals genomes in the population with the introduction of random copying errors (called mutation). This iterative process may result in an improved adaptive-fit between the phenotypes of individuals in a population and the environment.

4.2.4 Strategy

The objective of the Genetic Algorithm is to maximize the payoff of candidate solutions in the population against a cost function from the problem domain. The strategy for the Genetic Algorithm is to repeatedly employ surrogates for the recombination and mutation genetic mechanisms on the population of candidate solutions, where the cost function (also known as objective or fitness function) applied to a decoded representation of a candidate governs the probabilistic contributions a given candidate solution can make to the subsequent generation of candidate solutions.

4.2.5 Procedure

Algorithm 15 provides a pseudo-code listing of the Genetic Algorithm for minimizing a cost function.

Algorithm 15: Pseudo Code for the Genetic Algorithm.

Input: $Population_{size}$, ProblemSize, $P_{crossover}$, $P_{mutation}$
Output: S_{best}

```
Population ← InitializePopulation(Population_size, ProblemSize);
EvaluatePopulation(Population);
S_best ← GetBestSolution(Population);
while ¬StopCondition() do
    Parents ← SelectParents(Population, Population_size);
    Children ← 0;
    foreach Parent_1, Parent_2 ∈ Parents do
        Child_1, Child_2 ← Crossover(Parent_1, Parent_2, P_crossover);
        Children ← Mutate(Child_1, P_mutation);
        Children ← Mutate(Child_2, P_mutation);
    end
    EvaluatePopulation(Children);
    S_best ← GetBestSolution(Children);
    Population ← Replace(Population, Children);
end
return S_best;
```

4.2.6 Heuristics

- Binary strings (bitstrings) are the classical representation as they can be decoded to almost any desired representation. Real-valued variables can be decoded using the binary coded decimal method or the gray code method, the latter of which is generally preferred.
- Problem specific representations and customized genetic operators should be adopted, incorporating as much prior information about the problem domain as possible.
- The algorithm is highly-modular and a sub-field exists to study each sub-process, such as selection, recombination, mutation, and representation.
- The Genetic Algorithm is most commonly used as an optimization technique, although it should also be considered a general adaptive strategy [127].
- The schema theorem is a classical explanation for the power of the Genetic Algorithm proposed by Holland [120], and investigated by Goldberg under the name of the building block hypothesis [104].
- The size of the population must be large enough to provide sufficient coverage of the domain and mixing of the useful sub-components of the solution [102].
- The Genetic Algorithm is classically configured with a high probability of recombination (such as 95%-99% of the selected population) and a low probability of

mutation (such as $\frac{1}{L}$ where L is the number of components in a solution) [168, 5].

- The fitness-proportionate selection of candidate solutions to contribute to the next generation should be neither too greedy (to avoid the takeover of fitter candidate solutions) nor too random.

4.2.7 Code Listing

Listing 4.1 provides an example of the Genetic Algorithm implemented in the Ruby Programming Language. The demonstration problem is a maximizing binary optimization problem called OneMax that seeks a binary string of unity (all '1' bits). The objective function provides only an indication of the number of correct bits in a candidate string, not the positions of the correct bits.

The Genetic Algorithm is implemented with a conservative configuration including binary tournament selection for the selection operator, uniform crossover for the recombination operator, and point mutations for the mutation operator.

```
def onemax(bitstring)
  sum = 0
  bitstring.each_char {|x| sum+=1 if x=='1'}
  return sum
end

def binary_tournament(population)
  s1, s2 = population[rand(population.size)], population[rand(population.size)]
  return (s1[:fitness] > s2[:fitness]) ? s1 : s2
end

def point_mutation(bitstring, prob_mutation)
  child = ""
  bitstring.size.times do |i|
    bit = bitstring[i]
    child << ((rand()<prob_mutation) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

def uniform_crossover(parent1, parent2, p_crossover)
  return ""+parent1[:bitstring] if rand()>=p_crossover
  child = ""
  parent1[:bitstring].size.times do |i|
    child << ((rand()<0.5) ? parent1[:bitstring][i] : parent2[:bitstring][i])
  end
  return child
end

def reproduce(selected, population_size, p_crossover, p_mutation)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
```

```
    child[:bitstring] = uniform_crossover(p1, p2, p_crossover)
    child[:bitstring] = point_mutation(child[:bitstring], p_mutation)
    children << child
  end
  return children
end

def random_bitstring(num_bits)
  return (0...num_bits).inject(""){|s,i| s<<((rand<0.5) ? "1" : "0")}
end

def search(max_generations, num_bits, population_size, p_crossover, p_mutation)
  population = Array.new(population_size) do |i|
    {:bitstring=>random_bitstring(num_bits)}
  end
  population.each{|c| c[:fitness] = onemax(c[:bitstring])}
  gen, best = 0, population.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  while best[:fitness]!=num_bits and gen<max_generations
    selected = Array.new(population_size){|i| binary_tournament(population)}
    children = reproduce(selected, population_size, p_crossover, p_mutation)
    children.each{|c| c[:fitness] = onemax(c[:bitstring])}
    children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
    best = children.first if children.first[:fitness] >= best[:fitness]
    population = children
    gen += 1
    puts " > gen #{gen}, best: #{best[:fitness]}, #{best[:bitstring]}"
  end
  return best
end

max_generations = 100
population_size = 100
num_bits = 64
p_crossover = 0.98
p_mutation = 1.0/num_bits

best = search(max_generations, num_bits, population_size, p_crossover, p_mutation)
puts "done! Solution: f=#{best[:fitness]}, s=#{best[:bitstring]}"
```

Listing 4.1: Genetic Algorithm in the Ruby Programming Language

4.2.8 References

Primary Sources

Holland is the grandfather of the field that became Genetic Algorithms. Holland investigated adaptive systems in the late 1960s proposing an adaptive system formalism and adaptive strategies referred to as 'adaptive plans' [117, 113, 114]. Holland's theoretical framework was investigated and elaborated by his Ph.D. students at the University of Michigan. Rosenberg investigated a chemical and molecular model of a biological inspired adaptive plan [193]. Bagley investigated meta-environments and a genetic adap-

tive plan referred to as a genetic algorithm applied to a simple game called hexapawn [10]. Cavicchio further elaborated the genetic adaptive plan by proposing numerous variations, referring to some as 'reproductive plans' [131].

Other important contributions were made by Frantz who investigated what were referred to as genetic algorithms for search [84], and Hollstien who investigated genetic plans for adaptive control and function optimization [121]. De Jong performed a seminal investigation of the genetic adaptive model (genetic plans) applied to continuous function optimization and his suite of test problems adopted are still commonly used [129]. Holland wrote the the seminal book on his research focusing on the proposed adaptive systems formalism, the reproductive and genetic adaptive plans, and provided a theoretical framework for the mechanisms used and explanation for the capabilities of what would become genetic algorithms [120].

Learn More

The field of genetic algorithms is very large, resulting in large numbers of variations on the canonical technique. Goldberg provides a classical overview of the field in a review article [103], as does Mitchell [162]. Whitley describes a classical tutorial for the genetic algorithm covering both practical and theoretical concerns [236]. The classical book on genetic algorithms as an optimization and machine learning technique was written by Goldberg and provides an in-depth review and practical study of the approach [104]. Mitchell provides a contemporary reference text introducing the technique and the field [163]. Finally, Goldberg provides a modern study of the field, the lessons learned, and reviews the broader toolset of optimization algorithms that the field has produced [105].

4.3 Genetic Programming

Genetic Programming, GP.

4.3.1 Taxonomy

The Genetic Programming algorithm is an example of a Evolutionary Algorithm (EA) and belongs to the field of Evolutionary Computation (EC) and more broadly Computational Intelligence and Biologically Inspired Computation. The Genetic Programming algorithm is a sibling to other Evolutionary Algorithms such as the Genetic Algorithm, Evolution Strategies, Evolutionary Programming, and Learning Classifier Systems. Technically, the Genetic Programming algorithm is an extension of the Genetic Algorithm. The Genetic Algorithm is a parent to a host of variations and extensions.

4.3.2 Inspiration

The Genetic Algorithm is inspired by population genetics (including heredity and gene frequencies), and evolution at the population level, as well as the Mendelian understanding of the structure (such as chromosomes, genes, alleles) and mechanisms (such as recombination and mutation). This is the so-called new or modern synthesis of evolutionary biology.

4.3.3 Metaphor

Individuals of a population contribute their genetic material (called the genotype) proportional to their suitability of their expressed genome (called their phenotype) to their environment. The next generation is created through a process of mating that involves genetic operators such as recombination of two individuals genomes in the population and the introduction of random copying errors (called mutation). This iterative process may result in an improved adaptive-fit between the phenotypes of individuals in a population and the environment.

Programs may be evolved and used in a secondary adaptive process, where an assessment of candidates at the end of the secondary adaptive process is used for differential reproductive success in the first evolutionary process. This system may be understood as the inter-dependencies experienced in evolutionary development where evolution operates upon an embryo that in turn develops into an individual in an environment that eventually may reproduce.

4.3.4 Strategy

The objective of the Genetic Programming algorithm is to use induction to devise a computer program. This is achieved by using evolutionary operators on candidate programs with a tree structure to improve the adaptive fit between the population of candidate programs and an objective function. An assessment of a candidate solution involves its execution.

4.3.5 Procedure

Algorithm 16 provides a pseudo-code listing of the Genetic Programming algorithm for minimizing a cost function, based on Koza and Poli's tutorial [141].

The Genetic Program uses LISP-like symbolic expressions called S-expressions that represent the graph of a program with function nodes and terminal nodes. While the algorithm is running, the programs are treated like data, and when they are evaluated they are executed. The traversal of a program graph is always depth first, and functions must always return a value.

4.3.6 Heuristics

- The Genetic Programming algorithm was designed for inductive automatic programming and is well suited to symbolic regression, controller design, and machine learning tasks under the broader name of function approximation.
- Traditionally symbolic expressions are evolved and evaluated in a virtual machine, although the approach has been applied with real programming languages.
- The evaluation (fitness assignment) of a candidate solution typically takes the structure of the program into account, rewarding parsimony.
- The selection process should be balanced between random selection and greedy selection to bias the search towards fitter candidate solutions (exploitation), whilst promoting useful diversity into the population (exploration).
- A program may respond to zero or more input values and may produce one or more outputs.
- All functions used in the function node set must return a usable result. For example, the division function must return a value (such as zero or one) when a division by zero occurs.
- All genetic operations ensure (or should ensure) that syntactically valid and executable programs are produced as a result of their application.
- The Genetic Programming algorithm is commonly configured with a high-probability of crossover ($\geq 90\%$) and a low-probability of mutation ($\leq 1\%$). Other operators such as reproduction and architecture alterations are used with moderate-level probabilities and fill in the probabilistic gap.
- Architecture altering operations are not limited to the duplication and deletion of sub-structures of a given program.
- The crossover genetic operator in the algorithm is commonly configured to select a function as a the cross-point with a high-probability ($\geq 90\%$) and low-probability of selecting a terminal as a cross-point ($\leq 10\%$).

Algorithm 16: Pseudo Code for the Genetic Programming algorithm.

Input: $Population_{size}$, $nodes_{func}$, $nodes_{term}$, $P_{crossover}$, $P_{mutation}$, $P_{reproduction}$, $P_{alteration}$
Output: S_{best}

```
1  Population ← InitializePopulation(Population_size, nodes_func, nodes_term);
2  EvaluatePopulation(Population);
3  S_best ← GetBestSolution(Population);
4  while ¬StopCondition() do
5      Children ← 0;
6      while ¬StopCondition(Size(Children) < Population_size) do
7          Operator ← SelectGeneticOperator(P_crossover, P_mutation, P_reproduction,
           P_alteration);
8          if Operator ≡ CrossoverOperator then
9              Parent_1, Parent_2 ← SelectParents(Population, Population_size);
10             Child_1, Child_2 ← Crossover(Parent_1, Parent_2);
11             Children ← Child_1;
12             Children ← Child_2;
13         end
14         else if Operator ≡ MutationOperator then
15             Parent_1 ← SelectParents(Population, Population_size);
16             Child_1 ← Mutate(Parent_1);
17             Children ← Child_1;
18         end
19         else if Operator ≡ ReproductionOperator then
20             Parent_1 ← SelectParents(Population, Population_size);
21             Child_1 ← Reproduce(Parent_1);
22             Children ← Child_1;
23         end
24         else if Operator ≡ AlterationOperator then
25             Parent_1 ← SelectParents(Population, Population_size);
26             Child_1 ← AlterArchitecture(Parent_1);
27             Children ← Child_1;
28         end
29     end
30     EvaluatePopulation(Children);
31     S_best ← GetBestSolution(Children, S_best);
32     Population ← Replace(Population, Children);
33 end
34 return S_best;
```

- The function set may also include control structures such as conditional statements and loop constructs.
- The Genetic Programing algorithm can be realized as a stack-based virtual machine as opposed to a call graph [179].
- The Genetic Programming algorithm can make use of Automatically Defined Functions (ADFs) that are sub-graphs and are promoted to the status of functions for reuse and are co-evolved with the programs.
- The genetic operators employed during reproduction in the algorithm may be considered transformation programs for candidate solutions and may themselves be co-evolved in the algorithm [4].

4.3.7 Code Listing

Listing 4.2 provides an example of the Genetic Programming algorithm implemented in the Ruby Programming Language based on Koza and Poli's tutorial [141].

The demonstration problem is an instance of a symbolic regression, where a function must be devised to match a set of observations. In this case the target function is a quadratic polynomial $x^2 + x + 1$ where $x \in [-1, 1]$. The observations are generated directly from the target function without noise for the purposes of this example. In practical problems, if one knew and had access to the target function then the genetic program would not be required.

The algorithm is configured to search for a program with the function set $\{+, -, \times, \div\}$ and the terminal set $\{X, R\}$, where X is the input value, and R is a static random variable generated for a program $X \in [-5, 5]$. A division by zero returns a value of one. The fitness of a candidate solution is calculated by evaluating the program on range of random input values and calculating the Root Mean Squared Error (RMSE). The algorithm is configured with a 90% probability of crossover, 9% probability of reproduction (copying), and a 2% probability of mutation. For brevity, the algorithm does not implement the architecture altering genetic operation and does not bias crossover points towards functions over terminals.

```
1  def random_num(min, max)
2    return min + (max-min)*rand()
3  end
4
5  def print_program(node)
6    return node if !node.kind_of? Array
7    return "(#{node[0]}, #{print_program(node[1])}, #{print_program(node[2])})"
8  end
9
10 def eval_program(node, map)
11   if !node.kind_of? Array
12     return map[node].to_f if !map[node].nil?
13     return node.to_f
14   end
```

```
  arg1, arg2 = eval_program(node[1], map), eval_program(node[2], map)
  return 0 if node[0] === :/ and arg2 == 0.0
  return arg1.__send__(node[0], arg2)
end

def generate_random_program(max, funcs, terms, depth=0)
  if depth==max-1 or (depth>1 and rand()<0.1)
    t = terms[rand(terms.length)]
    return ((t=='R') ? random_num(-5.0, +5.0) : t)
  end
  depth += 1
  arg1 = generate_random_program(max, funcs, terms, depth)
  arg2 = generate_random_program(max, funcs, terms, depth)
  return [funcs[rand(funcs.length)], arg1, arg2]
end

def count_nodes(node)
  return 1 if !node.kind_of? Array
  a1 = count_nodes(node[1])
  a2 = count_nodes(node[2])
  return a1+a2+1
end

def target_function(input)
  return input**2 + input + 1
end

def fitness(program, num_trials)
  sum_error = 0.0
  num_trials.times do |i|
    input = random_num(-1.0, 1.0)
    error = eval_program(program, {'X'=>input}) - target_function(input)
    sum_error += error**2.0
  end
  return Math::sqrt(sum_error/num_trials.to_f)
end

def tournament_selection(population, num_bouts)
  best = population[rand(population.size)]
  (num_bouts-1).times do |i|
    candidate = population[rand(population.size)]
    best = candidate if candidate[:fitness] < best[:fitness]
  end
  return best
end

def replace_node(node, replacement, node_num, current_node=0)
  return replacement,(current_node+1) if current_node == node_num
  current_node += 1
  return node,current_node if !node.kind_of? Array
  a1, current_node = replace_node(node[1], replacement, node_num, current_node)
  a2, current_node = replace_node(node[2], replacement, node_num, current_node)
  return [node[0], a1, a2], current_node
```

```
68  end
69
70  def copy_program(node)
71    return node if !node.kind_of? Array
72    return [node[0], copy_program(node[1]), copy_program(node[2])]
73  end
74
75  def get_node(node, node_num, current_node=0)
76    return node,(current_node+1) if current_node == node_num
77    current_node += 1
78    return nil,current_node if !node.kind_of? Array
79    a1, current_node = get_node(node[1], node_num, current_node)
80    return a1,current_node if !a1.nil?
81    a2, current_node = get_node(node[2], node_num, current_node)
82    return a2,current_node if !a2.nil?
83    return nil,current_node
84  end
85
86  def prune(node, max_depth, terms, depth=0)
87    if depth >= max_depth-1
88      t = terms[rand(terms.length)]
89      return ((t=='R') ? random_num(-5.0, +5.0) : t)
90    end
91    depth += 1
92    return node if !node.kind_of? Array
93    a1 = prune(node[1], max_depth, terms, depth)
94    a2 = prune(node[2], max_depth, terms, depth)
95    return [node[0], a1, a2]
96  end
97
98  def crossover(parent1, parent2, max_depth, terms)
99    pt1, pt2 = rand(count_nodes(parent1)-2)+1, rand(count_nodes(parent2)-2)+1
100   tree1, c1 = get_node(parent1, pt1)
101   tree2, c2 = get_node(parent2, pt2)
102   child1, c1 = replace_node(parent1, copy_program(tree2), pt1)
103   child1 = prune(child1, max_depth, terms)
104   child2, c2 = replace_node(parent2, copy_program(tree1), pt2)
105   child2 = prune(child2, max_depth, terms)
106   return child1, child2
107 end
108
109 def mutation(parent, max_depth, functions, terms)
110   random_tree = generate_random_program(max_depth/2, functions, terms)
111   point = rand(count_nodes(parent))
112   child, count = replace_node(parent, random_tree, point)
113   child = prune(child, max_depth, terms)
114   return child
115 end
116
117 def search(max_generations, population_size, max_depth, num_trials, num_bouts,
        p_reproduction, p_crossover, p_mutation, functions, terminals)
118   population = Array.new(population_size) do |i|
119     {:program=>generate_random_program(max_depth, functions, terminals)}
```

```
  end
  population.each{|c| c[:fitness] = fitness(c[:program], num_trials)}
  best = population.sort{|x,y| x[:fitness] <=> y[:fitness]}.first
  max_generations.times do |gen|
    children = []
    while children.length < population_size
      operation = rand()
      parent = tournament_selection(population, num_bouts)
      child = {}
      if operation < p_reproduction
        child[:program] = copy_program(parent[:program])
      elsif operation < p_reproduction+p_crossover
        p2 = tournament_selection(population, num_bouts)
        c2 = {}
        child[:program], c2[:program] = crossover(parent[:program], p2[:program],
            max_depth, terminals)
        children << c2
      elsif operation < p_reproduction+p_crossover+p_mutation
        child[:program] = mutation(parent[:program], max_depth, functions, terminals)
      end
      children << child if children.length < population_size
    end
    children.each{|c| c[:fitness] = fitness(c[:program], num_trials)}
    population = children
    population.sort!{|x,y| x[:fitness] <=> y[:fitness]}
    best = population.first if population.first[:fitness] <= best[:fitness]
    puts " > gen #{gen}, fitness=#{best[:fitness]}"
    break if best[:fitness] == 0
  end
  return best
end

max_generations = 100
max_depth = 7
population_size = 100
num_trials = 15
num_bouts = 5
p_reproduction = 0.08
p_crossover = 0.90
p_mutation = 0.02
terminals = ['X', 'R']
functions = [:+, :-, :*, :/]

best = search(max_generations, population_size, max_depth, num_trials, num_bouts,
    p_reproduction, p_crossover, p_mutation, functions, terminals)
puts "done! Solution: f=#{best[:fitness]}, s=#{print_program(best[:program])}"
```

Listing 4.2: Genetic Programming algorithm in the Ruby Programming Language

4.3.8 References

Primary Sources

An early work by Cramer involved the study of a Genetic Algorithm using an expression tree structure for representing computer programs for primitive mathematical operations [41]. Koza is credited with the development of the field of Genetic Programming. An early paper by Koza referred to his hierarchical genetic algorithms as an extension to the simple genetic algorithm that use symbolic expressions (S-expressions) as a representation and were applied to a range of induction-style problems [136]. The seminal reference for the field is Koza's 1992 book on Genetic Programming [137].

