

Chapter 2

Heuristic Optimization

2.1 Introduction

2.1.1 The Problems with Optimization Problems

Optimization problems are concerned with finding the values for one or several decision variables that meet the objective(s) the best without violating the constraint(s). The identification of an efficient portfolio in the Markowitz model (1.7) on page 7 is therefore a typical optimization problem: the values for the decision variables x_i have to be found under the constraints that (i) they must not exceed certain bounds ((1.7f): $0 \leq x_i \leq 1$ and (1.7e): $\sum_i x_i = 1$) and (ii) the portfolio return must have a given expected value (constraint (1.7c)); the objective is to find values for the assets' weights that minimize the risk which is computed in a predefined way. If there are several concurring objectives, usually a trade-off between them has to be defined: In the modified objective function (1.7a*) on page 9, the objectives of minimizing the risk while maximizing the return are considered simultaneously.

The Markowitz model is a well-defined optimization model as the relationship between weight structure and risk and return is perfectly computable for any valid set of (exogenously determined) parameters for the assets' expected returns and (co-)variances (as well as, when applicable, the trade-off factor between portfolio risk and return). Nonetheless, there exists no general solution for this optimization problem because of the non-negativity constraint on the asset weights. Hence, there is no closed form solution as there is for the Black model (which is equal to the

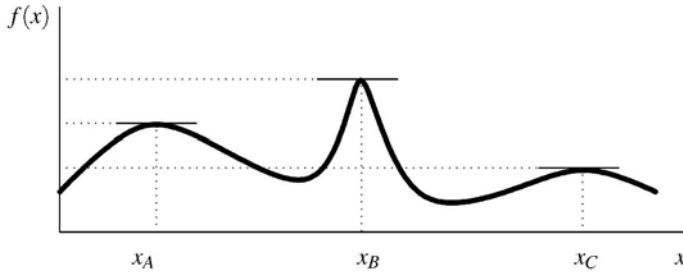


Fig. 2.1: Global and local optima

Markowitz model except for the non-negativity constraint). Though not solvable analytically, there exist numerical procedures by which the Markowitz model can be solved for a given set of parameters values.

Depending on the objective function, optimization problems might have multiple solutions some of which might be local optima. In Figure 2.1, e.g., a function $f(x)$ is depicted, and the objective might be to find the value for x where $f(x)$ reaches its highest value, i.e., $\max_x f(x)$. As can easily be seen, all three points x_A , x_B , and x_C are (local) maxima: the first order condition, $f'(x) = 0$, is satisfied (indicated by the horizontal tangency lines), and any slight increase or decrease of x would decrease the function's value: $f(x) \geq f(x \pm \varepsilon)|_{\varepsilon \rightarrow 0}$. Nonetheless, only x_B is a *global optimum* as it yields the highest overall value for the objective function, whereas x_A and x_C are just *local optima*. Unlike for this simple example, however, it is often difficult to determine whether an identified solution is a local or the global optimum as the solution space is too complex: All of the objective functions that will be considered in the main part of this contribution have more than one decision variable, the problem space is therefore multidimensional; and the objective functions are mostly discontinuous (i.e., the first derivatives are not well behaved or do not even exist).

In portfolio management, these difficulties with the objective functions are frequently observed when market frictions have to be considered. To find solutions anyway, common ways of dealing with them would be to either eliminate these frictions (leading to models that represent the real-world in a stylized and simplified way) or to approach them with inappropriate methods (which might lead to sub-optimal and misleading results without being able to recognize these errors). This

contribution is mainly concerned with the effects of market frictions on financial management which are therefore explicitly taken into account. Hence, for reliable results, an alternative class of optimization techniques has to be employed that are capable of dealing with these frictions, namely *heuristic optimization* techniques.

Opposed to the well-defined problems considered so far, there also exist problems where the underlying structure is unknown, partially hidden – or simply too complex to be modeled. When an underlying structure can be assumed or when there are pairs of input/output data, these questions can be approached, e.g., with *econometric*¹ or *Artificial Intelligence*² methods. In finance, time series analysis, pricing of complex securities, model selection problems, and artificial markets would be typical examples.³ In this contribution, however, only well-defined optimization problems will be considered.

2.1.2 Techniques for Hard Optimization Problems

2.1.2.1 Measuring Computational Complexity

Before introducing specific optimization methods, it might be helpful to find a classification for the size and complexity of the considered problems – and, in due course, a measure of the methods applied on them. The computational complexity of an optimization problem as well as optimization procedures (and algorithms in general) is frequently given in $\mathcal{O}(\cdot)$ notation which indicates the asymptotic time necessary to solve the problem when it involves n (instances of the) decision variables and the problem size is determined by the number of these decision variables. An algorithm of order $\mathcal{O}(n)$, e.g., will consume a processor time (CPU time) of $n \cdot c$, i.e., the time necessary for solving the problem increases linearly in the number of instances;⁴ a polynomial algorithm of order $\mathcal{O}(n^k)$ will consume $c \cdot n^k$ where k is

¹ See, e.g., Winker (2001) or Gourieroux and Jasiak (2001).

² See, e.g., Russell and Norvig (2003).

³ See, e.g., Winker and Gilli (2004), Kontoghiorghe, Rustem, and Siokos (2002), Rasmussen, Goldy, and Solli (2002) or LeBaron (1999).

⁴ Note that these considerations exclude aspects such as memory management or time consumed by interface communication. Practical implementations should account for these machine and programming environment depending characteristics.

a constant specific for the problem or algorithm. E.g., reading n pages of a book might consume linear time (i.e., $\mathcal{O}(n)$), but finding out whether there are any duplicates in a pile of n pages demands that each of them is compared to the remaining $(n - 1)$ pages, and the complexity becomes $\mathcal{O}(n \cdot (n - 1)) \approx \mathcal{O}(n^2)$ and is therefore quadratic in n .

The constant c will differ across programming platforms as well as actual CPU capacities. For sufficiently large n , the main contribution to computational time will come from the argument related to n or, if the argument consists of several components, the “worst” of them, i.e., the one that eventually outgrows the other: When the complexity is $\mathcal{O}(\ln(n) \cdot (n/k)^2)$ then, for any constant k and sufficiently large n , the quadratic term will outweigh the logarithmic term and the CPU time can be considered quadratic in n . A sole increase in CPU power which implies a reduction in c will therefore not have a sustainable effect on the (general) computability of a problem. A real improvement of the complexity can only be achieved when a better algorithm is found. This applies not only for the (sometimes called “easy”) polynomial problems where the exponent k can become quite large, too; it is all the more true for a special class of optimization and search problems: For the group of *non-deterministic polynomial time (NP) complete* problems, there is no deterministic algorithm known that can find an exact solution within polynomial time.⁵ This means that for deciding whether the found solution is optimal, the number of necessary steps is not a polynomial, but (at least) an exponential function of the problem size in the worst case. Two well-known problems of this group are the *Traveling Salesman Problem (TSP)*⁶ and the *Knapsack Problem (KP)*⁷. One of the difficulties with optimization

⁵ Actually, it can be shown that if one of these problems could be solved in polynomial time, this solution could be transferred and all of these problems could be solved in polynomial time.

⁶ In the TSP, a salesperson has to find the shortest route for traveling to n different cities, usually without visiting one city twice. The problem can be modeled by a graph where nodes are the cities and the arcs are the distances between any two cities. Finding the shortest route corresponds then to finding the shortest path through this graph (or the shortest *Hamiltonian cycle*, if the salesperson takes a round trip where the tour starts and ends in the same city). If the graph is fully connected, i.e., there is a direct connection between any two cities, there are $n!$ alternative routes to choose from – which also characterizes the worst case computational complexity of this problem.