Learn More

The field of Genetic Programming is vast, including many books, dedicated conferences and uncounted thousands of publications. Koza is generally credited with the development and popularizing of the field, publishing a large number of books and papers himself. Koza provides a practical introduction to the field as a tutorial [141], and provides recent overview of the broader field and usage of the technique [142].

In addition his the seminal 1992 book, Koza has released three more volumes in the series including volume II on automatically defined functions (ADFs) [138], volume III that considered the Genetic Programming Problem Solver (GPPS) for automatically defining the function set and program structure for a given problem [139], and volume IV that focuses on the human competitive results the technique is able to achieve in a routine manner [140]. All books are rich with targeted and practical demonstration problem instances.

Some additional excellent books include Banzhaf, et al's introduction to the field [11], Langdon and Poli's detailed look at the technique [147], and Poli, Langdon, and McPhee's contemporary and practical field guide to Genetic Programming [181].

4.4 Evolutionary Programming

Evolutionary Programming, EP.

4.4.1 Taxonomy

Evolutionary Programming is a Global Optimization algorithm and is an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. The approach is a sibling of other Evolutionary Algorithms such as the Genetic Algorithm, and Learning Classifier Systems. It is sometimes confused with Genetic Programming given the similarity in name, and more recently it shows a strong functional similarity to Evolution Strategies.

4.4.2 Inspiration

Evolutionary Programming is inspired by the theory of evolution by means of natural selection. Specifically, the technique is inspired by macro-level or the species-level process of evolution (phenotype, hereditary, variation) and is not concerned with the genetic mechanisms of evolution (genome, chromosomes, genes, alleles).

4.4.3 Metaphor

A population of a species reproduce, creating progeny with small phenotypical variation. The progeny and the parents compete based on their suitability to the environment, where the generally more fit members constitute the subsequent generation and are provided with the opportunity to reproduce themselves. This process repeats, improving the adaptive fit between the species and the environment.

4.4.4 Strategy

The objective of the Evolutionary Programming algorithm is to maximize the suitability of collection of candidate solutions in the context of an objective function from the domain. This objective is pursued by using an adaptive model with surrogates for the processes of evolution, specifically hereditary (reproduction with variation) under competition. The representation used for candidate solutions is directly assessable by a cost or objective function from the domain, and credit assignment is directly apportioned to this representation.

4.4.5 Procedure

Algorithm 17 provides a pseudo-code listing of the Evolutionary Programming algorithm for minimizing a cost function.

Algorithm 17: Pseudo Code for the Evolutionary Programming algorithm.

Input: $Population_{size}$, ProblemSize, BoutSize
Output: S_{best}

```
Population ← InitializePopulation(Population_size, ProblemSize);
EvaluatePopulation(Population);
S_best ← GetBestSolution(Population);
while ¬StopCondition() do
    Children ← 0;
    foreach Parent_i ∈ Population do
        Child_i ← Mutate(Parent_i);
        Children ← Child_i;
    end
    EvaluatePopulation(Children);
    S_best ← GetBestSolution(Children, S_best);
    Union ← Population + Children;
    foreach S_i ∈ Union do
        for 1 to BoutSize do
            S_j ← RandomSelection(Union);
            if Cost(S_i) < Cost(S_j) then
                Si_wins ← Si_wins + 1;
            end
        end
    end
    Population ← SelectBestByWins(Union, Population_size);
end
return S_best;
```

4.4.6 Heuristics

- The representation for candidate solutions should be domain specific, such as real numbers for continuous function optimization.
- The sample size (bout size) for tournament selection during competition is commonly between 5% and 10% of the population size.
- Evolutionary Programming traditionally only uses the mutation operator to create new candidate solutions from existing candidate solutions. The crossover operator that is used in some other Evolutionary Algorithms is not employed in Evolutionary Programming.
- Evolutionary Programming is concerned with the linkage between parent and child candidate solutions and is not concerned with surrogates for genetic mechanisms.
- Continuous function optimization is a popular application for the approach, where

real-valued representations are with a Gaussian-based mutation operator.

- The mutation-specific parameters used in the application of the algorithm to continuous function optimization can be adapted in concert with the candidate solutions [73].

4.4.7 Code Listing

Listing 4.3 provides an example of the Evolutionary Programming algorithm implemented in the Ruby Programming Language. The demonstration problem is an instance of a continuous function optimization that seeks $minf(x)$ where $f = \sum_{i=1}^{n} x_i^2$, $-5.0 \leq x_i \leq 5.0$ and $n = 2$. The optimal solution for this basin function is $(v_0, \ldots, v_{n-1}) = 0.0$. The algorithm is an implementation of Evolutionary Programming based on Fogel et al's classical implementation for continuous function optimization [73] with per-variable adaptive variance based on Fogel's description for a self-adaptive variation on page 160 of his 1995 book [74].

```
def objective_function(vector)
  return vector.inject(0.0) {|sum, x| sum + (x ** 2.0)}
end

def random_vector(problem_size, search_space)
  return Array.new(problem_size) do |i|
    search_space[i][0] + ((search_space[i][1] - search_space[i][0]) * rand())
  end
end

def random_gaussian
  u1 = u2 = w = g1 = g2 = 0
  begin
    u1 = 2 * rand() - 1
    u2 = 2 * rand() - 1
    w = u1 * u1 + u2 * u2
  end while w >= 1
  w = Math::sqrt((-2 * Math::log(w)) / w)
  g2 = u1 * w;
  g1 = u2 * w;
  return g1
end

def mutate(candidate, search_space)
  child = {}
  child[:vector], child[:strategy] = [], []
  candidate[:vector].each_with_index do |v_old, i|
    s_old = candidate[:strategy][i]
    v = v_old + s_old * random_gaussian()
    v = search_space[i][0] if v < search_space[i][0]
    v = search_space[i][1] if v > search_space[i][1]
    child[:vector] << v
    s = s_old + ((s_old<0) ? s_old*-1.0 : s_old)**0.5 * random_gaussian()
    child[:strategy] << s
```

```ruby
  end
  return child
end

def tournament(candidate, population, bout_size)
  candidate[:wins] = 0
  bout_size.times do |i|
    other = population[rand(population.length)]
    candidate[:wins] += 1 if candidate[:fitness] < other[:fitness]
  end
end

def search(max_generations, problem_size, search_space, pop_size, bout_size)
  strategy_space = Array.new(problem_size) do |i|
    [0, (search_space[i][1]-search_space[i][0])*0.02]
  end
  population = Array.new(pop_size) do |i|
    {:vector=>random_vector(problem_size, search_space),
      :strategy=>random_vector(problem_size, strategy_space)}
  end
  population.each{|c| c[:fitness] = objective_function(c[:vector])}
  gen, best = 0, population.sort{|x,y| x[:fitness] <=> y[:fitness]}.first
  max_generations.times do |gen|
    children = Array.new(pop_size) {|i| mutate(population[i], search_space)}
    children.each{|c| c[:fitness] = objective_function(c[:vector])}
    children.sort!{|x,y| x[:fitness] <=> y[:fitness]}
    best = children.first if children.first[:fitness] < best[:fitness]
    union = children+population
    union.each{|c| tournament(c, union, bout_size)}
    union.sort!{|x,y| y[:wins] <=> x[:wins]}
    population = union[0...pop_size]
    puts " > gen #{gen}, fitness=#{best[:fitness]}"
  end
  return best
end

max_generations = 200
population_size = 100
problem_size = 2
search_space = Array.new(problem_size) {|i| [-5, +5]}
bout_size = 5

best = search(max_generations, problem_size, search_space, population_size, bout_size)
puts "done! Solution: f=#{best[:fitness]}, s=#{best[:vector].inspect}"
```

Listing 4.3: Evolutionary Programming algorithm in the Ruby Programming Language

4.4.8 References

Primary Sources

Evolutionary Programming was developed by Lawrence Fogel, outlined in early papers (such as [76]) and later became the focus of his PhD dissertation [77]. Fogel focused on the use of an evolutionary process for the development of control systems using Finite State Machine (FSM) representations. Fogel's early work on Evolutionary Programming culminated in a book, co-authored with Owens and Walsh that elaborated the approach, focusing on the evolution of state machines for the prediction of symbols in time series data [80].

Learn More

The field of Evolutionary Programming lay relatively dormant for 30 years until it was revived by Fogel's son, David Fogel. Early works considered the application of Evolutionary Programming to control systems [203], and later function optimization (system identification) culminating in a book on the approach [71], and David Fogel's PhD dissertation [72]. Lawrence Fogel collaborated in the revival of the technique, including reviews [78, 79] and extensions on what became the focus of the approach on function optimization [73].

Yao, et al. provide a seminal study of Evolutionary Programming proposing an extension and racing it against the classical approach on a large number of test problems [244]. Finally, Porto provides an excellent contemporary overview of the field and the technique [182].

4.5 Evolution Strategies

Evolution Strategies, Evolution Strategy, Evolutionary Strategies, ES.

4.5.1 Taxonomy

Evolution Strategies is a global optimization algorithm and is an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. Evolution Strategies is a sibling technique to other Evolutionary Algorithms such as Genetic Algorithms, Genetic Programming, Learning Classifier Systems, and Evolutionary Programming. A popular descendant of the Evolution Strategy algorithm is the Covariance Matrix Adaptation Evolution Strategy.

4.5.2 Inspiration

Evolution Strategies is inspired by the theory of evolution by means of natural selection. Specifically, the technique is inspired by macro-level or the species-level process of evolution (phenotype, hereditary, variation) and is not concerned with the genetic mechanisms of evolution (genome, chromosomes, genes, alleles).

4.5.3 Metaphor

Evolution Strategies only briefly flirted with explanation via metaphor, and is less preferred to grounded probabilistic explanations.

4.5.4 Strategy

The objective of the Evolution Strategies algorithm is to maximize the suitability of collection of candidate solutions in the context of an objective function from a domain. The objective was classically achieved through the adoption of dynamic variation, a surrogate for descent with modification, where the amount of variation was adapted dynamically with performance-based heuristics. Contemporary approaches co-adapt parameters that control the amount and bias of variation with the candidate solutions.

4.5.5 Procedure

Instances of Evolution Strategy algorithms may be concisely described with a custom terminology in the form $(\mu, \lambda) - ES$, where μ is number of candidate solution in the parent generation, and λ is the number of candidate solutions generated from and replace the parent generation. In addition to the so-called comma-selection Evolution Strategy, a plus-selection variation may be defined $(\mu + \lambda) - ES$, where the best members of the union of the μ and λ generations complete based on objective fitness for a position in the next generation. The simplest configuration is the $(1 + 1) - ES$ which is a type of greedy hill climbing algorithm. Algorithm 18 provides a pseudo-code listing of the $(\mu+\lambda)-ES$ Evolution Strategy algorithm for minimizing a cost function. The algorithm

shows the adaptation of candidate solutions that co-adapt their own strategy parameters that influence the amount of mutation applied to a candidate solutions descendants.

Algorithm 18: Pseudo Code for the Evolution Strategies algorithm.

```
   Input: μ, λ, ProblemSize
   Output: S_best
 1 Population ← InitializePopulation(μ, ProblemSize);
 2 EvaluatePopulation(Population);
 3 S_best ← GetBest(Population, 1);
 4 while ¬StopCondition() do
 5 |   Children ← 0;
 6 |   i ← 0;
 7 |   while Size(Children) < λ do
 8 |   |   S_i ← 0;
 9 |   |   Si_problem ← Mutate(Pi_problem, Pi_strategy);
10 |   |   Si_strategy ← Mutate(Pi_strategy);
11 |   |   Children ← S_i;
12 |   |   i ← i + 1;
13 |   end
14 |   EvaluatePopulation(Children);
15 |   S_best ← GetBest(Children + S_best, 1);
16 |   Population ← GetBest(Children, μ);
17 end
18 return S_best;
```

4.5.6 Heuristics

- Evolution Strategies uses problem specific representations, such as real values for continuous function optimization.
- The algorithm is commonly configured such that $1 < \mu < \lambda < \infty$.
- The ratio of μ to λ influences the amount of selection pressure (greediness) exerted by the algorithm.
- A contemporary update to the algorithms notation includes a ρ as $(\mu/\rho, \lambda) - ES$ that specifies the number of parents that will contribute to each new candidate solution using a recombination operator.
- A classical rule used to govern the amount of mutation (standard deviation used in mutation for continuous function optimization) was the $\frac{1}{5}$-rule, where the ratio of successful mutations should be $\frac{1}{5}$ of all mutations. If it is greater the variance is increased, otherwise if the ratio is is less, the variance is decreased.

- The comma-selection variation of the algorithm can be good for dynamic problem instances given it's capability for continued exploration of the search space, whereas the plus-selection variation can be good for refinement and convergence.

4.5.7 Code Listing

Listing 4.4 provides an example of the Evolution Strategies algorithm implemented in the Ruby Programming Language. The demonstration problem is an instance of a continuous function optimization that seeks $minf(x)$ where $f = \sum_{i=1}^{n} x_i^2$, $-5.0 \leq x_i \leq 5.0$ and $n = 2$. The optimal solution for this basin function is $(v_0, \ldots, v_{n-1}) = 0.0$. The algorithm is a implementation of Evolution Strategies based on simple version described by Bäck and Schwefel [9], which was also used as the basis of a detailed empirical study [243]. The algorithm is an $(30 + 20) - ES$ Evolutionary Strategy that adapts both the problem and strategy (standard deviations) variables. More contemporary implementations may modify the strategy variables differently, and include an additional set of adapted strategy parameters to influence the direction of mutation (see [194] for a concise description).

```
def objective_function(vector)
  return vector.inject(0.0) {|sum, x| sum + (x ** 2.0)}
end

def random_vector(problem_size, search_space)
  return Array.new(problem_size) do |i|
    search_space[i][0] + ((search_space[i][1] - search_space[i][0]) * rand())
  end
end

def gaussian
  u1 = u2 = w = g1 = g2 = 0
  begin
    u1 = 2 * rand() - 1
    u2 = 2 * rand() - 1
    w = u1 * u1 + u2 * u2
  end while w >= 1
  w = Math::sqrt((-2 * Math::log(w)) / w)
  g2 = u1 * w;
  g1 = u2 * w;
  return g1
end

def mutate_problem(vector, stdevs, search_space)
  child = Array(vector.length)
  vector.each_with_index do |v, i|
    child[i] = v + stdevs[i] * gaussian()
    child[i] = search_space[i][0] if child[i] < search_space[i][0]
    child[i] = search_space[i][1] if child[i] > search_space[i][1]
  end
  return child
end
```

```
33
34 def mutate_strategy(stdevs)
35   tau = Math.sqrt(2.0*stdevs.length.to_f)**-1.0
36   tau_prime = Math.sqrt(2.0*Math.sqrt(stdevs.length.to_f))**-1.0
37   child = Array.new(stdevs.length) do |i|
38     stdevs[i] * Math::exp(tau_prime*gaussian() + tau*gaussian())
39   end
40   return child
41 end
42
43 def mutate(parent, search_space)
44   child = {}
45   child[:vector] = mutate_problem(parent[:vector], parent[:strategy], search_space)
46   child[:strategy] = mutate_strategy(parent[:strategy])
47   return child
48 end
49
50 def search(max_generations, problem_size, search_space, pop_size, num_children)
51   strategy_space = Array.new(problem_size) do |i|
52     [0, (search_space[i][1]-search_space[i][0])*0.02]
53   end
54   population = Array.new(pop_size) do |i|
55     {:vector=>random_vector(problem_size, search_space),
56       :strategy=>random_vector(problem_size, strategy_space)}
57   end
58   population.each{|c| c[:fitness] = objective_function(c[:vector])}
59   best = population.sort{|x,y| x[:fitness] <=> y[:fitness]}.first
60   max_generations.times do |gen|
61     children = Array.new(num_children) {|i| mutate(population[i], search_space)}
62     children.each{|c| c[:fitness] = objective_function(c[:vector])}
63     union = children+population
64     union.sort!{|x,y| x[:fitness] <=> y[:fitness]}
65     best = union.first if union.first[:fitness] < best[:fitness]
66     population = union[0...pop_size]
67     puts " > gen #{gen}, fitness=#{best[:fitness]}"
68   end
69   return best
70 end
71
72 max_generations = 200
73 pop_size = 30
74 num_children = 20
75 problem_size = 2
76 search_space = Array.new(problem_size) {|i| [-5, +5]}
77
78 best = search(max_generations, problem_size, search_space, pop_size, num_children)
79 puts "done! Solution: f=#{best[:fitness]}, s=#{best[:vector].inspect}"
```

Listing 4.4: Evolution Strategies algorithm in the Ruby Programming Language

4.5.8 References

Primary Sources

Evolution Strategies was developed by three students (Bienert, Rechenberg, Schwefel) at the Technical University in Berlin in 1964 in an effort to robotically optimize an aerodynamics design problem. The seminal work in Evolution Strategy was by Rechenberg's PhD thesis [190] that was later published as a book [189], both in German. Many technical reports and papers were published by Schwefel and Rechenberg, although the seminal paper published in English was by Klockgether and Schwefel on the two–phase nozzle design problem [134].

Learn More

Schwefel published his PhD dissertation [201] not long after Rechenberg that too was later published as a book [200] both in German. Schwefel's book was later translated into English was represents a classical reference for the technique [202]. Bäck, et al. provide a classical introduction to the technique, covering the history, development of the algorithm, and the steps that lead it to where it was in 1991 [8]. Beyer and Schwefel provide a contemporary introduction to the field that includes a detailed history of the approach, the developments and improvements since its inception, and an overview of the theoretical findings that have been made [27].

4.6 Differential Evolution

Differential Evolution, DE.

4.6.1 Taxonomy

Differential Evolution is a Stochastic Direct Search and Global Optimization algorithm, and is an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. It is related to sibling Evolutionary Algorithms such as the Genetic Algorithm and Evolutionary Programming, and Evolution Strategies, and shows some similarities to Particle Swarm Optimization.

4.6.2 Strategy

The Differential Evolution algorithm involves maintaining a population of candidate solutions subjected to iterations of recombination, evaluation, and selection. The recombination approach involves the creation of new candidate solution components based on the weighted difference between two randomly selected population members added to a third population member. This perturbs population members relative to the spread of the broader population. In conjunction with selection, the perturbation effect self-organizes the sampling of the problem space, bounding it to known areas of interest.

4.6.3 Procedure

Differential Evolution has a specialized nomenclature that describes the adopted configuration. This takes the form of DE/x/y/z, where x represents the solution to be perturbed (such a random or best). The y signifies the number of difference vectors used in the perturbation of x, where a difference vectors is the difference between two randomly selected although distinct members of the population. Finally, z signifies the recombination operator performed such as bin for binomial and exp for exponential.

Algorithm 19 provides a pseudo-code listing of the Differential Evolution algorithm for minimizing a cost function, specifically a DE/rand/1/bin configuration. Algorithm 20 provides a pseudo-code listing of the $NewSample$ function from the Differential Evolution algorithm.

4.6.4 Heuristics

- Differential evolution was designed for nonlinear, non-differentiable continuous function optimization.
- The weighting factor $F \in [0, 2]$ controls the amplification of differential variation, a value of 0.8 is suggested.
- the crossover weight $CR \in [0, 1]$ probabilistically controls the amount of recombination, a value of 0.9 is suggested.

Algorithm 19: Pseudo Code for the Differential Evolution algorithm.

```
Input: G, NP, F, CR
Output: S_best
Population ← InitializePopulation(G, NP);
EvaluateCost(Population);
S_best ← GetBestSolution(Population);
while ¬StopCondition() do
    NewPopulation ← 0;
    foreach P_i ∈ Population do
        S_i ← NewSample(P_i, Population, NP, F, CR);
        if Cost(S_i) ≤ Cost(P_i) then
            NewPopulation ← S_i;
        else
            NewPopulation ← P_i;
        end
    end
    Population ← NewPopulation;
    EvaluateCost(Population);
    S_best ← GetBestSolution(Population);
end
return S_best;
```

Algorithm 20: Pseudo Code for the NewSample function in the Differential Evolution algorithm.

Input: P_0, Population, NP, F, CR
Output: S

```
repeat
    P1 ← RandomMemeber(Population);
until P1 ≠ P0 ;
repeat
    P2 ← RandomMemeber(Population);
until P2 ≠ P0 ∨ P2 ≠ P1 ;
repeat
    P3 ← RandomMemeber(Population);
until P3 ≠ P0 ∨ P3 ≠ P1 ∨ P3 ≠ P2 ;
CutPoint ← RandomPosition(NP);
S ← 0;
for i to NP do
    if i ≡ CutPoint ∧ Rand() < CR then
        S_i ← P3_i + F × (P1_i- P2_i);
    else
        S_i ← P0_i;
    end
end
return S;
```

- The initial population of candidate solutions should be randomly generated from within the space of valid solutions.
- The popular configurations are DE/rand/1/* and DE/best/2/*.

4.6.5 Code Listing

Listing 4.5 provides an example of the Differential Evolution algorithm implemented in the Ruby Programming Language. The demonstration problem is an instance of a continuous function optimization that seeks $minf(x)$ where $f = \sum_{i=1}^{n} x_i^2$, $-5.0 \leq x_i \leq 5.0$ and $n = 3$. The optimal solution for this basin function is $(v_0, \ldots, v_{n-1}) = 0.0$. The algorithm is an implementation of Differential Evolution with the DE/rand/1/bin configuration proposed by Storn and Price [217].

```
def objective_function(vector)
  return vector.inject(0.0) {|sum, x| sum + (x ** 2.0)}
end

def random_vector(problem_size, search_space)
  return Array.new(problem_size) do |i|
    search_space[i][0] + ((search_space[i][1] - search_space[i][0]) * rand())
  end
end

def new_sample(p0, p1, p2, p3, f, cr, search_space)
  length = p0[:vector].length
  sample = {}
  sample[:vector] = []
  cut = rand(length-1) + 1
  length.times do |i|
    if (i==cut or rand() < cr)
      v = p3[:vector][i] + f * (p1[:vector][i] - p2[:vector][i])
      v = search_space[i][0] if v < search_space[i][0]
      v = search_space[i][1] if v > search_space[i][1]
      sample[:vector] << v
    else
      sample[:vector] << p0[:vector][i]
    end
  end
  return sample
end

def search(max_generations, np, search_space, g, f, cr)
  pop = Array.new(g) {|i| {:vector=>random_vector(np, search_space)} }
  pop.each{|c| c[:cost] = objective_function(c[:vector])}
  gen, best = 0, pop.sort{|x,y| x[:cost] <=> y[:cost]}.first
  max_generations.times do |gen|
    samples = []
    pop.each_with_index do |p0, i|
      p1 = p2 = p3 = -1
      p1 = rand(pop.length) until p1!=i
      p2 = rand(pop.length) until p2!=i and p2!=p1
```

```
      p3 = rand(pop.length) until p3!=i and p3!=p1 and p3!=p2
      samples << new_sample(p0, pop[p1], pop[p2], pop[p3], f, cr, search_space)
    end
    samples.each{|c| c[:cost] = objective_function(c[:vector])}
    nextgen = Array.new(g) do |i|
      (samples[i][:cost]<=pop[i][:cost]) ? samples[i] : pop[i]
    end
    pop = nextgen
    pop.sort{|x,y| x[:cost] <=> y[:cost]}
    best = pop.first if pop.first[:cost] < best[:cost]
    puts " > gen #{gen+1}, fitness=#{best[:cost]}"
  end
  return best
end


problem_size = 3
max_generations = 200
pop_size = 10*problem_size
weighting_factor = 0.8
crossover_factor = 0.9
search_space = Array.new(problem_size) {|i| [-5, +5]}

best = search(max_generations, problem_size, search_space, pop_size, weighting_factor,
    crossover_factor)
puts "done! Solution: f=#{best[:cost]}, s=#{best[:vector].inspect}"
```

Listing 4.5: Differential Evolution algorithm in the Ruby Programming Language

4.6.6 References

Primary Sources

The Differential Evolution algorithm was presented by Storn and Price in a technical report that considered DE1 and DE2 variants of the approach applied to a suite of continuous function optimization problems [215]. An early paper by Storn applied the approach to the optimization of an IIR-filter (Infinite Impulse Response) [213]. A second early paper applied the approach to a second suite of benchmark problem instances, adopting the contemporary nomenclature for describing the approach, including the DE/rand/1 and DE/best/2 variations [216]. The early work including technical reports and conference papers by Storn and Price culminated in a seminal journal article [217].

Learn More

A classical overview of Differential Evolution is presented by Price and Storn [184], and terse introduction to the approach for function optimization is presented by Storn [214]. A seminal extended description of the algorithm with sample applications was presented by Storn and Price as a book chapter [185]. Price, Storn, and Lampinen release a contemporary book dedicated to Differential Evolution including theory, benchmarks,

sample code and numerous application demonstrations [186]. Chakraborty also released a book considering extensions to the approach to address complexities such as rotation invariance and stopping criteria [39].

4.7 Grammatical Evolution

Grammatical Evolution, GE.

4.7.1 Taxonomy

Grammatical Evolution is a Global Optimization technique and an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. It may also be considered an algorithm for Automatic Programming. Grammatical Evolution is related to other Evolutionary Algorithms for evolving programs such as Genetic Programming, as well as the classical Genetic Algorithm that uses binary strings.

4.7.2 Inspiration

The Grammatical Evolution algorithm is inspired by the biological process used for generating a protein from genetic material as well as the broader genetic evolutionary process. The genome is comprised of DNA as a string of building blocks that are transcribed to RNA. RNA codons are in turn translated into sequences of amino acids and used in the protein. The resulting protein in its environment is the phenotype.

4.7.3 Metaphor

The phenotype is a computer program that is created from a binary string-based genome. The genome is decoded into a sequence of integers that are in turn mapped onto predefined rules that makeup the program. The mapping from genotype to the phenotype is many-to-many process that uses a wrapping feature. This is like the biological process observed in many bacteria, viruses, and mitochondria, where the same genetic material is used in the expression of different genes. The mapping adds robustness to the process both in the ability to adopt structure-agnostic genetic operators used during the evolutionary process on the sub-symbolic representation and the transcription of well-formed executable programs from the representation.

4.7.4 Strategy

The objective of Grammatical Evolution is to adapt an executable program to a problem specific objective function. This is achieved through an iterative process with surrogates of evolutionary mechanisms such as descent with variation, genetic mutation and recombination, and genetic transcription and gene expression. A population of programs are evolved in a sub-symbolic form as variable length binary strings and mapped to a symbolic and well-structured form as a context free grammar for execution.

4.7.5 Procedure

A grammar is defined in Backus Normal Form (BNF), which is a context free grammar expressed as a series of production rules comprised of terminals and non-terminals. A

variable-length binary string representation is used for the optimization process. Bits are read from the a candidate solutions genome in blocks of 8 and decoded to an integer (in the range between 0 and 2^{8-1}). If the end of the binary string is reached when reading integers, the reading process loops back to the start of the string, effectively creating a circular genome. The integers are mapped to expressions from the BNF until a complete syntactically correct expression is formed. This may not use a solutions entire genome, or use the decoded genome more than once given it's circular nature. Algorithm 21 provides a pseudo-code listing of the Grammatical Evolution algorithm for minimizing a cost function.

Algorithm 21: Pseudo Code for the Grammatical Evolution algorithm.

Input: Grammar, $Codon_{numbits}$ $Population_{size}$, $P_{crossover}$, $P_{mutation}$, P_{delete}, $P_{duplicate}$
Output: S_{best}

```
1  Population ← InitializePopulation(Population_size, Codon_numbits);
2  foreach S_i ∈ Population do
3      Si_integers ← Decode(Si_bitstring, Codon_numbits);
4      Si_program ← Map(Si_integers, Grammar);
5      Si_cost ← Execute(Si_program);
6  end
7  S_best ← GetBestSolution(Population);
8  while ¬StopCondition() do
9      Parents ← SelectParents(Population, Population_size);
10     Children ← 0;
11     foreach Parent_i, Parent_j ∈ Parents do
12         S_i ← Crossover(Parent_i, Parent_j, P_crossover);
13         Si_bitstring ← CodonDeletion(Si_bitstring, P_delete);
14         Si_bitstring ← CodonDuplication(Si_bitstring, P_duplicate);
15         Si_bitstring ← Mutate(Si_bitstring, P_mutation);
16         Children ← S_i;
17     end
18     foreach S_i ∈ Children do
19         Si_integers ← Decode(Si_bitstring, Codon_numbits);
20         Si_program ← Map(Si_integers, Grammar);
21         Si_cost ← Execute(Si_program);
22     end
23     S_best ← GetBestSolution(Children);
24     Population ← Replace(Population, Children);
25 end
26 return S_best;
```

4.7.6 Heuristics

- Grammatical Evolution was designed to optimize programs (such as mathematical equations) to specific cost functions.
- Classical genetic operators used by the Genetic Algorithm may be used in the Grammatical Evolution algorithm, such as point mutations and one-point crossover.
- Codon's (groups of bits mapped to an integer) are commonly fixed at 8-bits, proving a range of integers $\in [0, 2^{8-1}]$ that may be scaled to the range of rules using a modulo function.
- Additional genetic operators may be used with variable-length representations such as codon duplication (add to the end) and deletion.

4.7.7 Code Listing

Listing 4.6 provides an example of the Grammatical Evolution algorithm implemented in the Ruby Programming Language based on the version described by O'Neill and Ryan [173]. The demonstration problem is an instance of symbolic regression $f(x) = x^4 + x^3 + x^2 + x$, where $x \in [-1, 1]$. The grammar used in this problem is:

- Non-terminals: $N = \{expr, op, pre\_op\}$
- Terminals: $T = \{sin, cos, exp, log, +, -, /, *, x, 1.0\}$
- Expression (program): $S =< expr >$

The production rules for the grammar in BNF are:

- $< expr > ::= < expr >< op >< expr >, (< expr >< op >< expr >), < pre\_op > (< expr >), < var >$
- $< op > ::= +$, -, $\div$, $\times$
- $< pre\_op > ::=$ Sin, Cos, Exp, Log
- $< var > ::=$ x, 1.0

The algorithm uses point mutation and a codon-respecting one-point crossover operator. Binary tournament selection is used to determine the parent population's contribution to the subsequent generation. Binary strings are decoded to integers using the Binary Coded Decimal method. Candidate solutions are then mapped directly into executable ruby code and executed. A given candidate solution is evaluated by comparing its output against the target function and taking the sum of the absolute errors over a number of trials. The probabilities of point mutation, codon deletion, and codon duplication are hard coded as relative probabilities to each solution, although should be parameters of the algorithm. In this case they are heuristically defined as $\frac{1.0}{L}$, $\frac{0.5}{NC}$ and $\frac{1.0}{NC}$ respectively, where L is the total number of bits, and NC is the number of codons in a given candidate solution.

```ruby
def binary_tournament(population)
  s1, s2 = population[rand(population.size)], population[rand(population.size)]
  return (s1[:fitness] > s2[:fitness]) ? s1 : s2
end

def point_mutation(bitstring)
  rate = 1.0/bitstring.to_f
  child = ""
  bitstring.size.times do |i|
    bit = bitstring[i]
    child << ((rand()<rate) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

def one_point_crossover(parent1, parent2, p_crossover, codon_bits)
  return ""+parent1[:bitstring] if rand()>=p_crossover
  cut = rand([parent1.length, parent2.length].min/codon_bits)
  cut *= codon_bits
  p2length = parent2[:bitstring].length
  return parent1[:bitstring][0...cut]+parent2[:bitstring][cut...p2length]
end

def codon_duplication(bitstring, codon_bits)
  codons = bitstring.length/codon_bits
  return bitstring if rand() >= 1.0/codons.to_f
  return bitstring + bitstring[rand(codons)*codon_bits, codon_bits]
end

def codon_deletion(bitstring, codon_bits)
  codons = bitstring.length/codon_bits
  return bitstring if rand() >= 0.5/codons.to_f
  off = rand(codons)*codon_bits
  return bitstring[0...off] + bitstring[off+codon_bits...bitstring.length]
end

def reproduce(selected, population_size, p_crossover, codon_bits)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
    child[:bitstring] = one_point_crossover(p1, p2, p_crossover, codon_bits)
    child[:bitstring] = codon_deletion(child[:bitstring], codon_bits)
    child[:bitstring] = codon_duplication(child[:bitstring], codon_bits)
    child[:bitstring] = point_mutation(child[:bitstring])
    children << child
  end
  return children
end

def random_bitstring(num_bits)
  return (0...num_bits).inject(""){|s,i| s<<((rand<0.5) ? "1" : "0")}
end
```

```ruby

def decode_integers(bitstring, codon_bits)
  ints = []
  (bitstring.length/codon_bits).times do |off|
    codon = bitstring[off*codon_bits, codon_bits]
    sum, i = 0, 0
    codon.each_char {|x| sum+=((x=='1') ? 1 : 0) * (2 ** i);i+=1}
    ints << sum
  end
  return ints
end

def map(grammar, integers, max_depth)
  done, offset, depth = false, 0, 0
  symbolic_string = grammar["S"]
  begin
    done = true
    grammar.keys.each do |key|
      symbolic_string = symbolic_string.gsub(key) do |k|
        done = false
        set = (k=="EXP" and depth>=max_depth-1) ? grammar["VAR"] : grammar[k]
        integer = integers[offset].modulo(set.length)
        offset = (offset==integers.length-1) ? 0 : offset+1
        set[integer]
      end
    end
    depth += 1
  end until done
  return symbolic_string
end

def target_function(x)
  x**4.0 + x**3.0 + x**2.0 + x
end

def cost(program, bounds)
  errors = 0.0
  10.times do
    x = bounds[0] + ((bounds[1] - bounds[0]) * rand())
    expression = program.gsub("INPUT", x.to_s)
    target = target_function(x)
    begin score = eval(expression) rescue score = 0.0/0.0 end
    errors += (((score.nan? or score.infinite?) ? 0.0 : score) - target).abs
  end
  return errors
end

def evaluate(candidate, codon_bits, grammar, max_depth, bounds)
  candidate[:integers] = decode_integers(candidate[:bitstring], codon_bits)
  candidate[:program] = map(grammar, candidate[:integers], max_depth)
  candidate[:fitness] = cost(candidate[:program], bounds)
end

```

```ruby
def search(generations, pop_size, codon_bits, initial_bits, p_crossover, grammar,
    max_depth, bounds)
  pop = Array.new(pop_size) {|i| {:bitstring=>random_bitstring(initial_bits)}}
  pop.each{|c| evaluate(c,codon_bits, grammar, max_depth, bounds)}
  gen, best = 0, pop.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  generations.times do |gen|
    selected = Array.new(pop_size){|i| binary_tournament(pop)}
    children = reproduce(selected, pop_size, p_crossover,codon_bits)
    children.each{|c| evaluate(c,codon_bits, grammar, max_depth, bounds)}
    children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
    best = children.first if children.first[:fitness] >= best[:fitness]
    pop = children
    puts " > gen=#{gen}, f=#{best[:fitness]},
        codons=#{best[:bitstring].length/codon_bits}, s=#{best[:bitstring]}"
  end
  return best
end

grammar = {"S"=>"EXP",
  "EXP"=>[" EXP BINARY EXP ", " (EXP BINARY EXP) ", " UNIARY(EXP) ", " VAR "],
  "BINARY"=>["+", "-", "/", "*" ],
  "UNIARY"=>["Math.sin", "Math.cos", "Math.exp", "Math.log"],
  "VAR"=>["INPUT", "1.0"]}
max_depth = 7
bounds = [-1, +1]
generations = 100
pop_size = 100
codon_bits = 8
initial_bits = 10*codon_bits
p_crossover = 0.30

best = search(generations, pop_size, codon_bits, initial_bits, p_crossover, grammar,
    max_depth, bounds)
puts "done! Solution: f=#{best[:fitness]}, s=#{best[:program]}"
```

Listing 4.6: Grammatical Evolution algorithm in the Ruby Programming Language

4.7.8 References

Primary Sources

Grammatical Evolution was proposed by Ryan, Collins and O'Neill in a seminal conference paper that applied the approach to a symbolic regression problem [196]. The approach was born out of the desire for syntax preservation while evolving programs using the Genetic Programming algorithm. This seminal work was followed by application papers for a symbolic integration problem [169, 170] and solving trigonometric identities [197].

Learn More

O'Neill and Ryan provide a high-level introduction to Grammatical Evolution and early demonstration applications [171]. The same authors provide a through introduction to the technique and overview of the state of the field [173]. O'Neill and Ryan present a seminal reference for Grammatical Evolution in their book [172]. A second more recent book considers extensions to the approach improving its capability on dynamic problems [52].

4.8 Gene Expression Programming

Gene Expression Programming, GEP.

4.8.1 Taxonomy

Gene Expression Programming is a Global Optimization algorithm and an Automatic Programming technique, and it is an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. It is a sibling of other Evolutionary Algorithms such as a the Genetic Algorithm as well as other Evolutionary Automatic Programming techniques such as Genetic Programming and Grammatical Evolution.

4.8.2 Inspiration

Gene Expression Programming is inspired by the replication and expression of the DNA molecule, specifically at the gene level. The expression of a gene involves the transcription of its DNA to RNA which in turn forms amino acids that make up proteins in the phenotype of an organism. The DNA building blocks are subjected to mechanisms of variation (mutations such as coping errors) as well as recombination during sexual reproduction.

4.8.3 Metaphor

Gene Expression Programming uses a linear genome as the basis for genetic operators such as mutation, recombination, inversion, and transposition. The genome is comprised of chromosomes and each chromosome is comprised of genes that are translated into an expression tree to solve a given problem. The robust gene definition means that genetic operators can be applied to the sub-symbolic representation without concern for the structure of the resultant gene expression, providing separation of genotype and phenotype.

4.8.4 Strategy

The objective of the Gene Expression Programming algorithm is to improve the adaptive fit of an expressed program in the context of a problem specific cost function. This is achieved through the use of an evolutionary process that operates on a sub-symbolic representation of candidate solutions using surrogates for the processes (descent with modification) and mechanisms (genetic recombination, mutation, inversion, transposition, and gene expression) of evolution.