⁷ In the KP, a tourist finds a number of precious stones that differ in size and in value per size unit. As the capacity of her knapsack is limited, the task of this 1/0 knapsack problem is to select stones such that the value of the contents is maximized. Fast decisions to this problem are possible only under rare circumstances. See Kellerer, Pferschy, and Pisinger (2004) and section 4.2.1.

problems is that the complexity for solving them is not always apparent: Proofs that an optimization problem belongs to a certain complexity class are therefore helpful when deciding which solution strategy ought to be considered.⁸

2.1.2.2 Brute-Force Methods

Analytic, closed-form solutions to optimization problems are desirable as they allow for exact and straightforward answers. The major advantage of these solutions is that they can be derived without explicit knowledge of the included parameters' values: The optimization has to be done only once, and the result is in a form by which, given the relevant (exogenous) parameters, the optimal values for the decision variables can immediately be determined.

If such solutions do not exist, then the problem has usually to be solved for each individual set of parameters. The approach that needs the least optimization skills would be *complete enumeration* where simply all possible (and valid) values for the decision variables are tested. This approach has some severe downsides: first and foremost, it is frequently time-consuming far beyond acceptability. Second, it demands the set of candidate solutions to be discrete and finite; if the decision variables are continuous (i.e., have infinitely many alternatives) then they have to be discretized, i.e., transformed into a countable number of alternatives – ensuring that the actual optimum is not excluded due to too large steps while keeping the resulting number of alternatives manageable.

In chapter 4, e.g., the problem will be to select k out of N assets and optimize their weights. The problem size can quickly get out of hand: there are $\binom{N}{k} = N! / ((N-k)! \cdot k!)$ alternative combinations for selecting k out of N assets without optimizing the weights; correspondingly, the complexity of an exhaustive search would be $\mathcal{O}\left(\binom{N}{k}\right)$. Selecting just 10 out of 100 assets comes with $\binom{100}{10} = 1.73 \times 10^{13}$ alternatives. For each of these alternatives, the optimal weights had to be found: When the weights of 10 assets may be either zero or multiples of 10% and short-sales are disallowed, the granularity of the weights is $g = 1/10\%$ which comes with $k^g = 10^{10}$ possible weight structures per alternative; the complexity is then increased to $\mathcal{O}\left(\binom{N}{k} \cdot k^g\right)$. Having

⁸ See Knuth (1997) and Harel (1993).

a computer that is able to evaluate a million cases per second, complete enumeration would take 1.32×10^{11} years – which is approximately ten times the time since the Big Bang. When k is increased by just one additional asset from 10 to 11 (other things equal), the CPU time would increase to 233 times the time since the Big Bang; and if, in addition, the granularity would be increased to multiples of 5% (which would still be too rough for realistic applications), then the CPU time increased to more than 6 trillion times since the Big Bang.

Some of the problems dealt with in the following chapters have opportunity sets that are magnitudes larger; complete enumeration is therefore not a realistic alternative, nor would a sheer increase in computational power (by faster CPU's or having parallel computers) do the trick.

2.1.2.3 Traditional Numerical Methods and Algorithmic Approaches

Traditional numerical methods are usually based on iterative search algorithms that start with a (deterministic or arbitrary) solution which is iteratively improved according to some deterministic rule.⁹ For financial optimization, methods from *Mathematical Programming* are often applied as these methods can manage problems where the constraints contain not only equalities, but also inequalities. Which type of method should and can be applied depends largely on the type of problem:¹⁰

Linear Programming will be applied when the optimization problem has a linear objective function and its constraints, too, are all linear (in-)equalities. The most popular method is the *Simplex Algorithm* where first the inequalities are transformed into equalities by adding *slack variables* and then including and excluding *base variables* until the optimum is found. Though its worst case computational complexity is exponential, it is found to work quite efficiently

⁹ For a concise presentation of applications in economics, see Judd (1998).

¹⁰ The following list of methods is far from exhaustive. For more details, see, e.g., Hillier and Lieberman (2003) for a concise introduction to Operations Research, and Stepan and Fischer (2001) for quantitative methods in Business Administration. Hillier and Lieberman (1995) presents methods in mathematical programming, a presentation of major algorithmic concepts can be found in Knuth (1997). Seydel (2002) and Brandimarte (2002) tackle several issues in computational finance and present suitable numerical methods.

for many instances. Some parametric models for portfolio selection therefore prefer linear risk measures (accepting that these risk measures have less desirable properties than the variance).

Quadratic and Concave Programming can be applied when the constraints are linear (in-)equalities, yet the objective function is quadratic. This is the case for the Markowitz model (1.7); how this can be done, will be presented in section 2.1.2.4. When the Kuhn-Tucker conditions hold,¹¹ a modified version of the Simplex Algorithm exists that is capable of solving these problems – the computational complexity of which, however, is also exponential in the worst case.

Dynamic Programming is a general concept rather than a strict algorithm and applies to problems that have, e.g., a temporal structure. For financial multi-temporal problems, the basic idea would be to split the problem into several sub-problems which are all myopic, i.e., have no temporal aspects when considered separately. First, the sub-problem for the last period, T , is solved. Next the optimal solution for last but one period, $T - 1$, is determined, that leads to the optimal solution for T , and so on until all sub-problems are solved.

Stochastic Programming is concerned with optimization problems where (some of the) data incorporated in the objective function are uncertain. Usual approaches include recourse, assumption of different scenarios and sensitivity analyses.

Other types of Mathematical Programming include non-linear programming, integer programming, binary programming and others. For some specimen types of problems, algorithms exist that (tend to) find good solutions. To approach the optimization problem at hand, it has to be brought into a structure for which the method is considered to work – which, for financial optimization problems, often comes with the introduction of strong restrictions or assumptions on the “allowed” values for the decision variables or constraints.

Greedy Algorithms always prefer the next one step that yields the maximum improvement but does not assess its consequences. Given a current (suboptimal) solution, a greedy algorithm would search for a modified solution within

¹¹ See, e.g., Chiang (1984, section 21.4).

a certain neighborhood and choose the “best” among them. This approach is sometimes called *hill-climbing*, referring to a mountaineer who will choose her every next step in a way that brings the utmost increase. As these algorithms are focused on the next step only, they get easily stuck when there are many local optima and the initial values are not chosen well. Hence, this approach demands smooth solution spaces and a monotonous objective function for good solutions and is related to the concept of gradient search.

Gradient Search can be performed when the objective function $f(x)$ is differentiable and strictly convex¹² and the optimum can be found with the first order condition $\partial f / \partial x = 0$. Given the current candidate solution, the gradient $\nabla f(x') = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$ is computed for $x' = x$. The solution is readjusted according to $x' = x' + \delta \cdot \nabla f(x)$ which corresponds to $x'_j := x'_j + \delta \cdot \frac{\partial f}{\partial x_j} \Big|_{x=x'} \forall j$. This readjustment is repeated until the optimum x^* with $\nabla f(x) = 0$ is reached. Graphically speaking, this procedure determines the tangency at point x' and moves the decision variables towards values for which the tangency's slope is expected to be 0 and any slight change of any x_j would worsen the value of the objective function f .

Divide and Conquer Algorithms iteratively split the problem into sub-problems until the sub-problems can be solved in reasonable time. These partial results are then merged for the solution of the complete problem. These approaches demand that the original problem can be partitioned in a way that the quality of the solutions for sub-problems will not interfere with each other, i.e., that the sub-problems are not interdependent.

Branch and Bound Algorithms can be employed in some instances where parts of the opportunity set and candidate solutions can be excluded by selection tests. The idea is to iteratively split the opportunity space into subsets and identify as soon as possible those subsets where the optimum is definitely not a member of, mainly by either repeatedly narrowing the boundaries within which the solution must fall, by excluding infeasible solutions, or by “pruning” those solutions that are already outperformed by some other solution

¹² Here, maximization problems are considered. For minimization problems, the similar arguments for concave functions can be considered. Note that any maximization problem can be transformed into a minimization problem (usually by taking the inverse of the objective function or multiplying it by -1) and *vice versa*.

found so far. The opportunity set is therefore repeatedly narrowed down until either a single valid solution is found or until the problem is manageable with other methods, such as complete enumeration of all remaining solutions or another numerical method.