4.8.5 Procedure

A candidate solution is represented as a linear string of symbols called Karva notation or a K-expression, where each symbol maps to a function or terminal node. The linear representation is mapped to an expression tree in a breadth-first manner. A K-expression

is fixed length and is comprised of one or more sub-expressions (genes), which are also defined with a fixed length. A gene is comprised of two sections, a head which may contain any function or terminal symbols, and a tail section that may only contain terminal symbols. Each gene will always translate to a syntactically correct expression tree, where the tail portion of the gene provides a genetic buffer which ensures closure of the genes expression.

Algorithm 22 provides a pseudo-code listing of the Gene Expression Programming algorithm for minimizing a cost function.

Algorithm 22: Pseudo Code for the Gene Expression Programming algorithm.

Input: Grammar, $Population_{size}$, $Head_{length}$, $Tail_{length}$, $P_{crossover}$, $P_{mutation}$
Output: S_{best}

```
Population ← InitializePopulation(Population_size, Grammar, Head_length,
Tail_length);
foreach S_i ∈ Population do
    Si_program ← DecodeBreadthFirst(Si_genome, Grammar);
    Si_cost ← Execute(Si_program);
end
S_best ← GetBestSolution(Population);
while ¬StopCondition() do
    Parents ← SelectParents(Population, Population_size);
    Children ← 0;
    foreach Parent_1, Parent_2 ∈ Parents do
        Si_genome ← Crossover(Parent_1, Parent_2, P_crossover);
        Si_genome ← Mutate(Si_genome, P_mutation);
        Children ← S_i;
    end
    foreach S_i ∈ Population do
        Si_program ← DecodeBreadthFirst(Si_genome, Grammar);
        Si_cost ← Execute(Si_program);
    end
    S_best ← GetBestSolution(Children);
    Population ← Replace(Population, Children);
end
return S_best;
```

4.8.6 Heuristics

- The length of a chromosome is defined by the number of genes, where a gene length is defined by $h + t$. The h is a user defined parameter (such as 10), where as t is defined as $t = h(n - 1) + 1$. The n represents the maximum arity of functional nodes in the expression (such as 2 if the arithmetic functions $\times, \div, -, +$ are used).

- The mutation operate substituted expressions along the genome, although must respect the gene rules such that function and terminal nodes are mutated in the head of genes, whereas only terminal nodes are substituted in the tail of genes.
- Crossover occurs between two selected parents from the population and can occur based on a one-point cross, two point cross, or finally a gene-based approach were genes are selected from the parents with uniform probability.
- An inversion operator may be used with a low probability that reverses a small sequence of symbols (1-3) within a section of a gene (tail or head).
- A transposition operator may be used that has a number of different modes, including: duplicate a small sequences (1-3) from somewhere on a gene to the head, small sequences on a gene to the root of the gene, and moving of entire genes on the chromosome. In the case of intra-gene transpositions, the sequence in the head of the gene is moved down to accommodate the copied sequence and the length of the head is truncated to maintain consistent gene sizes.
- A '?' is included in the terminal set that represents a numeric constant from an array that are evolved on the end of the genome. The constants are read from the end of the genome and are substituted for '?' as the expression tree is created (in breadth first order). Finally the numeric constants are used as array indices in yet another chromosome of numerical values which are substituted into the expression tree.
- Mutation is low (such as $\frac{1}{L}$), selection can be any of the classical approaches (such as roulette wheel or tournament), and crossover rates are typically high (0.7 of offspring)
- Use multiple sub-expressions linked together on hard problems when one gene does not get much progress. The sub-expressions are linked using link expressions which are function nodes that are either statically defined (such as a conjunction) or evolved on the genome with the genes.

4.8.7 Code Listing

Listing 4.7 provides an example of the Gene Expression Programming algorithm implemented in the Ruby Programming Language based on the seminal version proposed by Ferreira [64]. The demonstration problem is an instance of symbolic regression $f(x) = x^4 + x^3 + x^2 + x$, where $x \in [-1, 1]$. The grammar used in this problem is: Functions: $F = \{+, -, \div, \times, \}$ and Terminals: $T = \{x\}$. The algorithm uses binary tournament selection, uniform crossover and point mutations. The K-expression is decoded to an expression tree in a breadth-first manner, which is then parsed depth first as an ruby expression string for display and direct evaluation.

```
1 def binary_tournament(population)
2   s1, s2 = population[rand(population.size)], population[rand(population.size)]
```

```
  return (s1[:fitness] > s2[:fitness]) ? s1 : s2
end

def point_mutation(grammar, genome, p_mutation, head_length)
  child, i = "", 0
  genome.each_char do |v|
    if rand()<p_mutation
      if (i<head_length)
        child << grammar["FUNC"][rand(grammar["FUNC"].length)]
      else
        child << grammar["TERM"][rand(grammar["TERM"].length)]
      end
    else
      child << v
    end
    i += 1
  end
  return child
end

def uniform_crossover(parent1, parent2, p_crossover)
  return ""+parent1 if rand()>=p_crossover
  child = ""
  parent1.length.times do |i|
    child << ((rand()<0.5) ? parent1[i] : parent2[i])
  end
  return child
end

def reproduce(grammar, selected, pop_size, p_crossover, p_mutation, head_length)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
    child[:genome] = uniform_crossover(p1[:genome], p2[:genome], p_crossover)
    child[:genome] = point_mutation(grammar, child[:genome], p_mutation, head_length)
    children << child
  end
  return children
end

def random_genome(grammar, head_length, tail_length)
  s = ""
  head_length.times { s<<grammar["FUNC"][rand(grammar["FUNC"].length)]}
  tail_length.times { s<<grammar["TERM"][rand(grammar["TERM"].length)]}
  return s
end

def target_function(x)
  x**4.0 + x**3.0 + x**2.0 + x
end

def cost(program, bounds)
```

```
  errors = 0.0
  10.times do
    x = bounds[0] + ((bounds[1] - bounds[0]) * rand())
    expression = program.gsub("x", x.to_s)
    target = target_function(x)
    begin score = eval(expression) rescue score = 0.0/0.0 end
    errors += (((score.nan? or score.infinite?) ? 0.0 : score) - target).abs
  end
  return errors
end

def breadth_first_mapping(genome, grammar)
  off, queue = 0, Array.new
  root = {}
  root[:node] = genome[off].chr;off+=1
  queue.push(root)
  while !queue.empty? do
    current = queue.shift
    if grammar["FUNC"].include?(current[:node])
      current[:left] = {}
      current[:left][:node] = genome[off].chr;off+=1
      queue.push(current[:left])
      current[:right] = {}
      current[:right][:node] = genome[off].chr;off+=1
      queue.push(current[:right])
    end
  end
  return root
end

def tree_to_string(exp)
  return exp[:node] if (exp[:left].nil? and exp[:right].nil?)
  left = tree_to_string(exp[:left])
  right = tree_to_string(exp[:right])
  return "(#{left} #{exp[:node]} #{right})"
end

def evaluate(candidate, grammar, bounds)
  candidate[:expression] = breadth_first_mapping(candidate[:genome], grammar)
  candidate[:program] = tree_to_string(candidate[:expression])
  candidate[:fitness] = cost(candidate[:program], bounds)
end

def search(grammar, bounds, head_length, tail_length, generations, pop_size,
    p_crossover, p_mutation)
  pop = Array.new(pop_size) do
    {:genome=>random_genome(grammar, head_length, tail_length)}
  end
  pop.each{|c| evaluate(c, grammar, bounds)}
  gen, best = 0, pop.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  generations.times do |gen|
    selected = Array.new(pop){|i| binary_tournament(pop)}
    children = reproduce(grammar, selected, pop_size, p_crossover, p_mutation,
```

```
        head_length)
    children.each{|c| evaluate(c, grammar, bounds)}
    children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
    best = children.first if children.first[:fitness] >= best[:fitness]
    pop = children
    gen += 1
    puts " > gen=#{gen}, f=#{best[:fitness]}, g=#{best[:genome]}"
  end
  return best
end

grammar = {"FUNC"=>["+","-","*","/"], "TERM"=>["x"]}
bounds = [-1, 1]
head_length = 24
tail_length = head_length * (2-1) + 1
generations = 150
pop_size = 100
p_crossover = 0.70
p_mutation = 2.0/(head_length+tail_length).to_f

best = search(grammar, bounds, head_length, tail_length, generations, pop_size,
    p_crossover, p_mutation)
puts "done! Solution: f=#{best[:fitness]}, g=#{best[:genome]}, b=#{best[:program]}"
```

Listing 4.7: Gene Expression Programming algorithm in the Ruby Programming Language

4.8.8 References

Primary Sources

The Gene Expression Programming algorithm was proposed by Ferreira in a paper that detailed the approach, provided a careful walkthrough of the process and operators and demonstrated the the algorithm on a number of benchmark problem instances such as symbolic regression [64].

Learn More

Ferreira provided an early and detailed introduction and overview of the approach as book chapter, providing a step-by-step walkthrough of the procedure and sample applications [65]. A similar more contemporary and detailed introduction is provided in a second book chapter [66]. Ferreira published a book on the approach in 2002 covering background, the algorithm, and demonstration applications which is now in its second edition [67].

4.9 Learning Classifier System

Learning Classifier System, LCS.

4.9.1 Taxonomy

The Learning Classifier System algorithm is both an instance of an Evolutionary Algorithm from the field of Evolutionary Computation and an instance of a Reinforcement Learning algorithm from Machine Learning. The Learning Classifier System is a theoretical system with a number of implementations. Two streams of classifier are the Pittsburgh-style that seeks to optimize whole classifier, and the Michigan-style that optimize responsive rulesets. The Michigan-style Learning Classifier is the most common and is comprised of two versions: the ZCS (zeroth-level classifier system) and the XCS (accuracy-based classifier system).

4.9.2 Strategy

The objective of the Learning Classifier System algorithm is to optimize payoff based on exposure to stimuli from a problem-specific environment. This is achieved by managing credit assignment for those rules that prove useful and searching for new rules and new variations on existing rules using an evolutionary process.

4.9.3 Procedure

The actors of the system include detectors, messages, effectors, feedback, and classifiers. Detectors are used by the system to perceive the state of the environment. Messages are the discrete information packets passed from the detectors into the system. The system performs information processing on messages, and messages may directly result in actions in the environment. Effectors control the actions of the system on and within the environment. In addition to the system actively perceiving via its detections, it may also receive directed feedback from the environment (payoff). Classifiers are condition-action rules that provides a filter for messages. If a message satisfies the conditional part of the classifier, the action of the classier triggers. Rules act as message processors. Messages are defined at a fixed length using a binary alphabet. A classifier is defined as a binary string with a ternary alphabet of $1, 0, \#$, where the $\#$ represents do not care (matching both a 1 or 0).

The processing loop for the Learning Classifier system is as follows: i) Messages from the environment are placed on the message list. ii) The conditions of each classifier are checked to see if they are satisfied by at least one message in the message list. iii) All classifiers that are satisfied participate in a competition, those that win post their action to the message list. iv) All messages directed to the effectors are executed (causing actions in the environment). v) All messages on the message list from the previous cycle are deleted (messages persist for a single cycle). The algorithm may be described in terms of the main processing loop and two sub-algorithms: a reinforcement learning

algorithm such as the bucket brigade algorithm or Q-learning, and a genetic algorithm for optimization of the system. Algorithm 23 provides a pseudo-code listing of the high-level processing loop of the Learning Classifier System, specifically the XCS as described by Butz and Wilson [37].

Algorithm 23: Pseudo Code for the Learning Classifier System algorithm.

```
Input: env
Output: Population
env ← InitializeEnvironment(env);
Population ← InitializePopulation();
ActionSet_{t-1} ← 0;
Input_{t-1} ← 0;
Reward_{t-1} ← 0;
while ¬StopCondition() do
    Input_t ← env;
    Matchset ← GenerateMatchSet(Population, Input_t);
    Prediction ← GeneratePrediction(Matchset);
    Action ← SelectionAction(Prediction);
    ActionSet_t ← GenerateActionSet(Action, Matchset);
    Reward_t ← ExecuteAction(Action, env);
    if ActionSet_{t-1} ≠ 0 then
        Payoff_t ← CalculatePayoff(Reward_{t-1}, Prediction);
        PerformLearning(ActionSet_{t-1}, Payoff_t, Population);
        RunGeneticAlgorithm(ActionSet_{t-1}, Input_{t-1}, Population);
    end
    if LastStepOfTask(env, Action) then
        Payoff_t ← Reward_t;
        PerformLearning(ActionSet_t, Payoff_t, Population);
        RunGeneticAlgorithm(ActionSet_t, Input_t, Population);
        ActionSet_{t-1} ← 0;
    else
        ActionSet_{t-1} ← ActionSet_t;
        Input_{t-1} ← Input_t;
        Reward_{t-1} ← Reward_t;
    end
end
```

4.9.4 Heuristics

The majority of the heuristics in this section are specific to the XCS Learning Classifier System as described by Butz and Wilson [37].

- Learning Classifier Systems are suited for problems with the following characteris-

tics: perpetually novel events with large amounts of noise, continual, and real-time requirements for action, implicitly or inexactly defined goals, and sparse payoff or reinforcement obtainable only through long sequences of tasks.

- The learning rate β for a classifiers expected payoff, error and fitness are typically in the range $\in [0.1, 0.2]$.
- The frequency of running the genetic algorithm θ_{GA} should be in the range $\in [25, 50]$.
- The discount factor used in multi-step programs γ are typically in the around 0.71.
- The minimum error for whereby classifiers are considered to have equal accuracy ϵ_0 are typically 10% of the maximum reward.
- The probability of crossover in the genetic algorithm χ are typically in the range $\in [0.5, 1.0]$.
- The probability of mutating a single position in a classifier in the genetic algorithm μ is typically in the range $\in [0.01, 0.05]$.
- The experience threshold during classifier deletion θ_{del} is typically about 20.
- The experience threshold for a classifier during subsumption θ_{sub} is typically around 20.
- The initial values for a classifiers expected payoff p_1, error ϵ_1, and fitness f_1 are typically small and close to zero.
- The probability of selecting a random action for the purposes of exploration p_{exp} is typically close to 0.5.
- The minimum number of different actions that must be specified in a match set θ_{mna} is usually the total number of possible actions in the environment for the input.
- Subsumption should be used on problem domains that are known contain well defined rules for mapping inputs to outputs.

4.9.5 Code Listing

Listing 4.8 provides an example of the Learning Classifier System algorithm implemented in the Ruby Programming Language. The problem is an instance of a Boolean multiplexer called the 6-multiplexer. It can be described as a classification problem, where each of the 2^6 patterns of bits is associated with a boolean class $\{1, 0\}$. For this problem instance, the first two bits may be decoded as an address into the remaining four bits that specify the class (for example in 100010, '10' decode to the index of '2' in the remaining 4 bits making the class '1'). In propositional logic this problem instance may