As mentioned earlier, a salient characteristic of these methods is that they work only for problems which satisfy certain conditions: The objective function must be of a certain type, the constraints must be expressible in certain formats, and so forth. Their application is therefore restricted to a rather limited set of problems. In practice, these limitations are often circumvented by modifying the problems and stating the problems in a way that they are solvable. Another main caveat of these optimization methods is that they are mostly based on rather strict deterministic rules. Hence, they might produce wrong solutions when the considered problem has not just one global, but also one or several local optima. Once deterministic search rules converge towards such local optima, they might have problems leaving them again (and therefore will never find the global optimum), given they converge in the first place. Also, deterministic rules have it that, by definition, for a given situation, there is a unique response. A deterministic search algorithm will therefore always produce the same result for a given problem when the search strategy cannot be influenced and the initial values, too, are chosen deterministically. This being the standard case, repeated runs will always report the same local optimum, in particular when the initial value for the search process is found with some deterministic rule, too.

In the lack of alternatives, however, financial optimization problems have often been modeled in a way that they can be solved with one of these methods. As a consequence, they either had to be rather restrictive or had to accept strong simplifications (such as the assumption of frictionless markets in order to satisfy the Kuhn-Tucker conditions), or accepted that the solutions are likely to be suboptimal (such as, e.g., in *Asset Class Management*, where the universe of available assets is split into submarkets (subportfolios) which are optimized independently in a “divide and conquer” fashion, ignoring the relationships between the classes). However, without this fitting of the problems to the available methods, the majority of (theoretical and practical) optimization problems in portfolio management could not readily be answered. Due to the fitting, on the other hand, it is difficult (and quite often impossible) to tell whether a reported solution is unique or just one out of many optima and how far away this reported solution is from the global optimum.

2.1.2.4 Portfolio Optimization with Linear and Quadratic Programming

Many software packages for optimization problems offer routines and standard solutions for Linear and Quadratic Programming Problems. Linear Programming (LP) can be applied when the objective function is linear in the decision variable and the constraints are all equalities or inequalities that, too, are linear. A general statement would be

$$\min_x f'x$$

subject to

$$Ax = a$$

$$Bx \leq b$$

where x is the vector of decision variables and A , B , a , and b are matrices and vectors, respectively, that capture the constraints. Note that any minimization problem can be transformed into a maximization problem simply by changing the sign of the objective function (i.e., by multiplying f with -1) and that by choosing the appropriate signs, inequalities of the type $Cx \geq c$ can be turned into $-Cx \leq -c$, i.e., b can contain upper and lower limits alike.

A simple application in portfolio selection might be to find the weights $x_i, i = 1, \dots, N$, that maximize the portfolio's expected return when the weight of each of the N assets must be within a given range, i.e., $x^l \leq x_i \leq x^u$, and the weights must add up to one, i.e., $\sum_i x_i = 1$. This can be achieved by setting $f = -r$; $A = \mathbf{1}_{1 \times N}$, $a = 1$, $B = [-I_{N \times N} \quad I_{N \times N}]'$ and $b = [-x^l \mathbf{1}_{1 \times N} \quad x^u \mathbf{1}_{1 \times N}]'$ where r is the vector of expected returns and I and $\mathbf{1}$ are the respective identity matrices and unity vectors with the dimensions as indexed. This approach, however, is not able to cope with variance or volatility as these are quadratic risk measures.

Quadratic Programming (QP) problems, like LP problems, have only constraints that can be expressed as linear (in-)equalities with respect to the decision variables; their objective function, however, allows for an additional term that is quadratic in the decision variables. A standard general statement therefore might read as follows:

$$\min_x f'x + \frac{1}{2}x'Hx$$

subject to

$$Ax = a$$

$$Bx \leq b.$$

This can be applied to determine the Markowitz efficient portfolio for a return of r_P by implementing the model (1.7) as follows. If r and Σ denote the return vector and covariance matrix, respectively, then $f = \mathbf{0}_{1 \times N}$, $H = 2\Sigma$, $A = [\mathbf{1}_{N \times 1} \ r']'$, $a = [1 \ r_P]'$, $B = -I_{N \times N}$ and $b = \mathbf{0}_{N \times 1}$ where $\mathbf{0}$ is the zero vector.

If, however, the whole efficient line of the Markowitz model is to be identified, then the objective function (1.7a*) is to be applied and the respective parameters are $f = -\lambda r$, $H = 2(1 + \lambda)\Sigma$, $A = \mathbf{1}_{1 \times N}$, $a = 1$, $B = -I_{N \times N}$ and $b = \mathbf{0}_{N \times 1}$. Since λ measures the trade-off between risk and return, $\lambda = 0$ will lead to the identification of the Minimum Variance Portfolio. On the other hand, $\lambda = 1$ puts all the weight on the expected return and will therefore report the portfolio with the highest possible yield which, for the given model with non-negativity constraints but no upper limits on x_i , will contain exclusively the one asset with the highest expected return. To identify the efficient portfolios between these two extremes, a usual way would be to increase λ in sufficiently small steps from zero to one and solve the optimization problem for these values.

In a Tobin framework as presented in section 1.1.2.3 where the set of risky assets is supplemented with one safe asset, the investor will be best off when investing an amount α into the safe asset and the remainder of $(1 - \alpha)$ into the tangency portfolio \mathcal{T} . Given an exogenously chosen value $0 < \alpha < 1$ (for convenience, $\alpha \rightarrow 0$), the respective parameters for the quadratic programming model are $f = -[r' \ r_s]'$, $H = 2 \begin{bmatrix} \Sigma & \mathbf{0}_{N \times 1} \\ \mathbf{0}_{1 \times N} & 0 \end{bmatrix}$, $A = \begin{bmatrix} \mathbf{1}_{1 \times N} & 1 \\ \mathbf{0}_{1 \times N} & 1 \end{bmatrix}$, $a = [1 \ \alpha]'$, $B = -I_{(N+1) \times (N+1)}$ and $b = \mathbf{0}_{(N+1) \times 1}$. The resulting vector x is of dimension $(N + 1) \times 1$, where the first N elements represent $(1 - \alpha) \cdot x_{\mathcal{T}}$, whereas the $(N + 1)$ -st element is the weight of the safe asset in the investor's overall portfolio and (by constraint) has the value of α . By the separation theorem, the weights for \mathcal{T} can then be determined by $x_{\mathcal{T}} = \frac{1}{1-\alpha} [x_1 \ \dots \ x_N]'$.

2.1.2.5 “Traditional” Deterministic versus Stochastic and Heuristic Methods

Classical optimization techniques as presented so far can be divided into two main groups. The first group of methods is based on exhaustive search or (complete) enumeration, i.e., testing all candidate solutions. The crux of approaches like branch and bound is to truncate as much of the search space as possible and hence to eliminate groups of candidates that can be identified as inferior beforehand. However, even after pruning the search space, the remaining number of candidates might still exceed the available capacities, provided the number of solutions is discrete and finite in the first place.