be described as $F = (\neg x_0)(\neg x_1)x_2 + (\neg x_0)x_1x_3 + x_0(\neg x_1)x_4 + x_0x_1x_5$. The algorithm is an instance of XCS based on the description provided by Butz and Wilson [37] with the parameters based on the application of XCS to Boolean multiplexer problems by Wilson [238, 239]. The population is grown as needed, and subsumption which would be appropriate for the Boolean multiplexer problem was not used for brevity. The multiplexer problem is a single step problem, so the complexities of delayed payoff are not required. A number of parameters were hard coded to recommended values, specifically: $\alpha = 0.1, \upsilon = 5, \delta = 0.1$ and $P_{\#} = \frac{1}{3}$.

```
def new_classifier(condition, action, gen)
  other = {}
  other[:condition], other[:action], other[:lasttime] = condition, action, gen
  other[:prediction], other[:error], other[:fitness] = 0.00001, 0.00001, 0.00001
  other[:experience], other[:setsize], other[:num] = 0.0, 1.0, 1.0
  return other
end

def copy_classifier(parent)
  copy = {}
  parent.keys.each {|k| copy[k] = (parent[k].kind_of? String) ? ""+parent[k] :
      parent[k]}
  copy[:num] = 1
  copy[:experience] = 0.0
  return copy
end

def generate_problem_string(length)
  return (0...length).inject(""){|s,i| s+((rand<0.5) ? "1" : "0")}
end

def neg(bit)
  return (bit==1) ? 0 : 1
end

def target_function(s)
  ints = Array.new(s.length){|i| s[i].chr.to_i}
  x0,x1,x2,x3,x4,x5 = ints
  return neg(x0)*neg(x1)*x2 + neg(x0)*x1*x3 + x0*neg(x1)*x4 + x0*x1*x5
end

def calculate_deletion_vote(classifier, pop, del_thresh)
  vote = classifier[:setsize] * classifier[:num]
  avg_fit = pop.inject(0.0){|s,c| s+c[:fitness]}/pop.inject(0.0){|s,c| s+c[:num]}
  derated = classifier[:fitness] / classifier[:num]
  if classifier[:experience] > del_thresh and derated < 0.1 * avg_fit
    vote *= avg_fit / derated
  end
  return vote
end

def delete_from_pop(pop, pop_size, del_thresh)
  total = pop.inject(0) {|s,c| s+c[:num]}
```

```
  return if total < pop_size
  pop.each {|c| c[:dvote] = calculate_deletion_vote(c, pop, del_thresh)}
  vote_sum = pop.inject(0.0) {|s,c| s+c[:dvote]}
  point = rand() * vote_sum
  vote_sum, index = 0.0, 0
  pop.each_with_index do |c,i|
    vote_sum += c[:dvote]
    if vote_sum > point
      index = i
      break
    end
  end
  if pop[index][:num] > 1
    pop[index][:num] -= 1
  else
    pop.delete_at(index)
  end
end

def generate_random_classifier(input, actions, gen)
  condition = ""
  input.each_char {|s| condition << ((rand<1.0/3.0) ? '#' : s)}
  action = actions[rand(actions.length)]
  return new_classifier(condition, action, gen)
end

def does_match(input, condition)
  i = 0
  condition.each_char do |c|
    return false if c!='#' and c!=input[i].chr
    i += 1
  end
  return true
end

def get_actions(pop)
  return [] if pop.empty?
  set = {}
  pop.each do |classifier|
    key = classifier[:action]
    set[key] = 0 if set[key].nil?
    set[key] += 1
  end
  return set.keys
end

def generate_match_set(input, pop, all_actions, gen, pop_size, del_thresh)
  match_set = pop.select{|c| does_match(input, c[:condition])}
  actions = get_actions(match_set)
  while actions.length < all_actions.length do
    remaining = all_actions - actions
    classifier = generate_random_classifier(input, remaining, gen)
    pop << classifier
```

```
96      match_set << classifier
97      delete_from_pop(pop, pop_size, del_thresh)
98      actions << classifier[:action]
99    end
100   return match_set
101 end
102
103 def generate_prediction(input, match_set)
104   prediction = {}
105   match_set.each do |classifier|
106     key = classifier[:action]
107     prediction[key] = {:sum=>0.0,:count=>0.0,:weight=>0.0} if prediction[key].nil?
108     prediction[key][:sum] += classifier[:prediction]*classifier[:fitness]
109     prediction[key][:count] += classifier[:fitness]
110   end
111   prediction.keys.each do |key|
112     prediction[key][:weight]=prediction[key][:sum]/prediction[key][:count]
113   end
114   return prediction
115 end
116
117 def select_action(prediction_array, p_explore)
118   keys = prediction_array.keys
119   return true, keys[rand(keys.length)] if rand() < p_explore
120   keys.sort!{|x,y| prediction_array[y][:weight]<=>prediction_array[x][:weight]}
121   return false, keys.first
122 end
123
124 def update_set(action_set, payoff, l_rate)
125   action_set.each do |c|
126     c[:experience] += 1.0
127     pdiff = payoff - c[:prediction]
128     c[:prediction] += (c[:experience]<1.0/l_rate) ? pdiff/c[:experience] : l_rate*pdiff
129     diff = pdiff.abs - c[:error]
130     c[:error] += (c[:experience]<1.0/l_rate) ? diff/c[:experience] : l_rate*diff
131     sum = action_set.inject(0.0) {|s,other| s+other[:num]-c[:setsize]}
132     c[:setsize] += (c[:experience]<1.0/l_rate) ? sum/c[:experience] : l_rate*sum
133   end
134 end
135
136 def update_fitness(action_set, min_error, l_rate)
137   sum = 0.0
138   accuracy = Array.new(action_set.length)
139   action_set.each_with_index do |c,i|
140     accuracy[i] = (c[:error]<min_error) ? 1.0 : 0.1*(c[:error]/min_error)**-5.0
141     sum += accuracy[i] * c[:num]
142   end
143   action_set.each_with_index do |c,i|
144     c[:fitness] += l_rate * (accuracy[i] * c[:num] / sum - c[:fitness])
145   end
146 end
147
148 def can_run_genetic_algorithm(action_set, gen, ga_freq)
```

```
149   total = action_set.inject(0.0) {|s,c| s+c[:lasttime]*c[:num]}
150   sum = action_set.inject(0.0) {|s,c| s+c[:num]}
151   if gen - (total/sum) > ga_freq
152     return true
153   end
154   return false
155 end
156 
157 def select_parent(pop)
158   sum = pop.inject(0.0) {|s,c| s+c[:fitness]}
159   point = rand() * sum
160   sum = 0
161   pop.each do |c|
162     sum += c[:fitness]
163     return c if sum > point
164   end
165 end
166 
167 def mutation(classifier, p_mut, action_set, input)
168   classifier[:condition].length.times do |i|
169     if rand() < p_mut
170       if classifier[:condition][i].chr == '#'
171         classifier[:condition][i] = input[i]
172       else
173         classifier[:condition][i] = '#'
174       end
175     end
176   end
177   if rand() < p_mut
178     new_action = nil
179     begin
180       new_action = action_set[rand(action_set.length)]
181     end until new_action != classifier[:action]
182     classifier[:action] = new_action
183   end
184 end
185 
186 def uniform_crossover(string1, string2)
187   rs = ""
188   string1.length.times do |i|
189     rs << ((rand()<0.5) ? string1[i] : string2[i])
190   end
191   return rs
192 end
193 
194 def insert_in_pop(classifier, pop)
195   pop.each do |c|
196     if classifier[:condition]==c[:condition] and classifier[:action]==c[:action]
197       c[:num] += 1
198       return
199     end
200   end
201   pop << classifier
```

```
202 end
203
204 def crossover(c1, c2, p1, p2)
205   c1[:condition] = uniform_crossover(p1[:condition], p2[:condition])
206   c2[:condition] = uniform_crossover(p1[:condition], p2[:condition])
207   c1[:prediction] = (p1[:prediction]+p2[:prediction])/2.0
208   c1[:error] = 0.25*(p1[:error]+p2[:error])/2.0
209   c1[:fitness] = 0.1*(p1[:fitness]+p2[:fitness])/2.0
210   c2[:prediction] = c1[:prediction]
211   c2[:error] = c1[:error]
212   c2[:fitness] = c1[:fitness]
213 end
214
215 def run_genetic_algorithm(all_actions, pop, action_set, input, gen, p_cross, p_mut,
        pop_size, del_thresh)
216   p1, p2 = select_parent(action_set), select_parent(action_set)
217   c1, c2 = copy_classifier(p1), copy_classifier(p2)
218   crossover(c1, c2, p1, p2) if rand() < p_cross
219   [c1,c2].each do |c|
220     mutation(c, p_mut, all_actions, input)
221     insert_in_pop(c, pop)
222     delete_from_pop(pop, pop_size, del_thresh)
223   end
224 end
225
226 def search(length, pop_size, max_gens, all_actions, p_explore, l_rate, min_error,
        ga_freq, p_cross, p_mut, del_thresh)
227   pop, abs = [], 0
228   max_gens.times do |gen|
229     input = generate_problem_string(length)
230     match_set = generate_match_set(input, pop, all_actions, gen, pop_size, del_thresh)
231     prediction_array = generate_prediction(input, match_set)
232     explore, action = select_action(prediction_array, p_explore)
233     action_set = match_set.select{|c| c[:action]==action}
234     expected = target_function(input)
235     payoff = ((expected-action.to_i)==0) ? 300.0 : 1.0
236     abs += (expected - action.to_i).abs.to_f
237     update_set(action_set, payoff, l_rate)
238     update_fitness(action_set, min_error, l_rate)
239     if can_run_genetic_algorithm(action_set, gen, ga_freq)
240       action_set.each {|c| c[:lasttime] = gen}
241       run_genetic_algorithm(all_actions, pop, action_set, input, gen, p_cross, p_mut,
            pop_size, del_thresh)
242     end
243     if (gen+1).modulo(50)==0
244       puts " >gen=#{gen+1} classifiers=#{pop.size}, error=#{abs.to_i}/50
            (#{(abs/50*100)}%)"
245       abs = 0
246     end
247   end
248   return pop
249 end
250
```

```
max_gens, length, pop_size = 5000, 6, 150
all_actions = ['0', '1']
l_rate, min_error = 0.2, 0.01
p_explore, p_cross, p_mut = 0.10, 0.80, 0.04
ga_freq, del_thresh = 50, 20

pop = search(length, pop_size, max_gens, all_actions, p_explore, l_rate, min_error,
    ga_freq, p_cross, p_mut, del_thresh)
puts "done! Solution: classifiers=#{pop.size}"
```

Listing 4.8: Learning Classifier System algorithm in the Ruby Programming Language

4.9.6 References

Primary Sources

Early ideas on the theory of Learning Classifier Systems were proposed by Holland [115, 119], culminating in a standardized presentation a few years later [116]. A number of implementations of the theoretical system were investigated, although a taxonomy of the two main streams was proposed by De Jong [130]: 1) Pittsburgh-style proposed by Smith [208, 207] and 2) Holland-style or Michigan-style Learning classifiers that are further comprised of the Zeroth-level classifier (ZCS) [237] and the accuracy-based classifier (XCS) [238].

Learn More

Booker, Goldberg, and Holland provide a classical introduction to Learning Classifier Systems including an overview of the state of the field and the algorithm in detail [31]. Wilson and Goldberg also provide a classical introduction and review of the approach, although take a more critical stance [240]. Holmes, et al. provide a contemporary review of the field focusing both on the approach and application areas to which the approach has been demonstrated successfully [122]. Lanzi, Stolzmann, and Wilson provide a seminal book in the field as a collection of papers covering the basics, advanced topics, and demonstration applications. A particular highlight from this book is the first section that provides a concise description of Learning Classifier Systems by many leaders and major contributors to the field [118], providing rare insight. Another paper from this book by Lanzi and Riolo provides a detailed review of the development of the approach as it matured throughout the 1990s. Bull and Kovacs a second book introductory book to the field focusing on the theory of the approach and its practical application [34].

4.10 Non-dominated Sorting Genetic Algorithm

Non-dominated Sorting Genetic Algorithm, Nondominated Sorting Genetic Algorithm, Fast Elitist Non-dominated Sorting Genetic Algorithm, NSGA, NSGA-II, NSGAII.

4.10.1 Taxonomy

The Non-dominated Sorting Genetic Algorithm is a Multiple Objective Optimization (MOO) algorithm and is an instance of an Evolutionary Algorithm (EA) from the field of Evolutionary Computation (EC). NSGA is an extension of the Genetic Algorithm (GA) for multiple objective function optimization. It is related to other Evolutionary Multiple Objective Optimization Algorithms (EMOO) (or Multiple Objective Evolutionary Algorithms MOEA) such as the Vector-Evaluated Genetic Algorithm (VEGA), Strength Pareto Evolutionary Algorithm (SPEA), and Pareto Archived Evolution Strategy (PAES). There are two versions of the algorithm, the classical NSGA and the updated and currently canonical form NSGA-II.

4.10.2 Strategy

The objective of the NSGA algorithm is to improve the adaptive fit of a population of candidate solutions to a Pareto front constrained by a set of objective functions. The algorithm uses an evolutionary process with surrogates for evolutionary operators including selection, genetic crossover, and genetic mutation. The population is sorted into a hierarchy of sub-populations based on the ordering of Pareto dominance. Similarity between members of each sub-group is evaluated on the Pareto front, and the resulting groups and similarity measures are used to promote a diverse front of non-dominated solutions.

4.10.3 Procedure

Algorithm 24 provides a pseudo-code listing of the Non-dominated Sorting Genetic Algorithm II (NSGA-II) for minimizing a cost function. The `SortByRankAndDistance` function orders the population into a hierarchy of non-dominated Pareto fronts. The `CrowdingDistanceAssignment` calculates the average distance between members of each front on the front itself. Refer to Deb et al. for a clear presentation of the pseudo code and explanation of these functions [51]. The `CrossoverAndMutation` function performs the classical crossover and mutation genetic operators of the Genetic Algorithm. Both the `SelectParentsByRankAndDistance` and `SortByRankAndDistance` functions discriminate members of the population first by rank (order of dominated precedence of the front to which the solution belongs) and then distance within the front (calculated by `CrowdingDistanceAssignment`).

Algorithm 24: Pseudo Code for the Non-dominated Sorting Genetic Algorithm II.

Input: $Population_{size}$, ProblemSize, $P_{crossover}$, $P_{mutation}$
Output: S_{best}

```
Population ← InitializePopulation(Population_size, ProblemSize);
EvaluateAgainstObjectiveFunctions(Population);
FastNondominatedSort(Population);
Selected ← SelectParentsByRank(Population, Population_size);
Children ← CrossoverAndMutation(Selected, P_crossover, P_mutation);
while ¬StopCondition() do
    EvaluateAgainstObjectiveFunctions(Children);
    Union ← Merge(Population, Children);
    Fronts ← FastNondominatedSort(Union);
    Parents ← 0;
    Front_L ← 0;
    foreach Front_i ∈ Fronts do
        CrowdingDistanceAssignment(Front_i);
        if Size(Parents)+Size(Front_i) > Population_size then
            Front_L ← i;
            Break();
        else
            Parents ← Merge(Parents, Front_i);
        end
    end
    if Size(Parents)<Population_size then
        Front_L ← SortByRankAndDistance(Front_L);
        for P_1 to P_{Population_size−Size(LastFront)} do
            Parents ← Pi;
        end
    end
    Selected ← SelectParentsByRankAndDistance(Parents, Population_size);
    Population ← Children;
    Children ← CrossoverAndMutation(Selected, P_crossover, P_mutation);
end
return Children;
```

4.10.4 Heuristics

- NSGA was designed for and is suited to continuous function multiple objective optimization problem instances.
- A binary representation can be used in conjunction with classical genetic operators such as one-point crossover and point mutation.
- A real-valued representation is recommended for continuous function optimization problems, in turn requiring representation specific genetic operators such as Simulated Binary Crossover (SBX) and polynomial mutation [48].

4.10.5 Code Listing

Listing 4.9 provides an example of the Non-dominated Sorting Genetic Algorithm II (NSGA-II) implemented in the Ruby Programming Language. The demonstration problem is an instance of continuous multiple objective function optimization called SCH (problem one in [51]). The problem seeks the minimum of two functions: $f1 = \sum_{i=1}^{n} x_i^2$ and $f2 = \sum_{i=1}^{n} (x_i - 2)^2$, $-10^3 \leq x_i \leq 10^3$ and $n = 1$. The optimal solution for this function are $x \in [0, 2]$. The algorithm is an implementation of NSGA-II based on the presentation by Deb, et al. [51]. The algorithm uses a binary string representation (16 bits per objective function parameter) that is decoded using the binary coded decimal method and rescaled to the function domain. The implementation uses a uniform crossover operator and point mutations with a fixed mutation rate of $\frac{1}{L}$, where L is the number of bits in a solution's binary string.

```
BITS_PER_PARAM = 16

def objective1(vector)
  return vector.inject(0.0) {|sum, x| sum + (x**2.0)}
end

def objective2(vector)
  return vector.inject(0.0) {|sum, x| sum + ((x-2.0)**2.0)}
end

def decode(bitstring, search_space)
  vector = []
  search_space.each_with_index do |bounds, i|
    off, sum, j = i*BITS_PER_PARAM, 0.0, 0
    bitstring[off...(off+BITS_PER_PARAM)].each_char do |c|
      sum += ((c=='1') ? 1.0 : 0.0) * (2.0 ** j.to_f)
      j += 1
    end
    min, max = bounds
    vector << min + ((max-min)/((2.0**BITS_PER_PARAM.to_f)-1.0)) * sum
  end
  return vector
end

```

```
def point_mutation(bitstring)
  child = ""
  bitstring.size.times do |i|
    bit = bitstring[i]
    child << ((rand()<1.0/bitstring.length.to_f) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

def uniform_crossover(parent1, parent2, p_crossover)
  return ""+parent1[:bitstring] if rand()>=p_crossover
  child = ""
  parent1[:bitstring].size.times do |i|
    child << ((rand()<0.5) ? parent1[:bitstring][i] : parent2[:bitstring][i])
  end
  return child
end

def reproduce(selected, population_size, p_crossover)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
    child[:bitstring] = uniform_crossover(p1, p2, p_crossover)
    child[:bitstring] = point_mutation(child[:bitstring])
    children << child
  end
  return children
end

def random_bitstring(num_bits)
  return (0...num_bits).inject(""){|s,i| s<<((rand<0.5) ? "1" : "0")}
end

def calculate_objectives(pop, search_space)
  pop.each do |p|
    p[:vector] = decode(p[:bitstring], search_space)
    p[:objectives] = []
    p[:objectives] << objective1(p[:vector])
    p[:objectives] << objective2(p[:vector])
  end
end

def dominates(p1, p2)
  p1[:objectives].each_with_index do |x,i|
    return false if x > p2[:objectives][i]
  end
  return true
end

def fast_nondominated_sort(pop)
  fronts = Array.new(1){[]}
  pop.each do |p1|
```

```
78      p1[:dom_count], p1[:dom_set] = 0, []
79      pop.each do |p2|
80        if dominates(p1, p2)
81          p1[:dom_set] << p2
82        elsif dominates(p2, p1)
83          p1[:dom_count] += 1
84        end
85      end
86      if p1[:dom_count] == 0
87        p1[:rank] = 0
88        fronts.first << p1
89      end
90    end
91    curr = 0
92    begin
93      next_front = []
94      fronts[curr].each do |p1|
95        p1[:dom_set].each do |p2|
96          p2[:dom_count] -= 1
97          if p2[:dom_count] == 0
98            p2[:rank] = (curr+1)
99            next_front << p2
100         end
101       end
102     end
103     curr += 1
104     fronts << next_front if !next_front.empty?
105   end while curr < fronts.length
106   return fronts
107 end
108
109 def calculate_crowding_distance(pop)
110   pop.each {|p| p[:distance] = 0.0}
111   num_obs = pop.first[:objectives].length
112   num_obs.times do |i|
113     pop.sort!{|x,y| x[:objectives][i]<=>y[:objectives][i]}
114     min, max = pop.first[:objectives][i], pop.last[:objectives][i]
115     range, inf = max-min, 1.0/0.0
116     pop.first[:distance], pop.last[:distance] = inf, inf
117     next if range == 0
118     (1...(pop.length-2)).each do |j|
119       pop[j][:distance] += (pop[j+1][:objectives][i] - pop[j-1][:objectives][i]) / range
120     end
121   end
122 end
123
124 def crowded_comparison_operator(x,y)
125   return y[:distance]<=>x[:distance] if x[:rank] == y[:rank]
126   return x[:rank]<=>y[:rank]
127 end
128
129 def better(x,y)
130   if !x[:distance].nil? and x[:rank] == y[:rank]
```

```
131     return (x[:distance]>y[:distance]) ? x : y
132   end
133   return (x[:rank]<y[:rank]) ? x : y
134 end
135
136 def select_parents(fronts, pop_size)
137   fronts.each {|f| calculate_crowding_distance(f)}
138   offspring = []
139   last_front = 0
140   fronts.each do |front|
141     break if (offspring.length+front.length) > pop_size
142     front.each {|p| offspring << p}
143     last_front += 1
144   end
145   if (remaining = pop_size-offspring.length) > 0
146     fronts[last_front].sort! {|x,y| crowded_comparison_operator(x,y)}
147     offspring += fronts[last_front][0...remaining]
148   end
149   return offspring
150 end
151
152 def weighted_sum(x)
153   return x[:objectives].inject(0.0) {|sum, x| sum+x}
154 end
155
156 def search(problem_size, search_space, max_gens, pop_size, p_crossover)
157   pop = Array.new(pop_size) do |i|
158     {:bitstring=>random_bitstring(problem_size*BITS_PER_PARAM)}
159   end
160   calculate_objectives(pop, search_space)
161   fast_nondominated_sort(pop)
162   selected = Array.new(pop_size){better(pop[rand(pop_size)], pop[rand(pop_size)])}
163   children = reproduce(selected, pop_size, p_crossover)
164   calculate_objectives(children, search_space)
165   max_gens.times do |gen|
166     union = pop + children
167     fronts = fast_nondominated_sort(union)
168     offspring = select_parents(fronts, pop_size)
169     selected = Array.new(pop_size){better(offspring[rand(pop_size)],
          offspring[rand(pop_size)])}
170     pop = children
171     children = reproduce(selected, pop_size, p_crossover)
172     calculate_objectives(children, search_space)
173     best = children.sort!{|x,y| weighted_sum(x)<=>weighted_sum(y)}.first
174     best_s = "[x=#{best[:vector]}, objs=#{best[:objectives].join(', ')}]"
175     puts " > gen=#{gen+1}, fronts=#{fronts.length}, best=#{best_s}"
176   end
177   return children
178 end
179
180 max_gens = 50
181 pop_size = 100
182 p_crossover = 0.98
```

```
problem_size = 1
search_space = Array.new(problem_size) {|i| [-1000, 1000]}

pop = search(problem_size, search_space, max_gens, pop_size, p_crossover)
puts "done!"
```

Listing 4.9: Non-dominated Sorting Genetic Algorithm II (NSGA-II) in the Ruby Programming Language

4.10.6 References

Primary Sources

Srinivas and Deb proposed the NSGA algorithm inspired by Goldberg's notion of a non-dominated sorting procedure [212]. Goldberg proposed a non-dominated sorting procedure in his book in considering the biases in the Pareto optimal solutions provided by VEGA [104]. Srinivas and Deb's NSGA used the sorting procedure as a ranking selection method, and a fitness sharing niching method to maintain stable sub-populations across the Pareto front. Deb, et al. later extended NSGA to address three criticism of the approach: i) the $O(mN^3)$ time complexity, the lack of elitism, and the need for a sharing parameter for the fitness sharing niching method [50, 51].

Learn More

Deb provides in depth coverage of Evolutionary Multiple Objective Optimization algorithms in his book, including a detailed description of the NSGA in Chapter 5 [49].

4.11 Strength Pareto Evolutionary Algorithm

Strength Pareto Evolutionary Algorithm, SPEA, SPEA2.

4.11.1 Taxonomy

Strength Pareto Evolutionary Algorithm is a Multiple Objective Optimization (MOO) algorithm and an Evolutionary Algorithm (EA) from the field of Evolutionary Computation (EC). It belongs to the field of Evolutionary Multiple Objective (EMO) algorithms. Strength Pareto Evolutionary Algorithm is an extension of the Genetic Algorithm for multiple objective optimization problems. It is related to sibling Evolutionary Algorithms such as Non-dominated Sorting Genetic Algorithm (NSGA), Vector-Evaluated Genetic Algorithm (VEGA), and Pareto Archived Evolution Strategy (PAES). There are two versions of SPEA, the original SPEA algorithm and the extension SPEA2. Additional extensions include SPEA+ and iSPEA.