The second type comprises techniques that are typically based on the differential calculus, i.e., they apply the first order conditions and push the decision variables towards values where the first derivative or gradient of the objective function is (presumably) zero. An implicit assumption is that there is just one optimum and/or that the optimum can be reached on a “direct path” from the starting point. The search process itself is usually based on deterministic numerical rules. This implies that, given the same initial values, repeated runs will always report the same result – which, as argued, is not necessarily a good thing: repeated runs with same (deterministically generated) initial values will report the same results, unable to judge whether the global or just a local optimum has been found. To illustrate this problem, reconsider the function depicted in Figure 2.1 on page 39. If the initial guess is a value for x that is near one of the local optima x_A or x_C , then a traditional numerical procedure is likely to end up at the local maximum closest to the initial guess, and the global optimum, x_B , will remain undiscovered. In practice, the deterministic behavior and the straightforward quest for the closest optimum from the current solutions perspective can be a serious problem, in particular when there are many local optima which are “far apart” from the global optimum, but close to the starting value. Also, slight improvements in the objective function might come with substantially different values for the decision variables.

One radical alternative to deterministic methods would be *Monte Carlo (MC) search*: A large number of random (yet valid with respect to the constraints) guesses for values for the decision variables are generated and the respective values of the

objective function are determined.¹³ With a sufficiently large number of independent guesses, this approach is likely to eventually identify the optimum or at least to identify regions within which it is likely or unlikely to be found. This concept is much more flexible than numerical methods as its main restrictions are *a priori* the availability of a suitable random number generator and the time necessary to perform a sufficiently large number of tries. It can therefore be applied to narrow down the search space which could then be approached with numerical methods. The major downside of it is, however, that it might be quite inefficient and inexact: Quite often, significant parts of the opportunity set can quickly be identified as far from the actual optimum; further search in this “region” is therefore just time consuming.

Heuristic search methods and *heuristic optimization techniques* also incorporate stochastic elements. Unlike Monte Carlo search, however, they have mechanisms that drive the search towards promising regions of the opportunity space. They therefore combine the advantages of the previously presented approaches: much like numerical methods, they aim to converge to the optimum in course of iterated search, yet they are less likely to end up in a local optimum and, above all, are very flexible and therefore are less restricted (or even perfectly unrestricted) to certain forms of constraints.

The heuristics discussed in due course and applied in the main part of this contribution were designed to solve optimization problems by repeatedly generating and testing new solutions. These techniques therefore address problems where there actually exist a well-defined model and objective function. If this is not the case, there exist alternative methods in *soft computing*¹⁴ and *computational intelligence*¹⁵.

¹³ It is not always possible to guarantee beforehand that none of the constraints is violated; also, ascertaining that only valid candidate solutions are generated might be computationally costly. In these cases, a simple measure would be not to care about these constraints when generating the candidate solutions but to add a *punishment term* to the objective value when this candidate turns out to be invalid.

¹⁴ Coined by the inventor of *fuzzy logic*, Lotfi A. Zadeh, the term *soft computing* refers to methods and procedures that not only tolerate uncertainty, fuzziness, imprecision and partial correctness but also make use of them; see, e.g., Zadeh and Garibaldi (2004).

¹⁵ Introduced by James Bezdek, *computational intelligence* refers to methods that use numerical procedures to simulate intelligent behavior; see Bezdek (1992, 1994).

A popular method of this type are *Neural Networks* which mimic the natural brain process while learning by a non-linear regression of input-output data.¹⁶

2.2 Heuristic Optimization Techniques

2.2.1 Underlying Concepts

The toy manufacturer Hasbro, Inc., produces a popular game called *Mastermind*. The rules of this game are rather simple: one player selects four colored pegs and the other player has to guess their color and sequence within a limited number of trials. After each guess, the second player is told how many of the guessed pegs are of the right color and how many are the right color and in the right position. The problem is therefore well-defined, as there are a clear objective function and a well-defined underlying model: though the “parameters” of the latter are hidden to the second player, it produces a uniquely defined feedback for any possible guess within a game.

Although there are 360 different combinations in the standard case¹⁷ the second player is supposed to find the right solution within eight guesses or less. Complete enumeration is therefore not possible. The typical beginner’s approach is to perform a Monte Carlo search by trying several perfectly random guesses (or the other player’s favorite colors) and hoping to find the solution either by sheer chance or by eventually interpreting the outcome of the independent guesses. With unlimited guesses, this strategy will eventually find the solution; when limited to just eight guesses, the hit rate is disappointingly low.

More advanced players also start off with a perfectly random guess, but they reduce the “degree of randomness” in the subsequent guesses by considering the outcomes from the previous guesses: E.g., when the previous guess brought two white

¹⁶ See Russell and Norvig (2003) for a general presentation; applications to time series forecasting are presented in Azoff (1994).

¹⁷ The standard case demands all four pegs to be of different color with six colors to choose from. Alternative versions allow for “holes” in the structure, repeated colors and/or also the “white” and “black” pegs, used to indicate “correct color” and “correct color and position”, respectively – resulting in up to 6 561 combinations.

pegs (i.e., only two right colors, none in the right position, and two wrong colors), the next guess should contain some variation in the color; if the answer were four white pegs (i.e., all the colors are right, yet all in the wrong position), the player can concentrate on the order of the previously used pegs rather than experimenting with new colors. The individual guesses are therefore not necessarily independent, yet (usually) there is no deterministic rule for how to make the next guess. *Mastermind* might therefore serve as an example where the solution to a problem can be found quite efficiently by applying an appropriate *heuristic optimization method*.

2.2.2 Characteristics of Heuristic Optimization Methods

The central common feature of all *heuristic optimization (HO)* methods is that they start off with a more or less arbitrary initial solution, iteratively produce new solutions by some generation rule and evaluate these new solutions, and eventually report the best solution found during the search process. The execution of the iterated search procedure is usually halted when there has been no further improvement over a given number of iterations (or further improvements cannot be expected); when the found solution is good enough; when the allowed CPU time (or other external limit) has been reached; or when some internal parameter terminates the algorithm's execution. Another obvious halting condition would be exhaustion of valid candidate solutions – a case hardly ever realized in practice.

Since HO methods may differ substantially in their underlying concepts, a general classification scheme is difficult to find. Nonetheless, the following list highlights some central aspects that allow for comparisons between the methods.¹⁸ With the rapidly increasing number of new heuristics and variants or combinations of already existing ones, the following list and the examples given therein are far from exhaustive.

Generation of new solutions. A new solution can be generated by modifying the current solution (*neighborhood search*) or by building a new solution based on past experience or results. In doing so, a deterministic rule, a random guess or a combination of both (e.g., deterministically generating a number of alternatives and randomly selecting one of them) can be employed.

¹⁸ For an alternative classification, see, e.g., Silver (2002) and Winker and Gilli (2004).

Treatment of new solutions. In order to overcome local optima, HO methods usually consider not only those new solutions that lead to an immediate improvement, but also some of those that are knowingly inferior to the best solution found so far. To enforce convergence, however, inferior solutions might either be included only when not being too far from the known optimum or might be given a smaller “weight.” Also, the best found solution so far might be reinforced (*elitist principle*), new solutions might be ranked and only the best of them are kept for future consideration, etc. The underlying acceptance rules can be deterministic or contain certain randomness.

Number of search agents. Whereas in some methods, a single agent aims to improve her solution, population based methods often make use of collective knowledge gathered in past iterations.

Limitations of the search space. Given the usually vast search space, new solutions can be found by searching within a certain neighborhood of a search agent’s current solution or of what the population (implicitly) considers promising. Some methods, on the other hand explicitly exclude certain neighborhoods or regions to avoid cyclic search paths or spending too much computation time on supposedly irrelevant alternatives.

Prior knowledge. When there exist general guidelines of what is likely to make a good solution, this prior knowledge can be incorporated in the choice of the initial solutions or in the search process (*guided search*). Though the inclusion of prior knowledge might significantly reduce the search space and increase the convergence speed, it might also lead to inferior solutions as the search might get guided in the wrong direction or the algorithm might have severe problems in overcoming local optima. Prior knowledge is therefore found in a rather limited number of HO methods and there, too, rather an option than a prerequisite.