4.11.2 Strategy

The objective of the algorithm is to locate and and maintain a front of non-dominated Pareto optimal solutions. This is achieved by using an evolutionary process (with surrogate procedures for genetic recombination and mutation) to explore the search space, and a selection process that uses a combination of the degree to which a candidate solution is dominated (strength) and an estimation of density of the Pareto front as an assigned fitness. An archive of the Pareto front is maintained separate from the population of candidate solutions used in the evolutionary process, providing a form of elitism.

4.11.3 Procedure

Algorithm 25 provides a pseudo-code listing of the Strength Pareto Evolutionary Algorithm 2 (SPEA2) for minimizing a cost function. The `CalculateRawFitness` function calculates the raw fitness as the sum of the strength values of the solutions that dominate a given candidate, where strength is the number of solutions that a give solution dominate. The `CandidateDensity` function estimates the density of an area of the Pareto front as $\frac{1.0}{\sigma^k+2}$ where σ^k is the Euclidean distance of the objective values between a given solution the kth nearest neighbor of the solution, and k is the square root of the size of the population and archive combined. The `PopulateWithRemainingBest` function iteratively fills the archive with the remaining candidate solutions in order of fitness. The `RemoveMostSimilar` function truncates the archive population removing those members with the smallest σ^k values as calculated against the archive. The `SelectParents` function selects parents from a population using a Genetic Algorithm selection method such as binary tournament selection. The `CrossoverAndMutation` function performs the crossover and mutation genetic operators from the Genetic Algorithm.

Algorithm 25: Pseudo Code for the Strength Pareto Evolutionary Algorithm 2 (SPEA2).

```
Input: Population_size, Archive_size, ProblemSize, P_crossover, P_mutation
Output: Archive
Population ← InitializePopulation(Population_size, ProblemSize);
Archive ← 0;
while True do
    for S_i ∈ Population do
        Si_objectives ← CalculateObjectives(S_i);
    end
    Union ← Population + Archive;
    for S_i ∈ Union do
        Si_raw ← CalculateRawFitness(S_i, Union);
        Si_density ← CalculateSolutionDensity(S_i, Union);
        Si_fitness ← Si_raw + Si_density;
    end
    Archive ← GetNonDominated(Union);
    if Size(Archive) < Archive_size then
        PopulateWithRemainingBest(Union, Archive, Archive_size);
    end
    else if Size(Archive) > Archive_size then
        RemoveMostSimilar(Archive, Archive_size);
    end
    if StopCondition() then
        Archive ← GetNonDominated(Archive);
        Break();
    else
        Selected ← SelectParents(Archive, Population_size);
        Population ← CrossoverAndMutation(Selected, P_crossover, P_mutation);
    end
end
return Archive;
```

4.11.4 Heuristics

- SPEA was designed for and is suited to combinatorial and continuous function multiple objective optimization problem instances.
- A binary representation can be used for continuous function optimization problems in conjunction with classical genetic operators such as one-point crossover and point mutation.
- A k value of 1 may be used for efficiency whilst still providing useful results.
- The size of the archive is commonly smaller than the size of the population.
- There is a lot of room for implementation optimizations in density and Pareto dominance calculations.

4.11.5 Code Listing

Listing 4.10 provides an example of the Strength Pareto Evolutionary Algorithm 2 (SPEA2) implemented in the Ruby Programming Language. The demonstration problem is an instance of continuous multiple objective function optimization called SCH (problem one in [51]). The problem seeks the minimum of two functions: $f1 = \sum_{i=1}^{n} x_i^2$ and $f2 = \sum_{i=1}^{n}(x_i - 2)^2$, $-10^3 \leq x_i \leq 10^3$ and $n = 1$. The optimal solution for this function are $x \in [0, 2]$. The algorithm is an implementation of SPEA2 based on the presentation by Zitzler, Laumanns, and Thiele [251]. The algorithm uses a binary string representation (16 bits per objective function parameter) that is decoded using the binary coded decimal method and rescaled to the function domain. The implementation uses a uniform crossover operator and point mutations with a fixed mutation rate of $\frac{1}{L}$, where L is the number of bits in a solution's binary string.

```
BITS_PER_PARAM = 16

def objective1(vector)
  return vector.inject(0.0) {|sum, x| sum + (x**2.0)}
end

def objective2(vector)
  return vector.inject(0.0) {|sum, x| sum + ((x-2.0)**2.0)}
end

def decode(bitstring, search_space)
  vector = []
  search_space.each_with_index do |bounds, i|
    off, sum, j = i*BITS_PER_PARAM, 0.0, 0
    bitstring[off...(off+BITS_PER_PARAM)].each_char do |c|
      sum += ((c=='1') ? 1.0 : 0.0) * (2.0 ** j.to_f)
      j += 1
    end
    min, max = bounds
    vector << min + ((max-min)/((2.0**BITS_PER_PARAM.to_f)-1.0)) * sum
```

```
21    end
22    return vector
23  end
24
25  def point_mutation(bitstring)
26    child = ""
27    bitstring.size.times do |i|
28      bit = bitstring[i]
29      child << ((rand()<1.0/bitstring.length.to_f) ? ((bit=='1') ? "0" : "1") : bit)
30    end
31    return child
32  end
33
34  def uniform_crossover(parent1, parent2, p_crossover)
35    return ""+parent1[:bitstring] if rand()>=p_crossover
36    child = ""
37    parent1[:bitstring].size.times do |i|
38      child << ((rand()<0.5) ? parent1[:bitstring][i] : parent2[:bitstring][i])
39    end
40    return child
41  end
42
43  def reproduce(selected, population_size, p_crossover)
44    children = []
45    selected.each_with_index do |p1, i|
46      p2 = (i.even?) ? selected[i+1] : selected[i-1]
47      child = {}
48      child[:bitstring] = uniform_crossover(p1, p2, p_crossover)
49      child[:bitstring] = point_mutation(child[:bitstring])
50      children << child
51    end
52    return children
53  end
54
55  def random_bitstring(num_bits)
56    return (0...num_bits).inject(""){|s,i| s<<((rand<0.5) ? "1" : "0")}
57  end
58
59  def calculate_objectives(pop, search_space)
60    pop.each do |p|
61      p[:vector] = decode(p[:bitstring], search_space)
62      p[:objectives] = []
63      p[:objectives] << objective1(p[:vector])
64      p[:objectives] << objective2(p[:vector])
65    end
66  end
67
68  def dominates(p1, p2)
69    p1[:objectives].each_with_index do |x,i|
70      return false if x > p2[:objectives][i]
71    end
72    return true
73  end
```

```ruby

def weighted_sum(x)
  return x[:objectives].inject(0.0) {|sum, x| sum+x}
end

def distance(c1, c2)
  sum = 0.0
  c1.each_with_index {|x,i| sum += (c1[i]-c2[i])**2.0}
  return Math.sqrt(sum)
end

def calculate_dominated(pop)
  pop.each do |p1|
    p1[:dom_set] = pop.select {|p2| dominates(p1, p2) }
  end
end

def calculate_raw_fitness(p1, pop)
  return pop.inject(0.0) do |sum, p2|
    (dominates(p2, p1)) ? sum + p2[:dom_set].size.to_f : sum
  end
end

def calculate_density(p1, pop)
  pop.each {|p2| p2[:dist] = distance(p1[:objectives], p2[:objectives])}
  list = pop.sort{|x,y| x[:dist]<=>y[:dist]}
  k = Math.sqrt(pop.length).to_i
  return 1.0 / (list[k][:dist] + 2.0)
end

def calculate_fitness(pop, archive, search_space)
  calculate_objectives(pop, search_space)
  union = archive + pop
  calculate_dominated(union)
  union.each do |p1|
    p1[:raw_fitness] = calculate_raw_fitness(p1, union)
    p1[:density] = calculate_density(p1, union)
    p1[:fitness] = p1[:raw_fitness] + p1[:density]
  end
end

def environmental_selection(pop, archive, archive_size)
  union = archive + pop
  environment = union.select {|p| p[:fitness]<1.0}
  if environment.length < archive_size
    union.sort!{|x,y| x[:fitness]<=>y[:fitness]}
    union.each do |p|
      environment << p if p[:fitness] >= 1.0
      break if environment.length >= archive_size
    end
  elsif environment.length > archive_size
    begin
      k = Math.sqrt(environment.length).to_i
```

```
    environment.each do |p1|
      environment.each {|p2| p2[:dist] = distance(p1[:objectives], p2[:objectives])}
      list = environment.sort{|x,y| x[:dist]<=>y[:dist]}
      p1[:density] = list[k][:dist]
    end
    environment.sort!{|x,y| x[:density]<=>y[:density]}
    environment.shift
  end until environment.length >= archive_size
 end
 return environment
end

def binary_tournament(pop)
 s1, s2 = pop[rand(pop.size)], pop[rand(pop.size)]
 return (s1[:fitness] < s2[:fitness]) ? s1 : s2
end

def search(problem_size, search_space, max_gens, pop_size, archive_size, p_crossover)
 pop = Array.new(pop_size) do |i|
  {:bitstring=>random_bitstring(problem_size*BITS_PER_PARAM)}
 end
 gen, archive = 0, []
 begin
  calculate_fitness(pop, archive, search_space)
  archive = environmental_selection(pop, archive, archive_size)
  best = archive.sort{|x,y| weighted_sum(x)<=>weighted_sum(y)}.first
  puts ">gen=#{gen}, best: x=#{best[:vector]}, objs=#{best[:objectives].join(', ')}"
  if gen >= max_gens
   archive = archive.select {|p| p[:fitness]<1.0}
   break
  else
   selected = Array.new(pop_size){binary_tournament(archive)}
   pop = reproduce(selected, pop_size, p_crossover)
   gen += 1
  end
 end while true
 return archive
end

max_gens = 50
pop_size = 80
archive_size = 40
p_crossover = 0.90
problem_size = 1
search_space = Array.new(problem_size) {|i| [-1000, 1000]}

pop = search(problem_size, search_space, max_gens, pop_size, archive_size, p_crossover)
puts "done!"
```

Listing 4.10: Strength Pareto Evolutionary Algorithm 2 SPEA2) in the Ruby Programming Language

4.11.6 References

Primary Sources

Zitzler and Thiele introduced the Strength Pareto Evolutionary Algorithm as a technical report on a multiple objective optimization algorithm with elitism and clustering along the Pareto front [252]. The technical report was later published [253]. The Strength Pareto Evolutionary Algorithm was developed as a part of Zitzler PhD thesis [248]. Zitzler, Laumanns, and Thiele later extended SPEA to address some inefficiencies the approach, called SPEA2 that was released as a technical report [250] and later published [251]. SPEA2 provided a fine-grained fitness assignment, density estimation on the Pareto front, and an archive truncation operator.

Learn More

Zitzler, Laumanns, and Bleuler provide a tutorial on SPEA2 as a book chapter that considers the basics of multiple objective optimization, and the differences from SPEA and the other related Multiple Objective Evolutionary Algorithms [249].

Chapter 5

Probabilistic Algorithms

5.1 Overview

todo

5.2 Cross-Entropy Method

The heading and alternate headings for the algorithm description.

5.2.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.2.2 Inspiration

A textual description of the inspiring system.

5.2.3 Metaphor

A textual description of the algorithm by analogy.

5.2.4 Strategy

A textual description of the information processing strategy.

5.2.5 Procedure

A pseudo code description of the algorithms procedure.

5.2.6 Heuristics

A bullet-point listing of best practice usage.

5.2.7 Code Listing

A code listing and a terse description of the listing.

5.2.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.3 Population-Based Incremental Learning

The heading and alternate headings for the algorithm description.

5.3.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.3.2 Inspiration

A textual description of the inspiring system.

5.3.3 Metaphor

A textual description of the algorithm by analogy.

5.3.4 Strategy

A textual description of the information processing strategy.

5.3.5 Procedure

A pseudo code description of the algorithms procedure.

5.3.6 Heuristics

A bullet-point listing of best practice usage.

5.3.7 Code Listing

A code listing and a terse description of the listing.

5.3.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.4 Probabilistic Incremental Program Evolution

The heading and alternate headings for the algorithm description.

5.4.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.4.2 Inspiration

A textual description of the inspiring system.

5.4.3 Metaphor

A textual description of the algorithm by analogy.

5.4.4 Strategy

A textual description of the information processing strategy.

5.4.5 Procedure

A pseudo code description of the algorithms procedure.

5.4.6 Heuristics

A bullet-point listing of best practice usage.

5.4.7 Code Listing

A code listing and a terse description of the listing.

5.4.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.5 Compact Genetic Algorithm

The heading and alternate headings for the algorithm description.

5.5.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.5.2 Inspiration

A textual description of the inspiring system.

5.5.3 Metaphor

A textual description of the algorithm by analogy.

5.5.4 Strategy

A textual description of the information processing strategy.

5.5.5 Procedure

A pseudo code description of the algorithms procedure.

5.5.6 Heuristics

A bullet-point listing of best practice usage.

5.5.7 Code Listing

A code listing and a terse description of the listing.

5.5.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.6 Extended Compact Genetic Algorithm

The heading and alternate headings for the algorithm description.

5.6.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.6.2 Inspiration

A textual description of the inspiring system.

5.6.3 Metaphor

A textual description of the algorithm by analogy.

5.6.4 Strategy

A textual description of the information processing strategy.

5.6.5 Procedure

A pseudo code description of the algorithms procedure.

5.6.6 Heuristics

A bullet-point listing of best practice usage.

5.6.7 Code Listing

A code listing and a terse description of the listing.

5.6.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.7 Bayesian Optimization Algorithm

The heading and alternate headings for the algorithm description.

5.7.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.7.2 Inspiration

A textual description of the inspiring system.

5.7.3 Metaphor

A textual description of the algorithm by analogy.

5.7.4 Strategy

A textual description of the information processing strategy.

5.7.5 Procedure

A pseudo code description of the algorithms procedure.

5.7.6 Heuristics

A bullet-point listing of best practice usage.

5.7.7 Code Listing

A code listing and a terse description of the listing.

5.7.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.8 Hierarchical Bayesian Optimization Algorithm

The heading and alternate headings for the algorithm description.

5.8.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.8.2 Inspiration

A textual description of the inspiring system.

5.8.3 Metaphor

A textual description of the algorithm by analogy.

5.8.4 Strategy

A textual description of the information processing strategy.

5.8.5 Procedure

A pseudo code description of the algorithms procedure.

5.8.6 Heuristics

A bullet-point listing of best practice usage.

5.8.7 Code Listing

A code listing and a terse description of the listing.

5.8.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.9 Univariate Marginal Distribution Algorithm

The heading and alternate headings for the algorithm description.

5.9.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.9.2 Inspiration

A textual description of the inspiring system.

5.9.3 Metaphor

A textual description of the algorithm by analogy.

5.9.4 Strategy

A textual description of the information processing strategy.

5.9.5 Procedure

A pseudo code description of the algorithms procedure.

5.9.6 Heuristics

A bullet-point listing of best practice usage.

5.9.7 Code Listing

A code listing and a terse description of the listing.

5.9.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.10 Bivariate Marginal Distribution Algorithm

The heading and alternate headings for the algorithm description.

5.10.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.10.2 Inspiration

A textual description of the inspiring system.

5.10.3 Metaphor

A textual description of the algorithm by analogy.

5.10.4 Strategy

A textual description of the information processing strategy.

5.10.5 Procedure

A pseudo code description of the algorithms procedure.

5.10.6 Heuristics

A bullet-point listing of best practice usage.

5.10.7 Code Listing

A code listing and a terse description of the listing.

5.10.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.11 Gaussian Adaptation

The heading and alternate headings for the algorithm description.

5.11.1 Taxonomy

A small tree diagram showing related fields and algorithms.

5.11.2 Inspiration

A textual description of the inspiring system.

5.11.3 Metaphor

A textual description of the algorithm by analogy.

5.11.4 Strategy

A textual description of the information processing strategy.

5.11.5 Procedure

A pseudo code description of the algorithms procedure.

5.11.6 Heuristics

A bullet-point listing of best practice usage.

5.11.7 Code Listing

A code listing and a terse description of the listing.

5.11.8 References

An bullet-point annotated reference list of primary sources and useful resources.

5.12 Summary

todo

Chapter 6

Swarm Algorithms

6.1 Overview

todo

6.2 Particle Swarm Optimization

The heading and alternate headings for the algorithm description.

6.2.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.2.2 Inspiration

A textual description of the inspiring system.

6.2.3 Metaphor

A textual description of the algorithm by analogy.

6.2.4 Strategy

A textual description of the information processing strategy.

6.2.5 Procedure

A pseudo code description of the algorithms procedure.

6.2.6 Heuristics

A bullet-point listing of best practice usage.

6.2.7 Code Listing

A code listing and a terse description of the listing.

6.2.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.3 AntNet

The heading and alternate headings for the algorithm description.

6.3.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.3.2 Inspiration

A textual description of the inspiring system.

6.3.3 Metaphor

A textual description of the algorithm by analogy.

6.3.4 Strategy

A textual description of the information processing strategy.

6.3.5 Procedure

A pseudo code description of the algorithms procedure.

6.3.6 Heuristics

A bullet-point listing of best practice usage.

6.3.7 Code Listing

A code listing and a terse description of the listing.

6.3.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.4 Ant System

The heading and alternate headings for the algorithm description.

6.4.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.4.2 Inspiration

A textual description of the inspiring system.

6.4.3 Metaphor

A textual description of the algorithm by analogy.

6.4.4 Strategy

A textual description of the information processing strategy.

6.4.5 Procedure

A pseudo code description of the algorithms procedure.

6.4.6 Heuristics

A bullet-point listing of best practice usage.

6.4.7 Code Listing

A code listing and a terse description of the listing.

6.4.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.5 MAX-MIN Ant System

The heading and alternate headings for the algorithm description.

6.5.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.5.2 Inspiration

A textual description of the inspiring system.

6.5.3 Metaphor

A textual description of the algorithm by analogy.

6.5.4 Strategy

A textual description of the information processing strategy.

6.5.5 Procedure

A pseudo code description of the algorithms procedure.

6.5.6 Heuristics

A bullet-point listing of best practice usage.

6.5.7 Code Listing

A code listing and a terse description of the listing.

6.5.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.6 Rank-Based Ant System

The heading and alternate headings for the algorithm description.

6.6.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.6.2 Inspiration

A textual description of the inspiring system.

6.6.3 Metaphor

A textual description of the algorithm by analogy.

6.6.4 Strategy

A textual description of the information processing strategy.

6.6.5 Procedure

A pseudo code description of the algorithms procedure.

6.6.6 Heuristics

A bullet-point listing of best practice usage.

6.6.7 Code Listing

A code listing and a terse description of the listing.

6.6.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.7 Ant Colony System

The heading and alternate headings for the algorithm description.

6.7.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.7.2 Inspiration

A textual description of the inspiring system.

6.7.3 Metaphor

A textual description of the algorithm by analogy.

6.7.4 Strategy

A textual description of the information processing strategy.

6.7.5 Procedure

A pseudo code description of the algorithms procedure.

6.7.6 Heuristics

A bullet-point listing of best practice usage.

6.7.7 Code Listing

A code listing and a terse description of the listing.

6.7.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.8 Multiple Ant Colony System

The heading and alternate headings for the algorithm description.

6.8.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.8.2 Inspiration

A textual description of the inspiring system.

6.8.3 Metaphor

A textual description of the algorithm by analogy.

6.8.4 Strategy

A textual description of the information processing strategy.

6.8.5 Procedure

A pseudo code description of the algorithms procedure.

6.8.6 Heuristics

A bullet-point listing of best practice usage.

6.8.7 Code Listing

A code listing and a terse description of the listing.

6.8.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.9 Population-based Ant Colony Optimization

The heading and alternate headings for the algorithm description.

6.9.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.9.2 Inspiration

A textual description of the inspiring system.

6.9.3 Metaphor

A textual description of the algorithm by analogy.

6.9.4 Strategy

A textual description of the information processing strategy.

6.9.5 Procedure

A pseudo code description of the algorithms procedure.

6.9.6 Heuristics

A bullet-point listing of best practice usage.

6.9.7 Code Listing

A code listing and a terse description of the listing.

6.9.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.10 Bees Algorithm

The heading and alternate headings for the algorithm description.

6.10.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.10.2 Inspiration

A textual description of the inspiring system.

6.10.3 Metaphor

A textual description of the algorithm by analogy.

6.10.4 Strategy

A textual description of the information processing strategy.

6.10.5 Procedure

A pseudo code description of the algorithms procedure.

6.10.6 Heuristics

A bullet-point listing of best practice usage.

6.10.7 Code Listing

A code listing and a terse description of the listing.

6.10.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.11 Bacterial Foraging Optimization Algorithm

The heading and alternate headings for the algorithm description.

6.11.1 Taxonomy

A small tree diagram showing related fields and algorithms.

6.11.2 Inspiration

A textual description of the inspiring system.

6.11.3 Metaphor

A textual description of the algorithm by analogy.

6.11.4 Strategy

A textual description of the information processing strategy.

6.11.5 Procedure

A pseudo code description of the algorithms procedure.

6.11.6 Heuristics

A bullet-point listing of best practice usage.

6.11.7 Code Listing

A code listing and a terse description of the listing.

6.11.8 References

An bullet-point annotated reference list of primary sources and useful resources.

6.12 Summary

todo

Chapter 7

Immune Algorithms

7.1 Overview

todo

7.2 Clonal Selection Algorithm

The heading and alternate headings for the algorithm description.

7.2.1 Taxonomy

A small tree diagram showing related fields and algorithms.

7.2.2 Inspiration

A textual description of the inspiring system.

7.2.3 Metaphor

A textual description of the algorithm by analogy.

7.2.4 Strategy

A textual description of the information processing strategy.

7.2.5 Procedure

A pseudo code description of the algorithms procedure.

7.2.6 Heuristics

A bullet-point listing of best practice usage.

7.2.7 Code Listing

A code listing and a terse description of the listing.

7.2.8 References

An bullet-point annotated reference list of primary sources and useful resources.

7.3 Negative Selection Algorithm

The heading and alternate headings for the algorithm description.

7.3.1 Taxonomy

A small tree diagram showing related fields and algorithms.

7.3.2 Inspiration

A textual description of the inspiring system.

7.3.3 Metaphor

A textual description of the algorithm by analogy.

7.3.4 Strategy

A textual description of the information processing strategy.

7.3.5 Procedure

A pseudo code description of the algorithms procedure.

7.3.6 Heuristics

A bullet-point listing of best practice usage.

7.3.7 Code Listing

A code listing and a terse description of the listing.

7.3.8 References

An bullet-point annotated reference list of primary sources and useful resources.

7.4 Artificial Immune Recognition System

The heading and alternate headings for the algorithm description.

7.4.1 Taxonomy

A small tree diagram showing related fields and algorithms.

7.4.2 Inspiration

A textual description of the inspiring system.

7.4.3 Metaphor

A textual description of the algorithm by analogy.

7.4.4 Strategy

A textual description of the information processing strategy.

7.4.5 Procedure

A pseudo code description of the algorithms procedure.

7.4.6 Heuristics

A bullet-point listing of best practice usage.

7.4.7 Code Listing

A code listing and a terse description of the listing.

7.4.8 References

An bullet-point annotated reference list of primary sources and useful resources.

7.5 Immune Network Algorithm

The heading and alternate headings for the algorithm description.

7.5.1 Taxonomy

A small tree diagram showing related fields and algorithms.

7.5.2 Inspiration

A textual description of the inspiring system.

7.5.3 Metaphor

A textual description of the algorithm by analogy.

7.5.4 Strategy

A textual description of the information processing strategy.

7.5.5 Procedure

A pseudo code description of the algorithms procedure.

7.5.6 Heuristics

A bullet-point listing of best practice usage.

7.5.7 Code Listing

A code listing and a terse description of the listing.

7.5.8 References

An bullet-point annotated reference list of primary sources and useful resources.

7.6 Dendritic Cell Algorithm

The heading and alternate headings for the algorithm description.

7.6.1 Taxonomy

A small tree diagram showing related fields and algorithms.

7.6.2 Inspiration

A textual description of the inspiring system.

7.6.3 Metaphor

A textual description of the algorithm by analogy.

7.6.4 Strategy

A textual description of the information processing strategy.

7.6.5 Procedure

A pseudo code description of the algorithms procedure.

7.6.6 Heuristics

A bullet-point listing of best practice usage.

7.6.7 Code Listing

A code listing and a terse description of the listing.

7.6.8 References

An bullet-point annotated reference list of primary sources and useful resources.

7.7 Summary

todo

Part III

Extensions

Chapter 8

Advanced Topics

A chapter focused on applying, testing, visualizing, saving results, and comparing algorithms. The meta concerns once an algorithm is selected for a given practical problem solving scenario.

8.1 Programming Paradigms

Algorithms can be implements on many different programming paradigms. Take the GA for example and realize it using a bunch of different paradigms.

8.1.1 Procedural Programming

The GA under a procedural paradigm

8.1.2 Object-Oriented Programming

The GA under a object oriented paradigm. Strategy pattern. modular operators, etc.

8.1.3 Agent Oriented Programming

A GA under an agent oriented programming paradigm. not really suited. algorithm as an agent with goals?

8.1.4 Functional Programming

The GA under a functional paradigm. closure etc

8.1.5 Meta-Programming

A GA under meta programming. A DSL i guess.

8.1.6 Flow Programming

A GA under a data flow or pipeline model.

8.1.7 Map Reduce

A GA under a map reduce paradigm.

8.2 Devising New Algorithms

A methodology for devising new unconventional optimization algorithms...

8.2.1 Conceptual Framework for Bio-Inspired Algorithms

A generic methodology for devising new biologically inspired algorithms

8.2.2 Information Processing Methodology

An info processing centric approach to devising new algorithms

8.2.3 Investigation

small models, rigor

8.2.4 Communication

you need to effectively describe them, like as in this book! goal is to be known and used, make it open and usable by anyone. like open source, documented, common languages, benchmark problems, a website, lots of papers

8.3 Testing Algorithms

This section will focus on the problem that 'adaptive systems work even when they are not implemented correctly' (they work in-spite of the developer). Topics will include unit testing algorithms, system testing software, specific concerns when testing inspired algorithms, examples of testing algorithms with the ruby unit testing framework, examples of testing algorithms with rspec.

8.3.1 Types of Testing

unit, TDD, system, user acceptance, black box, white box

8.3.2 Algorithm Testing Methodology

testing is hard these systems 'work' even with bugs, hard to test present a methodology for testing - discrete unit tests, behavior testing

8.3.3 Example

develop and show tests for the GA

8.4 Visualizing Algorithms

This section will focus on the use of visualization as a low-fidelity form of system testing. Topics will include free visualization packages such as R, GNUPlot and Processing. Examples visualizing a decision surface, a functions response surface, and candidate solutions.

8.4.1 Visualizing

we can do it as a form of testing. research aid - view on a complex process, can observe, take notes, formulate hypothesis think of all the measures you can, than measure them

Offline Plots

examples?

Online Plots

examples?

8.4.2 Visualization Tools

can use lots of things, can use lots of things

8.4.3 Example

Visuzlize genes through time for a ga run, with fitness graphs, and plots of domain

8.5 Saving Algorithm Results

This section will focus on algorithms and techniques as a fallible means to an end and the need to maintain save results. Topics will include check-pointing, storage in a database, storage on the filesystem, and algorithm restarting. Examples will be given for database, filesystem checkpointing and algorithm restarting.

8.5.1 Check-pointing

algorithms crash and it sucks, need to be able to pickup where you left off

8.5.2 Share Results

make them public with papers and source code

8.5.3 Example

show an example of check pointing

8.6 Comparing Algorithms

This section will focus on comparing algorithm's based on the solutions they provide. Topics will include the use statistical hypothesis testing and free software such as R, algorithm parameter selection, distribution testing, distribution comparisons. Examples will be given for algorithm parameter selection, result distribution classification, and pair-wise result distribution comparison.

8.6.1 No Free Lunch

all same over all problems with no prior info

8.6.2 Benchmarking

standard problem instances what problems? what algorithms? what configurations what are you measuring? what are you comparing?

8.6.3 Statistical Hypothesis Testing

you need stats or you will be killed by Zed Shaw need stats to compare results

8.6.4 Example

genetic algorithm vs something, use R to compare

8.7 Summary

We learned lots of advanced topics, there are more.

Index

Bibliography

[1] S. Aaronson. NP-complete problems and physical reality. *ACM SIGACT News (COLUMN: Complexity theory)*, 36(1):30–52, 2005.

[2] M. M. Ali, C. Storey, and A. Trn. Application of stochastic global optimization algorithms to practical problems. *Journal of Optimization Theory and Applications*, 95(3):545–563, 1997.

[3] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50:5–43, 2003.

[4] Peter J. Angeline. Two self-adaptive crossover operators for genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 89–110. MIT Press, 1996.

[5] Thomas Back. Optimal mutation rates in genetic search. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 2–9, 1993.

[6] Thomas Bäck, D.B Fogel, and Z Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. IoP, 2000.

[7] Thomas Bäck, D.B Fogel, and Z Michalewicz, editors. *Evolutionary Computation 2: Advanced Algorithms and Operations*. IoP, 2000.

[8] Thomas Bäck, Frank Hoffmeister, and Hans paul Schwefel. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9, 1991.

[9] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

[10] John Daniel Bagley. *The behavior of adaptive systems which employ genetic and correlation algorithms*. PhD thesis, University of Michigan, 1967.

[11] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

[12] J.F. Bard, T.A. Feo, and S. Holland. A GRASP for scheduling printed wiring board assembly. *I.I.E. Trans.*, 28:155–165, 1996.

[13] R. Battiti and M. Brunato. *Handbook of Metaheuristics*, chapter Reactive Search Optimization: Learning while Optimizing. Springer Verlag, 2nd edition, 2009.

[14] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*. Springer, 2008.

[15] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. Technical Report TR-95-052, International Computer Science Institute, Berkeley, CA, 1995.

[16] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4):610–637, 2001.

[17] R. Battiti and G. Tecchiolli. Parallel biased search for combinatorial optimization: genetic algorithms and tabu. *Microprocessors and Microsystems*, 16(7):351–367, 1992.

[18] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[19] R. Battiti and G. Tecchiolli. Simulated annealing and tabu search in the long run: a comparison on qap tasks. *Computer and Mathematics with Applications*, 28(6):1–8, 1994.

[20] R. Battiti and G. Tecchiolli. Local search with memory: Benchmarking rts. *Operations Research Spektrum*, 17(2/3):67–86, 1995.

[21] R. Battiti and G. Tecchiolli. Training neural nets with the reactive tabu search. *IEEE Transactions on Neural Networks*, 6(5):1185–1200, 1995.

[22] Roberto Battiti. Machine learning methods for parameter tuning in heuristics. In *5th DIMACS Challenge Workshop: Experimental Methodology Day*, 1996.

[23] E. B. Baum. Towards practical "neural" computation for combinatorial optimization problems. In *AIP conference proceedings: Neural Networks for Computing*, pages 53–64, 1986.

[24] J. Baxter. Local optima avoidance in depot location. *Journal of the Operational Research Society*, 32:815–819, 1981.

[25] Janine M. Benyus. *Biomimicry: innovation inspired by nature*. Quill, 1998.

[26] D. Bergemann and J. Valimaki. Bandit problems. Cowles Foundation Discussion Papers 1551, Cowles Foundation, Yale University, January 2006.

[27] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies: A comprehensive introduction. *Natural Computing: an international journal*, 1(1):3–52, 2002.

[28] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[29] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[30] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems*. Oxford University Press US, 1999.

[31] L. B. Booker, D. E. Goldberg, and J. H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40:235–282, 1989.

[32] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[33] Samuel H. Brooks. A discussion of random methods for seeking maxima. *Operations Research*, 6(2):244–251, 1958.

[34] Larry Bull and Tim Kovacs. *Foundations of learning classifier systems*. Springer, 2005.

[35] E. K. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, and S. Schulenburg. *Handbook of Metaheuristics*, chapter Hyper-heuristics: An emerging direction in modern search technology, pages 457–474. Kluwer, 2003.

[36] E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyper-heuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470, 2003.

[37] M. V. Butz and S. W. Wilson. An algorithmic description of xcs. *Journal of Soft Computing*, 6(3–4):144–153, 2002.

[38] Leandro N. De Castro and Fernando J. Von Zuben. *Recent developments in biologically inspired computing*. Idea Group Inc, 2005.

[39] Uday K. Chakraborty. *Advances in Differential Evolution*. Springer, 2008.

[40] David Corne, Marco Dorigo, and Fred Glover. *New ideas in optimization*. McGraw-Hill, 1999.

[41] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187, 1985.

[42] D. Cvijovic and J. Klinowski. Taboo search: An approach to the multiple minima problem. *Science*, 267:664–666, 1995.

[43] Charles Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* John Murray, 1859.

[44] L. Davis. Bit-climbing, representational bias, and test suite design. In *Proceedings of the fourth international conference on genetic algorithms*, pages 18–23, 1991.

[45] Richard Dawkins. *The selfish gene.* Oxford University Press, 1976.

[46] L. N. de Castro and F. J. Von Zuben. *Recent developments in biologically inspired computing*, chapter From biologically inspired computing to natural computing. Idea Group, 2005.

[47] Leandro N. de Castro and Jonathan Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach.* Springer, 2002.

[48] K. Deb and R. B. Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9:115–148, 1995.

[49] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms.* John Wiley and Sons, 2001.

[50] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *Parallel Problem Solving from Nature PPSN VI*, 1917:849–858, 2000.

[51] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[52] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments.* Springer, 2009.

[53] Daniel C. Dennett. *Darwin's Dangerous Idea.* Simon & Schuster, 1995.

[54] Marco Dorigo and Thomas Stützle. *Ant colony optimization.* MIT Press, 2004.

[55] Stefan Droste, Thomas Jansen, and Ingo Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory of Computing Systems*, 39(4):525–544, 2006.

[56] Agoston E. Eiben and James E. Smith. *Introduction to evolutionary computing.* Springer, 2003.

[57] Andries P. Engelbrecht. *Computational intelligence: an introduction.* John Wiley and Sons, second edition, 2007.

[58] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[59] T.A. Feo, J. Bard, and S. Holland. A grasp for scheduling printed wiring board assembly. Technical Report TX 78712-1063, Operations Research Group, Department of Mechanical Engineering, The University of Texas at Austin, 1993.

[60] T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

[61] T.A. Feo, K. Sarathy, and J. McGahan. A grasp for single machine scheduling with sequence dependent setup costs and linear delay penalties. Technical Report TX 78712-1063, Operations Research Group, Department of Mechanical Engineering, The University of Texas at Austin, 1994.

[62] T.A. Feo, K. Venkatraman, and J.F. Bard. A GRASP for a difficult single machine scheduling problem. *Computers & Operations Research*, 18:635–643, 1991.

[63] Thomas A. Feo, Kishore Sarathy, and John McGahan. A grasp for single machine scheduling with sequence dependent setup costs and linear delay penalties. *Computers & Operations Research*, 23(9):881–895, 1996.

[64] C. Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[65] C. Ferreira. *Soft Computing and Industry: Recent Applications*, chapter Gene Expression Programming in Problem Solving, pages 635–654. Springer-Verlag, 2002.

[66] C. Ferreira. *Recent Developments in Biologically Inspired Computing*, chapter Gene Expression Programming and the Evolution of computer programs, pages 82–103. Idea Group Publishing, 2005.

[67] Candida Ferreira. *Gene expression programming: mathematical modeling by an artificial intelligence*. Springer-Verlag, second edition, 2006.

[68] P. Festa and M. G. C. Resende. *Essays and Surveys on Metaheuristics*, chapter GRASP: An annotated bibliography, pages 325–367. Kluwer Academic Publishers, 2002.

[69] C.-N. Fiechter. A parallel tabu search algorithm for large traveling salesman problems. *Discrete Applied Mathematics*, 3(6):243–267, 1994.

[70] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008.

[71] D. B. Fogel. *System Identification Through Simulated Evolution: A Machine Learning Approach to Modeling*. Needham Heights, 1991.

[72] D. B. Fogel. *Evolving artificial intelligence*. PhD thesis, University of California, San Diego, CA, USA, 1992.

[73] D. B. Fogel, L. J. Fogel, and J.W. Atmar. Meta-evolutionary programming. In *Proc. 25th Asilomar Conf. Signals, Systems, and Computers*, pages 540–545, 1991.

[74] David B. Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence*. IEEE Press, 1995.

[75] David B. Fogel. *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1998.

[76] L. J. Fogel. Autonomous automata. *Industrial Research*, 4:14–19, 1962.

[77] L. J. Fogel. *On the Organization of Intellect*. PhD thesis, UCLA, 1964.

[78] L. J. Fogel. The future of evolutionary programming. In *Proceedings of the Conference on Signals, Systems and Computers*, 1990.

[79] L. J. Fogel. *Computational Intelligence: Imitating Life*, chapter Evolutionary Programming in Perspective: the Top-down View, pages 135–146. IEEE Press, 1994.

[80] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution*. Wiley, 1966.

[81] N. Forbes. Biologically inspired computing. *Computing in Science and Engineering*, 2(6):83–87, 2000.

[82] Nancy Forbes. *Imitation of Life: How Biology Is Inspiring Computing*. The MIT Press, 2005.

[83] Stephanie Forrest and Melanie Mitchell. Relative building-block fitness and the building-block hypothesis. In *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, 1993.

[84] D. R. Frantz. *Non-linearities in genetic adaptive search*. PhD thesis, University of Michigan, 1972.

[85] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, 1990.

[86] Douglas Futuyma. *Evolution*. Sinauer Associates Inc., 2nd edition, 2009.

[87] Michel Gendreau. *Handbook of Metaheuristics*, chapter 2: An Introduction to Tabu Search, pages 37–54. Springer, 2003.

[88] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

[89] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(5):533–549, 1986.

[90] F. Glover. Tabu search – Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[91] F Glover. Tabu search – Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[92] F. Glover. Tabu search: A tutorial. *Interfaces*, 4:74–94, 1990.

[93] F. Glover. Tabu search for nonlinear and parametric optimization (with links to genetic algorithms). *Discrete Applied Mathematics*, 49:231–255, 1994.

[94] F. Glover. *Artificial Evolution*, chapter A Template For Scatter Search And Path Relinking, page 13. Sprinter, 1998.

[95] F. Glover. *New Ideas in Optimization*, chapter Scatter search and path relinking, pages 297–316. McGraw-Hill Ltd., 1999.

[96] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39(3):653–684, 2000.

[97] F. Glover, M. Laguna, and R. Martí. *Advances in Evolutionary Computation: Theory and Applications*, chapter Scatter Search, pages 519–537. Springer-Verlag, 2003.

[98] F. Glover and C. McMillan. The general employee scheduling problem: an integration of ms and ai. *Computers and Operations Research*, 13(5):536–573, 1986.

[99] Fred Glover and Gary A. Kochenberger. *Handbook of metaheuristics*. Springer, 2003.

[100] Fred Glover and Eric Taillard. A user's guide to tabu search. *Annals of Operations Research*, 41(1):1–28, 1993.