Flexibility for specific constraints. Whereas there exist true general purpose methods that can be applied to virtually any type of optimization problem, some methods are tailor-made to particular types of constraints and are therefore difficult to apply to other classes of optimization problems.

Other aspects allow for testing and ranking different algorithms and might also affect the decision which method to select for a particular optimization problem:

Ease of implementation. The (in-)flexibility of the concept, the complexity of the necessary steps within an iteration step, the number of parameters and the time necessary to find appropriate values for these parameters are a common first selection criterion.

Computational complexity. For HO methods the complexity depends merely depends on the costs for evaluating per candidate solution, on the number of iterations, and, if applicable, on the population size and the costs of administering the population. Though the number of iterations (and population's size) will usually increase for larger problem spaces, the resulting increase in computational costs is usually substantially lower than it would be for traditional methods. Hence, the computational complexity of HO methods is comparatively low; even for NP complete problems, many HO algorithms have at most polynomial complexity. General statements, however, are difficult to make due to the differences in the HO techniques and, above all, the differences in the optimization problems' complexities.

Convergence speed. The CPU time (or, alternatively, the number of evaluated candidate solutions) until no further improvement is found, is often used as a measure to compare different algorithms. Speed might be a salient property of an algorithm in practical solutions – though not too meaningful when taken as a sole criterion as it does not necessarily differentiate between local and global optimum convergence and as long as a “reasonable” time limit is not exceeded.

Reliability. For some major heuristics, proofs exist that these methods will converge towards the global optimum – given sufficient computation time and an appropriate choice of parameters. In practice, one often has to accept a trade-off between low computational time (or high convergence speed) and the chance that the global optimum is missed. With the inherent danger of getting stuck in a local optimum, heuristics are therefore frequently judged by their ratio of reporting local optima or other inferior solutions.

To reduce the vagueness of these aspects, section 2.3 presents some of the major and commonly used heuristics that are typical representatives for this type of methods and that underline the differences in the methods with regard to the above

mentioned aspects: In *Threshold Accepting* (section 2.3.1), one solution is considered at a time and iteratively modified until it reaches an optimum; in *Evolution Based Methods* (section 2.3.2), a number of promising solutions are further evolved at the same time; in *Ant Systems* (section 2.3.3), collective experience is used; and *Memetic Algorithms* (section 2.3.4) are a typical example for successful hybrid algorithms where the advantages of several methods could be combined.¹⁹ There is neither a heuristic that outperforms all other heuristics whatever the optimization problem, nor can one provide a general implementation scheme regardless of the problem type.²⁰ The presentation in this introductory chapter is therefore reduced to the fundamental underlying ideas of the heuristics; more detailed descriptions will be offered when applied in the subsequent chapters.

2.3 Some Selected Methods

2.3.1 Simulated Annealing and Threshold Accepting

Kirkpatrick, Gelatt, and Vecchi (1983) present one of the simplest and most general HO techniques which turned out to be one of the most efficient ones, too: *Simulated Annealing* (SA). This algorithm mimics the crystallization process during cooling or annealing: When the material is hot, the particles have high kinetic energy and move more or less randomly regardless of their and the other particles' positions. The cooler the material gets, however, the more the particles are "torn" towards the direction that minimizes the energy balance. The SA algorithm does the same when searching for the optimal values for the decision parameters: It repeatedly suggests random modifications to the current solution, but progressively keeps only those that improve the current situation.

SA applies a probabilistic rule to decide whether the new solution replaces the current one or not. This rule considers the change in the objective function (mea-

¹⁹ For general presentations and comparisons of HO methods, see, e.g., Osman and Kelly (1996), Tallard, Gambardella, Gendreau, and Potvin (2001), Michalewicz and Fogel (1999), Aarts and Lenstra (2003) or Winker and Gilli (2004). Osman and Laporte (1996) offer an extensive bibliography of the theory and application of meta-heuristics, including 1 380 references. Ausiello and Protasi (1995) investigate local search heuristics with respect to NP optimization problems.

²⁰ See, e.g., Hertz and Widmer (2003).

```

generate random valid solution  $x$ ;
REPEAT
    generate new solution  $x'$  by randomly modifying
        the current solution  $x$ ;
    evaluate new solution  $x'$ ;
    IF acceptance criterion is met THEN;
        replace  $x$  with  $x'$ ;
    END;
    adjust acceptance criterion;
UNTIL halting criterion is met;

```

Listing 2.1: Basic structure for Simulated Annealing (SA) and Threshold Accepting (TA)

sureing the improvement/impairment) and an equivalent to “temperature” (reflecting the progress in the iterations). Dueck and Scheuer (1990) suggest a deterministic acceptance rule instead which makes the algorithm even simpler: Accept any random modification unless the resulting impairment exceeds a certain threshold; this threshold is lowered over the iterations. This algorithm is known as *Threshold Accepting (TA)*.

Listing 2.1 summarizes the pseudo-code for SA and TA where the values for the elements of a vector x are to be optimized; SA will be presented in more details when applied in chapter 3; different acceptance criteria will be compared in section 6.3.1. Both SA and TA usually start off with a random solution and generate new solutions by perfectly random search within the current solution’s neighborhood. In either method, the acceptance of impairments allows to overcome local optima. To avoid a Monte Carlo search path, however, improvements are more likely to be accepted than impairments at any stage, and with decreasing tolerance on impairments, the search strategy shifts towards a hill-climbing search.

SA and TA are both extremely flexible methods which are rather easy to implement. Both are general purpose approaches which cause relatively little computational complexity and for which convergence proofs exist.²¹ Single agent neighborhood search methods such as SA and TA have proofed successful when the solution space is not too rough, i.e., if the number of local optima is not too large.²²

²¹ See Aarts and van Laarhoven (1985) and Althöfer and Koschnik (1991), respectively.

²² For a concise presentation of TA, its properties and applications in economics as well as issues related to evaluating heuristically obtained results, see Winker (2001).

```

generate  $P$  random solutions  $x_1 \dots x_P$ ;
REPEAT
  FOR each parent individual  $i = 1 \dots P$ 
    generate offspring  $x'_i$  by randomly modifying
      the "parent"  $x_i$ ;
    evaluate new solution  $x'_i$ ;
  END;
  rank parents and offspring;
  select the best  $P$  of these solutions for new parent population;
UNTIL halting criterion met;

```

Listing 2.2: Basic structure for Evolutionary Strategies (ES)

2.3.2 Evolution Based and Genetic Methods

Inspired by their natural equivalent, the ideas of simulated evolution and artificial life have gained some tradition in machine learning and, eventually, in heuristic optimization.²³ One of the first algorithms actually addressing an optimization problem are *Evolutionary Strategies (ES)* by Rechenberg (1965). Here, a population of P initial solution vectors is generated. In each of the following iteration steps, each individual is treated as a parent that produces one offspring by adding a random modification to the parent's solution. From the now doubled population, only the best P agents are selected which will constitute the parent population in the next generation. Listing 2.2 summarizes the main steps of this original concept. Later versions offer modifications and improvements; in Rechenberg (1973), e.g., multiple parents generate a single offspring.