[101] Fred W. Glover and Manuel Laguna. *Tabu Search*. Springer, 1998.

[102] D. E. Goldberg, K. Deb, and J. H. Clark. Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 6:333–362, 1992.

[103] David E. Goldberg. Genetic and evolutionary algorithms come of age. *Communications of the ACM*, 37(3):113–119, 1994.

[104] David Edward Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, 1989.

[105] David Edward Goldberg. *The design of innovation: lessons from and for competent genetic algorithms*. Springer, 2002.

[106] I Guyon and A Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[107] P. Hansen and N. Mladenovic. *Meta-heuristics, Advances and trends in local search paradigms for optimization*, chapter An introduction to Variable neighborhood search, pages 433–458. Kluwer Academic Publishers, 1998.

[108] P. Hansen and N. Mladenovic. *Handbook of Applied Optimization*, chapter Variable neighbourhood search, pages 221–234. Oxford University Press, 2002.

[109] P. Hansen and N. Mladenovic. *Handbook of metaheuristics*, chapter 6: Variable Neighborhood Search, pages 145–184. Springer, 2003.

[110] Pierre Hansen and Nenad Mladenovic. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[111] Pierre Hansen, Nenad Mladenovic, and Dionisio Perez-Britos. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4):1381–1231, 2001.

[112] J.P. Hart and A.W. Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6:107–114, 1987.

[113] J. H. Holland. Information processing in adaptive systems. In *Processing of Information in the Nervous System*, pages 330–338, 1962.

[114] J. H. Holland. Adaptive plans optimal for payoff-only environments. In *Proceedings of the Second Hawaii Conference on Systems Sciences*, 1969.

[115] J. H. Holland. *Progress in Theoretical Biology IV*, chapter Adaptation, pages 263–293. Academic Press, 1976.

[116] J. H. Holland. Adaptive algorithms for discovering and using general patterns in growing knowledge-bases. *International Journal of Policy Analysis and Information Systems*, 4:217–240, 1980.

[117] John H. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM)*, 9(3):297–314, 1962.

[118] John H. Holland, Lashon B. Booker, Marco Colombetti, Marco Dorigo, David E. Goldberg, Stephanie Forrest, Rick L. Riolo, Robert E. Smith, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson. *Learning classifier systems: from foundations to applications*, chapter What is a learning classifier system?, pages 3–32. Springer, 2000.

[119] John H. Holland and Judith S. Reitman. Cognitive systems based on adaptive algorithms. *ACM SIGART Bulletin*, 63:49, 1977.

[120] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[121] R. B. Hollstien. *Artificial genetic adaptation in computer control systems*. PhD thesis, The University of Michigan, 1971.

[122] J. H. Holmes, P. L. Lanzi, W. Stolzmann, and S. W. Wilson. Learning classifier systems: New models, successful applications. *Information Processing Letters*, 82:23–30, 2002.

[123] R. Horst, P. M. Pardalos, and N. V. Thoai. *Introduction to Global Optimization*. Kluwer Academic Publishers, 2nd edition, 2000.

[124] Julian Huxley. *Evolution: The Modern Synthesis*. Allen & Unwin, 1942.

[125] D. S. Johnson. Local optimization and the travelling salesman problem. In *Proceedings of the 17th Colloquium on Automata, Languages, and Programming*, pages 446–461, 1990.

[126] D. S. Johnson and L. A. McGeoch. *Local Search in Combinatorial Optimization*, chapter The travelling salesman problem: A case study in local optimization, pages 215–310. John Wiley & Sons, 1997.

[127] Kenneth A. De Jong. Genetic algorithms are NOT function optimizers. In *Proceedings of the Second Workshop on Foundations of Genetic Algorithms*, pages 5–17. Morgan Kaufmann, 1992.

[128] Kenneth A. De Jong. *Evolutionary computation: a unified approach*. MIT Press, 2006.

[129] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan Ann Arbor, MI, USA, 1975.

[130] Kenneth De Jong. Learning with genetic algorithms: An overview. *Machine Learning*, 3:121–138, 1988.

[131] Daniel Joseph Cavicchio Jr. *Adaptive Search Using Simulated Evolution*. PhD thesis, The University of Michigan, 1970.

[132] A. Juels and M. Wattenberg. Stochastic hill climbing as a baseline method for evaluating genetic algorithms. Technical report, University of California, Berkeley, 1994.

[133] Dean C. Karnopp. Random search techniques for optimization problems. *Automatica*, 1(2–3):111–121, 1963.

[134] Jürgen Klockgether and Hans-Paul Schwefel. Two–phase nozzle and hollow core jet experiments. In *Proc. Eleventh Symp. Engineering Aspects of Magnetohydrodynamics*, pages 141–148. California Institute of Technology, 1970.

[135] John Knox. Tabu search performance on the symmetric traveling salesman problem. *Computers & Operations Research*, 21(8):867–876, 1994.

[136] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, 1989.

[137] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, 1992.

[138] John R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT Press, 1994.

[139] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. *Genetic programming III: darwinian invention and problem solving*. Morgan Kaufmann, 1999.

[140] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic programming IV: routine human-competitive machine intelligence*. Springer, 2003.

[141] John R. Koza and Riccardo Poli. *Introductory Tutorials in Optimization, Search and Decision Support*, chapter 8: A Genetic Programming Tutorial. 2003.

[142] John R. Koza and Riccardo Poli. *Search Methodologies*, chapter 5: Genetic Programming; John Koza and Riccardo Poli. 2005.

[143] J. Kregting and R. C. White. Adaptive random search. Technical Report TH-Report 71-E-24, Eindhoven University of Technology, Eindhoven, Netherlands, 1971.

[144] M. Kudo and J. Sklansky. Comparison of algorithms that select features for pattern classifiers. *Pattern Recognition*, 33:25–41, 2000.

[145] O. Yu. Kul'chitskii. Random-search algorithm for extrema in functional space under conditions of partial uncertainty. *Cybernetics and Systems Analysis*, 12(5):794–801, 1976.

[146] Manuel Laguna and Rafael Martí. *Scatter search: methodology and implementations in C*. Kluwer Academic Publishers, 2003.

[147] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[148] L. T. Lau. *Guided Genetic Algorithm*. PhD thesis, Department of Computer Science, University of Essex, 1999.

[149] T.L. Lau and E.P.K. Tsang. The guided genetic algorithm and its application to the general assignment problems. In *IEEE 10th International Conference on Tools with Artificial Intelligence (ICTAI'98)*, 1998.

[150] R. M. Lewis, V. T., and M. W. Trosset. Direct search methods: then and now. *Journal of Computational and Applied Mathematics*, 124:191–207, 2000.

[151] H. R. Lourenco, O. Martin, and T. Stützle. A beginners introduction to iterated local search. In *Proceedings 4th Metaheuristics International Conference (MIC2001)*, 2001.

[152] George F. Luger and William A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Benjamin/Cummings Pub. Co., second edition, 1993.

[153] Sean Luke. *Essentials of Metaheuristics*. (self-published) http://cs.gmu.edu/∼sean/book/metaheuristics, 2009.

[154] P. Marrow. Nature-inspired computing technology and applications. *BT Technology Journal*, 18(4):13–23, 2000.

[155] Rafael Martí, Manuel Lagunab, and Fred Glover. Principles of scatter search. *European Journal of Operational Research*, 169(1):359–372, 2006.

[156] O. Martin and S. W. Otto. Combining simulated annealing with local search heuristics. *Annals of Operations Research*, 63:57–75, 1996.

[157] O. Martin, S. W. Otto, and E. W. Felten. Large-step markov chains for the traveling salesman problems. *Complex Systems*, 5(3):299–326, 1991.

[158] S. F. Masri, G. A. Bekey, and F. B. Safford. Global optimization algorithm using adaptive random search. *Applied Mathematics and Computation*, 7(4):353–376, 1980.

[159] Zbigniew Michalewicz and David B. Fogel. *How to solve it: modern heuristics*. Springer, 2004.

[160] P. Mills. *Extensions to Guided Local Search*. PhD thesis, Department of Computer Science, University of Essex, 2002.

[161] Patrick Mills, Edward Tsang, and John Ford. Applying an extended guided local search on the quadratic assignment problem. *Annals of Operations Research*, 118:121–135, 2003.

[162] M. Mitchell. Genetic algorithms: An overview. *Complexity*, 1(1):31–39, 1995.

[163] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[164] Melanie Mitchell and John H. Holland. When will a genetic algorithm outperform hill climbing? In *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1993.

[165] N. Mladenovic. A variable neighborhood algorithm - a new metaheuristic for combinatorial optimization. In *Abstracts of papers presented at Optimization Days*, 1995.

[166] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[167] Heinz Muhlenbein. Evolution in time and space - the parallel genetic algorithm. In *Foundations of Genetic Algorithms*, 1991.

[168] Heinz Muhlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In *Parallel Problem Solving from Nature 2*, pages 15–26, 1992.

[169] M. O'Neill and C. Ryan. Grammatical evolution: A steady state approach. In *Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms*, pages 419–423, 1998.

[170] M. O'Neill and C. Ryan. Grammatical evolution: A steady state approach. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, 1998.

[171] M. O'Neill and C. Ryan. Under the hood of grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999.

[172] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer, 2003.

[173] Michael ONeill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[174] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Courier Dover Publications, 1998.

[175] P.M. Pardalos, L.S. Pitsoulis, and M.G.C. Resende. A parallel grasp implementation for the quadratic assignment problem. In *Parallel Algorithms for Irregularly Structured Problems (Irregular94)*, pages 111–130. Kluwer Academic Publishers, 1995.

[176] R. Paton. *Computing With Biological Metaphors*, chapter Introduction to computing with biological metaphors, pages 1–8. Chapman & Hall, 1994.

[177] G. Paun. Bio-inspired computing paradigms (natural computing). *Unconventional Programming Paradigms*, 3566:155–160, 2005.

[178] W. Pedrycz. *Computational Intelligence: An Introduction*. CRC Press, 1997.

[179] T. Perkis. Stack-based genetic programming. In *Proc IEEE Congress on Computational Intelligence*, 1994.

[180] L. Pitsoulis and M. G. C. Resende. *Handbook of Applied Optimization*, chapter Greedy randomized adaptive search procedures, pages 168–181. Oxford University Press, 2002.

[181] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Programmers Guide to Genetic Programming*. Lulu Enterprises, 2008.

[182] V. William Porto. *Evolutionary Computation 1: Basic Algorithms and Operations*, chapter 10: Evolutionary Programming, pages 89–102. IoP Press, 2000.

[183] M. Prais and C.C. Ribeiro. Reactive grasp: An application to a matrix decomposition problem in tdma traffic assignment. *INFORMS Journal on Computing*, 12:164–176, 2000.

[184] K. Price and R. Storn. Differential evolution: Numerical optimization made easy. *Dr. Dobb's Journal*, pages 18–24, 1997.

[185] Kenneth V. Price. *New Ideas in Optimization*, chapter An introduction to differential evolution, pages 79–108. McGraw-Hill Ltd., UK, 1999.

[186] Kenneth V. Price, Rainer M. Storn, and Jouni A. Lampinen. *Differential evolution: a practical approach to global optimization*. Springer, 2005.

[187] Helena Ramalhinho-Loureno, Olivier C. Martin, and Thomas Stützle. *Handbook of Metaheuristics*, chapter Iterated Local Search, pages 320–353. Springer, 2003.

[188] L. A. Rastrigin. The convergence of the random search method in the extremal control of a many parameter system. *Automation and Remote Control*, 24:1337–1342, 1963.

[189] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog Verlag, 1973.

[190] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technical University of Berlin, Department of Process Engineering, 1971.

[191] M. G. C. Resende and C. C. Ribeiro. *Handbook of Metaheuristics*, chapter Greedy randomized adaptive search procedures, pages 219–249. Kluwer Academic Publishers, 2003.

[192] H. Robbins. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.*, 58:527–535, 1952.

[193] Richard Rosenberg. *Simulation of genetic populations with biochemical properties*. PhD thesis, University of Michigan, 1967.

[194] Günter Rudolph. *Evolutionary Computation 1: Basic Algorithms and Operations*, chapter 9: Evolution Strategies, pages 81–88. IoP Press, 2000.

[195] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition, 2009.

[196] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming*, 1998.

[197] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Solving trigonometric identities. In *Proceedings of Mendel 1998: 4th International Mendel Conference on Genetic Algorithms, Optimisation Problems, Fuzzy Logic, Neural Networks, Rough Sets.*, pages 111–119, 1998.

[198] Gnther Schrack and Mark Choit. Optimized relative step size random searches. *Mathematical Programming*, 10(1):230–244, 1976.

[199] M. Schumer and K. Steiglitz. Adaptive step size random search. *IEEE Transactions on Automatic Control*, 13(3):270–276, 1968.

[200] H. P. Schwefel. *Numerische Optimierung von Computer – Modellen mittels der Evolutionsstrategie*. Birkhaeuser, 1977.

[201] Hans-Paul Schwefel. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Technical University of Berlin, Department of Process Engineering, 1975.

[202] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, 1981.

[203] A.V. Sebald and D.B. Fogel. Design of SLAYR neural networks using evolutionary programming. In *Proceedings. of the 24th Asilomar Conf. on Signals, Systems and Computers*, pages 1020–1024, 1990.

[204] Yuhui Shi, editor. *Swarm intelligence*. Morgan Kaufmann, 2001.

[205] David B. Skalak. Prototype and feature selection by sampling and random mutation hill climbing algorithms. In *Proceedings of the eleventh international conference on machine learning*, pages 293–301. Morgan Kaufmann, 1994.

[206] A. Sloman. *Evolving Knowledge in Natural Science and Artificial Intelligence*, chapter Must intelligent systems be scruffy? Pitman, 1990.

[207] S. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proceedings 8th International Joint Conference on Artificial Intelligence*, pages 422–425, 1983.

[208] Stephen Frederick Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, Department of Computer Science, University of Pittsburgh, 1980.

[209] F. J.. Solis and J. B. Wets. Minimization by random search techniques. *Mathematics of Operations Research*, 6:19–30, 1981.

[210] James C. Spall. *Introduction to stochastic search and optimization: estimation, simulation, and control.* John Wiley and Sons, 2003.

[211] James C. Spall. *Handbook of computational statistics: concepts and methods*, chapter 6. Stochastic Optimization, pages 169–198. Springer, 2004.

[212] N. Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[213] R. Storn. Differential evolution design of an iir-filter. In *Proceedings IEEE Conference Evolutionary Computation*, pages 268–273. IEEE, 1996.

[214] R. Storn. On the usage of differential evolution for function optimization. In *Proceedings Fuzzy Information Processing Society, 1996 Biennial Conference of the North American*, pages 519–523, 1996.

[215] R. Storn and K. Price. Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, International Computer Science Institute, Berkeley, CA, 1995.

[216] R. Storn and K. Price. Minimizing the real functions of the icec'96 contest by differential evolution. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 842–844. IEEE, 1996.

[217] R. Storn and K. Price. Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

[218] T. Stützle. Applying iterated local search to the permutation flow shop problem. Technical Report AIDA9804, FG Intellektik, TU Darmstadt, 1998.

[219] T. Stützle. Iterated local search for the quadratic assignment problem. Technical Report AIDA-99-03, FG Intellektik, FB Informatik, TU Darmstadt, 1999.

[220] Thomas Stützle and Holger H. Hoos. Analyzing the run-time behaviour of iterated local search for the tsp. In *Proceedings III Metaheuristics International Conference*, 1999.

[221] Thomas G. Stützle. *Local Search Algorithms for Combinatorial Problems: Analysis, Improvements, and New Applications.* PhD thesis, Darmstadt University of Technology, Department of Computer Science, 1998.

[222] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation.* John Wiley and Sons, 2009.

[223] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide.* Pragmatic Bookshelf, second edition, 2004.

[224] E. P. K Tsang and C. J. Wang. A generic neural network approach for constraint satisfaction problems. In Taylor G, editor, *Neural network applications*, pages 12–22, 1992.

[225] A. Trn, M.M. Ali, and S. Viitanen. Stochastic global optimization: Problem classes and solution techniques. *Journal of Global Optimization*, 14:437–447, 1999.

[226] C Voudouris. *Guided local search for combinatorial optimisation problems.* PhD thesis, Department of Computer Science, University of Essex, Colchester, UK, July 1997.

[227] C. Voudouris and E. P. K. Tsang. *Handbook of metaheuristics*, chapter 7: Guided Local Search, pages 185–218. Springer, 2003.

[228] C. Voudouris and E.P.K. Tsang. Guided local search joins the elite in discrete optimisation. In *Proceedings, DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimisation*, 1998.

[229] Chris Voudouris and Edward Tsang. The tunneling algorithm for partial csps and combinatorial optimization problems. Technical Report CSM-213, Department of Computer Science, University of Essex, Colchester, C04 3SQ, UK, 1994.

[230] Chris Voudouris and Edward Tsang. Function optimization using guided local search. Technical Report CSM-249, Department of Computer Science University of Essex Colchester, CO4 3SQ, UK, 1995.

[231] Chris Voudouris and Edward Tsang. Guided local search. Technical Report CSM-247, Department of Computer Science, University of Essex, Colchester, C04 3SQ, UK, 1995.

[232] Edward Tsang & Chris Voudouris. Fast local search and guided local search and their application to british telecoms workforce scheduling problem. Technical Report CSM-246, Department of Computer Science University of Essex Colchester CO4 3SQ, 1995.

[233] C. J. Wang and E. P. K. Tsang. Solving constraint satisfaction problems using neural networks. In *Proc. Second International Conference on Artificial Neural Networks*, pages 295–299, November 18–20, 1991.

[234] Thomas Weise. *Global Optimization Algorithms - Theory and Application.* Thomas Weise, 2009-06-26 edition, 2007.

[235] R. C. White. A survey of random methods for parameter optimization. *Simulation*, 17(1):197–205, 1971.

[236] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[237] S. W. Wilson. Zcs: A zeroth level classifier systems. *Evolutionary Computation*, 2:1–18, 1994.

[238] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3:149–175, 1995.

[239] S. W. Wilson. Generalization in the xcs classifier system. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998.

[240] Stewart W. Wilson and David E. Goldberg. A critical review of classifier systems. In *Proceedings of the third international conference on Genetic algorithms*, pages 244–255, 1989.

[241] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical report, Santa Fe Institute, Sante Fe, NM, USA, 1995.

[242] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(67):67–82, 1997.

[243] Xin Yao and Yong Liu. Fast evolution strategies. In *Proceedings of the 6th International Conference on Evolutionary Programming VI*, pages 151–162, 1997.

[244] Xin Yao, Yong Liu, and Guangming Lin. Evolutionary programming made faster. *IEEE Transactions on Evolutionary Computation*, 3(2):82–102, 1999.

[245] Z. B. Zabinsky. *Stochastic adaptive search for global optimization*. Kluwer Academic Publishers, 2003.

[246] Lotfi Asker Zadeh, George J. Klir, and Bo Yuan. *Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers*. World Scientific, 1996.

[247] A. A. Zhigljavsky. *Theory of Global Random Search*. Kluwer Academic, 1991.

[248] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Shaker Verlag, Aachen, Germany, 1999.

[249] E. Zitzler, M. Laumanns, and S. Bleuler. *Metaheuristics for Multiobjective Optimisation*, chapter A Tutorial on Evolutionary Multiobjective Optimization, pages 3–37. Springer, 2004.

[250] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland, May 2001.

[251] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. *Evolutionary Methods for Design Optimisation and Control*, pages 95–100, 2002.

[252] E. Zitzler and L. Thiele. An evolutionary algorithm for multiobjective optimization: The strength pareto approach. Technical Report 43, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland, May 1998.

[253] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.