Evolution based methods gained significant recognition with the advent of *Genetic Algorithms (GA)*. Based on some of his earlier writings as well as related approaches in the literature, Holland (1975) attributes probabilities for reproduction to the individual "chromosomes," x_i , that reflect their relative fitness within the population. In the sense of the "survival of the fittest" principle, high fitness increases the chances of (multiple) reproduction, low fitness will ultimately lead to extinction. New offspring is generated by combining the chromosomes of two parent chromosomes; in the simplest case this cross-over can be done by "cutting" each parent's chromosomes into two pieces and creating two siblings by recombining each par-

²³ A survey on these topics can be found in Fogel (1998).

```
generate  $P$  random chromosomes;  
REPEAT  
  determine fitness of all chromosomes  $i = 1 \dots P$ ;  
  determine replication probabilities  $p_i$  based on relative fitness;  
  FOR number of reproductions;  
    randomly select two parents based on  $p_i$ ;  
    generate two children by cross-over operation on parents;  
  END;  
  insert offspring into the population;  
  remove  $P$  chromosomes based on inverse replication probability;  
  apply mutation to some/all individuals;  
UNTIL halting criterion met;
```

Listing 2.3: Basic structure for Genetic Algorithms (GA)

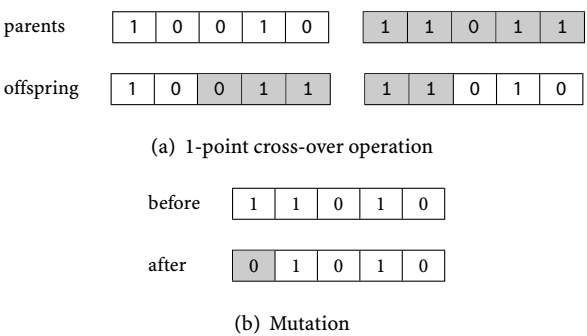


Fig. 2.2: Examples for evolutionary operators on binary strings

ent’s first part with the other parent’s second part (see Figure 2.2(a)). In addition mutation can take place, again by randomly modifying an existing (i.e., parent’s or newly generated offspring’s) solution (see Figure 2.2(b)).

Over the last decades, GA have become the prime method for evolutionary optimization – with a number of suggestions for alternative cross-over operations (not least because GA were originally designed for chromosomes coded as binary strings), mutation frequency, cloning (i.e., unchanged replication) of existing chromosomes, etc. Listing 2.3 therefore indicates just the main steps of a GA; the structure of actual implementations might differ. Fogel (2001) offers a concise overview of methods and literature in evolutionary computation.

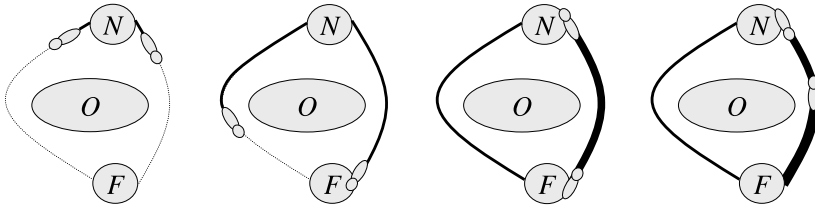


Fig. 2.3: Simple foraging example for a colony with two ants

Whereas SA and TA are single-agent methods where a solution is persistently modified (or “mutated”), evolutionary methods have to administer whole populations. At the same time, they all derive their new solutions by modifying existing current solutions. Evolutionary methods are more demanding to implement than are SA and TA. Also, they are more time-consuming because of their computational costs for administrating the population. At the same time, they are less likely to get stuck in local optima as the respective chromosomes are likely to be eventually be replaced with “fitter” alternatives.

2.3.3 Ant Systems and Ant Colony Optimization

Evolution has provided ants with a simple, yet enormously efficient method of finding shortest paths.²⁴ While traveling, ants lay pheromone trails which help themselves and their followers to orientate.

To illustrate the underlying principle, we assume a nest N and a food source F are separated by an obstacle O (Figure 2.3) and that there are two alternative routes leaving N both leading to F , yet different in length. Since the colony has no information which of the two routes to choose, the population (here consisting of two ants) is likely to split up and each ant selects a different trail. Since the route on the right is shorter, the ant on it reaches F while the other ant is still on its way. Supplied with food, the ant wants to return to the nest and finds a pheromone trail (namely its own) on one of the two possible ways back and will therefore select this alternative with a higher probability. If it actually chooses this route, it lays a second

²⁴ See Goss, Aron, Deneubourg, and Pasteels (1989).

pheromone trail while returning to the nest. Meanwhile the second ant has reached F and wants to bring the food to the nest. Again, F can be left on two routes: the left one (=long) has now one trail on it, the right one (=short) has already two trails. As the ant prefers routes with more pheromone in it, it is likely to return on the right path – which is the shorter one and will then have a third trail on it (versus one on the left path). The next time the ants leave the nest, they already consider the right route to be more attractive and are likely to select it over the left one. In real live, this self-reinforcing principle is enhanced by two further effects: shorter routes get more pheromone trails as ants can travel on them more often within the same time span than they could on longer routes; and old pheromone trails tend to evaporate making routes without new trails less attractive.

Based on this reinforcement mechanism, the tendency towards the shorter route will increase. At the same time, there remains a certain probability that routes with less scent will be chosen; this assures that new, yet unexplored alternatives can be considered. If these new alternatives turn out to be shorter (e.g., because to a closer food source), the ant principle will enforce it, and – on the long run – it will become the new most attractive route; if it is longer, the detour is unlikely to have a lasting impression on the colony's behavior.

Dorigo, Maniezzo, and Colorni (1991) transfer this metaphor to heuristic optimization called *Ant System* (AS) by having a population of artificial ants search in a graph where the knots correspond to locations and the arcs represent the amount of pheromone, i.e., attractiveness of choosing the path linking these locations. Being placed at an arbitrary location and having to decide where to move next, the artificial ant will choose (among the feasible) routes those with higher probability that are marked with more pheromone. The pheromone is usually administered in a pheromone matrix where two basic kinds of updates take place: on the one hand, new trails are added that are the stronger the more often they are chosen and the better the corresponding result; on the other hand, trails evaporate making rarely chosen paths even less attractive.

Since the original concept of AS parallels the Traveling Salesman Problem,²⁵ Listing 2.4 presents this algorithm for the task of finding the shortest route when a given number of cities have to be visited. Meanwhile, there exist several extensions and

²⁵ See section 2.1.2.1.

```
initialize trails and parameters;
REPEAT
  FOR all ants do;
    deposit ant at a random location;
    REPEAT
      select randomly next city according to pheromone trail;
    UNTIL route complete;
    determine tour length;
  END;
  let a fixed proportion of all pheromone trails evaporate;
  FOR all ants DO;
    add pheromone to chosen paths (more for shorter tours);
  END;
UNTIL halting criterion met;
```

Listing 2.4: Basic Structure for Ant System (AS)

variants most of which suggest improved trail update rules or selection procedures leading to higher reliability. Also there exist modifications to open this algorithm for optimization problems other than ordering. A survey can be found in Bonabeau, Dorigo, and Theraulaz (1999).

The concept of the pheromone matrix facilitates the gathering and sharing of collective knowledge and experience: While in the previously presented methods SA, TA and GA derive their new solutions from one (or two paternal) existing solution(s) and adding a random term to it, the contributions of many ants (from the current and past generations) support the generation of a new solution. As a result, ant based systems usually have high convergence speed and reliability – yet are also computationally more demanding as trail updates and the generation of new solutions is more complex. Another disadvantage is that ant based algorithms are less flexible in their application.

2.3.4 Memetic Algorithms

Single agent neighborhood search methods such as SA or TA, where one solution is modified step by step until convergence, are successful in particular when there is a limited number of local optima, when the agent can at least roughly figure out in which direction the global optimum can be expected and when this optimum

```
initialize population;  
REPEAT  
    perform individual neighborhood search;  
    compete;  
    perform individual neighborhood search;  
    cooperate;  
    adjust acceptance criterion;  
UNTIL halting criterion met;
```

Listing 2.5: Basic Structure for a Memetic Algorithm (MA)

is easily reachable given the step size and the distance between initial and optimal solution. If the algorithm appears to have problems of finding a solution or is likely to get stuck in local optima, one common remedy is to have a higher number of independent runs with different starting points, i.e., the optimization problem is solved repeatedly, and eventually the best of all found solutions is reported. Though the advantage of the independence between the runs is that mislead paths to local optima cannot be misleading in the current search, prior experience is lost and has to be gathered again. This increases inefficiency and run time. In population based methods such as GA, a whole population of agents produces several solutions at a time, which are regularly compared and the best of which are combined or re-used for new solutions. Population based methods therefore tend to be more likely to (eventually) overcome local optima. At the same time, they might have problems when already being close to the optimum where local neighborhood search would easily do the trick.

Moscato (1989) therefore suggests a method that combines the advantages of both concepts by having a population of agents that individually perform local search in a SA like fashion. In addition to the agents' independent neighborhood searches, they also compete and cooperate: competition is done in a tournament fashion where one agent challenges another and, if winning, imposes his solution onto the challenged agent; cooperation can be achieved by combining solutions with a cross-over operation as known, e.g., from GA. Unlike in other evolutionary methods, however, replacement in competition and cooperation uses the SA acceptance criterion instead of the replication probabilities and is therefore less time consuming. Listing 2.5 indicates the main steps of a simple version of MA.

This algorithm was inspired by a concept of Oxford zoologist Richard Dawkins who found that ideas and cultural units sometimes behave like “selfish” genes: they might be passed on from one person to another, they might be combined with other ideas, they mutate over time, and they have a tendency to self-replication. To resemble these properties, Dawkins introduced the term *meme* that reflects the French word for “self,” *même*, and is pronounced in a way that it rhymes with “gene.”²⁶

MA as presented here²⁷ is a typical *hybrid algorithm* that combines elements of other algorithms and enhances them with original ideas and approaches. Compared to the other algorithms presented so far, MA has lower computational complexity than GA (yet, of course, higher complexity than a pure SA implementation). Being more flexible in shifting between independent neighborhood search and joint population search, they are more flexible than the methods they are built on.

2.4 Heuristic Optimization at Work

2.4.1 Estimating the Parameters for GARCH Models

2.4.1.1 *The Estimation Problem*

In section 1.1.3, different ways for estimating the volatility were presented, including GARCH models where the volatility can change over time and is assumed to follow an autoregressive process. Applying these models, however, is not always trivial as the parameters have to be estimated by maximizing the likelihood function (1.14) (see page 22) which might have many local optima. In the lack of closed-form solutions, traditional numerical procedures are usually employed – which might produce quite different results.

²⁶ See Dawkins (1976, chapter 7). For a more in-depth presentation and discussion of the *meme* concept and its application in social sciences, see, e.g., Blackmore (1999).

²⁷ Meanwhile, the literature holds many different versions of Memetic Algorithms, some of which are population based whereas others aren’t, where the local search is not based on SA but on alternative methods such as Fred Glover’s *Tabu Search* (where a list of recently visited solutions is kept that must not be revisited again in order to avoid cycles), etc.; more details can be found in several contributions in Corne, Glover, and Dorigo (1999).

Given a time series r_t , Fiorentini, Calzolari, and Panattoni (1996) consider the simple GARCH(1,1) model (our notation)

$$r_t = \mu - e_t, \quad e_t | \Omega_{t-1} \sim N(0, \sigma_t^2) \quad (2.1a)$$

$$\sigma_t^2 = \alpha_0 + \alpha_1 \cdot e_{t-1}^2 + \beta_1 \cdot \sigma_{t-1}^2 \quad (2.1b)$$

with the objective of maximizing the conditional likelihood function, apart from the constant $-T/2 \cdot \ln(2 \cdot \pi)$,

$$\max_{\psi} \mathcal{L}(\psi) = \sum_{t=1}^T \left(-\frac{1}{2} \ln(\sigma_t^2) - \frac{1}{2} \frac{e_t^2}{\sigma_t^2} \right) \quad (2.1c)$$

where $\psi = [\mu, \alpha_0, \alpha_1, \beta_1]$ is the vector of decision variables and Ω_{t-1} is the information set available at time $t - 1$. They present a closed-form analytical expressions for the second derivatives of (2.1c) which can be used for initial values of ψ ; for the actual search, they test gradient methods.

Based on these results, Bollerslev and Ghysels (1996) provide parameter estimations for the daily German mark/British pound exchange rate.²⁸ Their estimates for the coefficients are then used for benchmarks by Brooks, Burke, and Persaud (2001) who estimate the parameters for the same data set with nine different specialized software packages. They find that only one of these packages is able to hit the benchmark coefficients and Hessian-based standard errors using the default settings. As this data set has become a benchmark problem for GARCH estimation²⁹ it can be used as a first example to illustrate how a heuristic optimization algorithm might be implemented and how the algorithm's performance can be optimized.

2.4.1.2 A Simple Heuristic Approach

To illustrate how to use heuristic optimization techniques and to test whether the obtained results are reliable, we approach the maximization problem (2.1) with one of the simpler of the introduced HO techniques, namely Simulated Annealing (SA) which has been introduced in section 2.3.1. Based on the pseudo-code in listing 2.1, the SA heuristic includes the following steps:

²⁸ The data, comprising 1974 observations, are available at www.amstat.org/publications/jbes/ftp.html → viewing existing publications → JBES APR-96 Issue → bollerslev.sec41.dat.

²⁹ See also McCullough and Renfro (1999).

- First, initial values for all decision variables (collected in the vector $\boldsymbol{\psi} = [\mu, \alpha_0, \alpha_1, \beta_1]$) are generated by random guesses. The only prerequisite for these guesses is that the guessed values are “valid” with respect to the constraints.
- The main part of SA consists of a series of iterations where the following steps will be repeated:
 - The algorithm produces a new candidate solution, $\boldsymbol{\psi}'$, by modifying the current solution, $\boldsymbol{\psi}$. To achieve this, one element j from $\boldsymbol{\psi}$ is selected arbitrarily. Then its current value is changed randomly. Formally, $\psi'_j = \psi_j + u \cdot \tilde{z}$ where $\tilde{z} \in [-1, +1]$ is an equally distributed random number. The other elements of $\boldsymbol{\psi}$ are left unchanged, i.e., $\psi'_k = \psi_k \forall k \neq j$.
 - Having generated a new candidate solution, $\boldsymbol{\psi}'$, the change in the objective function (here: the log-likelihood function) is calculated: $\Delta\mathcal{L} = \mathcal{L}(\boldsymbol{\psi}') - \mathcal{L}(\boldsymbol{\psi})$. According to the SA principle, a stochastic acceptance criterion for the new solution is applied that takes the change in the objective function, $\Delta\mathcal{L}$, into account as well as how progressed the algorithm is: In early iterations, even large impairments have a considerable chance of being accepted while in latter iterations, the criterion is increasingly less tolerant in accepting impairments. Usually, the acceptance criterion is the *Metropolis function* which will be presented in due course.
Based on this criterion’s decision, the current solution is either replaced with the new one (i.e., $\boldsymbol{\psi} \leftarrow \boldsymbol{\psi}'$) or not (i.e., $\boldsymbol{\psi}$ is left unchanged).
 - The acceptance criterion is to be modified over the course of iterations. SA is an analogue to the natural crystallization process while cooling. SA’s acceptance therefore involves a “temperature” T which is gradually lowered. The effect of this will be discussed in due course.

These steps of suggesting a new candidate solution and deciding whether to accept it for a new candidate solution or not (plus modifying the acceptance criterion), are repeated until some halting criterion is met. For the following implementation, the number of iterations is determined beforehand.

Listing 2.6 provides a pseudocode for the algorithm as presented. As the counter for the iterations starts with the value 2, the number of candidate solutions pro-

```

Initialize  $\psi$  with random values;

FOR i := 2 TO I do
     $\psi' := \psi$ ;
     $j := \text{RandomInteger} \in [1, \dots, \text{narg}(\psi)]$ ;
     $\tilde{z}_i := \text{RandomValue} \in [-1, +1]$ ;
     $\psi'_j := \psi_j + u_i \cdot \tilde{z}_i$ ;
     $\Delta\mathcal{L} := \mathcal{L}(\psi') - \mathcal{L}(\psi)$ ;
    IF  $\Delta\mathcal{L} > 0$  THEN
         $\psi := \psi'$ 
    ELSE
        with probability  $p = p(\Delta\mathcal{L}, T_i) = \exp\left(\frac{\Delta\mathcal{L}}{T_i}\right)$  DO
             $\psi := \psi'$ 
        END;
    END;

    % New overall best solution?
    IF  $\mathcal{L}(\psi) > \mathcal{L}(\psi^*)$  THEN
         $\psi^* := \psi$ ;
    END;

    Lower temperature:  $T_{i+1} := T_i \cdot \gamma_T$ ;
    If applicable:
        Adjust neighborhood range  $u_{i+1} := u_i \cdot \gamma_u$ ;
END;
Report best solution  $\psi^*$ ;

```

Listing 2.6: Pseudo-code for GARCH parameter estimation with Simulated Annealing

duced by the algorithm (including the initialization) is equal to I; note also that the iteration loop will be entered only if $I \geq 2$ and skipped otherwise.

A salient ingredient for an efficiently implement HO algorithm are proper values for the algorithm's parameters. For Simulated Annealing, the relevant parameters and aspects include the admitted run time (i.e., the number of iterations), a concept of "neighborhood" (i.e., the modification of ψ), and the acceptance criterion (i.e., a suitable cooling plan). What aspects should be considered in finding values for the respective parameters, will be discussed in the following section.

Unfortunately, there is no unique recipe for how to approach this task. Actually, it can be considered a demanding optimization problem in itself – which is partic-

ularly tricky: A certain parameter setting will not produce a unique, deterministic result but rather various results that are (more or less) randomly distributed; the task is therefore to find a combination where the distribution of the reported results is favorable. And as with many demanding problems, there are many possible parameter settings that appear to work equally well, yet it is hard to tell which one is actually the best among them.

Generally speaking, a good parameter setting is one where the algorithm finds reliable solutions within reasonable time. A common way for tuning the algorithm's parameters is to predefine several plausible parameter settings and to perform a series of independent experiments with each of these settings. The results can then be evaluated statistically, e.g., by finding the median or the quantiles of the reported solutions, and eventually select the parameter setting for which the considered statistics are the best; this approach will be used in the following section. Alternative approaches include *response surface analysis* and *regression analysis* where a functional relationship between the algorithm's parameters and the quality of the reported solutions is considered.

For complex algorithms where the number of parameters is high and their effects on the algorithm's quality are highly interdependent, a preselection of plausible parameter values is more difficult; in these circumstances, the parameter values can be found either by a Monte Carlo search – or by means of a search heuristic.

2.4.2 Tuning the Heuristic's Parameters

2.4.2.1 Neighborhood Range

General Considerations Simulated Annealing is a typical neighborhood search strategy as it produces new solutions that are close to the current solutions. It does so by slightly modifying one or several of the decision variables, in the above implementation by adding a random term to the current value: $\psi'_j := \psi_j + u \cdot \tilde{z}$. \tilde{z} is typically a normally or equally distributed random number; here it is chosen to be equally distributed within $[-1, +1]$. The parameter u defines what is considered a neighboring solution: the larger u , the larger the “area” surrounding ψ_j within which the new solution will be, and *vice versa*. Here, “small” and “large” steps have

to be seen relative to the variable that is to be changed; hence, the proper value for u will also depend on the magnitude of the different ψ_j 's.

Large values for u allow fast movements through the solution space – yet also increase the peril that the optimum is simply stepped over and therefore remains unidentified. Smaller step widths, on the other hand, increase the number of steps necessary to trespass a certain distance; if u is rather small, the number of iterations has to be high. Furthermore, u is salient for overcoming local optima: To escape a local optimum, a sequence of (interim) impairments of the objective function has to be accepted; the smaller u , the longer this sequence is. Smaller values for u demand a more tolerant acceptance criterion which might eventually lead to a perfectly random search strategy, not really different from a Monte Carlo search. With a strict acceptance criterion, small values for u will enforce an uphill search and therefore help to find the optimum close to the current position which might be advantageous in an advance stage of the search.

All this indicates that it might be favorable to have large values for u during the first iteration steps and small values during the last. Also, it might be reasonable to allow for different values for each decision variable if there are large differences in the plausible ranges for the values of the different decision variables.

Finding Proper Values Given the data set for which the GARCH model is to be estimated, the optimal value for $\psi_1 = \mu$ can be supposed to be in the range $[-1, +1]$. Also, it is reasonable to assume that the estimated variance should be non-negative and finite at any point of time. For the variables α_0 , α_1 , and β_1 in equation (2.1b) (represented in the algorithm by ψ_2 , ψ_3 , and ψ_4), it is plausible to assume that their values are non-negative, but do not exceed 1; hence, their optimal values are expected in the range $[0, +1]$.

As only one of these decision variables is modified per iteration and the number of iterations might be rather small, we will test three alternatives where u_1 will have an initial value of 0.05, 0.025, or 0.01; the actual modification, $u_1 \cdot \tilde{z}$, will then be equally distributed in the range $[-u_1, +u_1]$.

As argued, it might be reasonable to narrow the neighborhood in the course of the search process. We will therefore test four different versions where u is kept either constant; the value of u in the terminal iteration I is u 's initial value divided by

10, 100, or 1 000. The value for u shall be lowered gradually in the course of iterations according to $u_{i+1} = u_i \cdot \gamma_u$. This implies that the parameter γ_u is to be determined according to $\gamma_u = \sqrt[I]{u_i/u_1}$. With the chosen values for u_1 and u_I , γ_u can take the values 1, $\sqrt[I]{0.1}$, $\sqrt[I]{0.01}$, and $\sqrt[I]{0.001}$.

2.4.2.2 The Number of Iterations

General Considerations For some heuristic optimization algorithms, there exist proofs that the global optimum will be identified – eventually. In practical applications, concessions have to be made in order to find solutions within reasonable time. This is primarily done by restrictions on the run time or on the number of iterations. For the latter alternative, common solutions include convergence criteria and upper limits on the number of iterations. Convergence criteria assume that the algorithm has found a solution which is either the global solution – or some local optimum which is unlikely to be escaped and computation time therefore ought to be used for new runs.

As indicated above, selecting the number of iterations is related to finding the parameter for the step size, u , and *vice versa*: The neighborhood range should be large enough that the optimum can actually be reached within the chosen number of iterations and from any arbitrary starting point. Also, for some problems it might be reasonable to have more runs with independent initial guesses for the decision variables, whereas for others it might be advantageous to have fewer runs, yet with more iterations per run.

Finding Proper Values The algorithm will report a solution which it has actually guessed and reached by chance at one stage of the search process. The algorithm, however, does not have a mechanism that guides the search, e.g., by using gradients, estimating the step width with some interpolation procedure or based on past experience. At the same time, we demand a high precision for the parameters. The algorithm is therefore conceded 50 000 guesses before it reports a solution; these guesses can be spent on few independent runs with many iterations or the other way round. We distinguish four versions where all guesses are used on one search run (i.e., the number of iterations is set to $I = 50\,000$), 5 and 50 independent runs with $I = 10\,000$ and 1 000 iterations, respectively, and version where no iterative search

is performed but all the guesses are used on perfectly random values; this last version corresponds to a Monte Carlo search and can serve as a benchmark on whether the iterative search by SA has a favorable effect on the search process or whether a perfectly random search strategy might be enough.

2.4.2.3 Acceptance criterion

General Considerations In Simulated Annealing the acceptance probability, p , is often determined via the *Metropolis function*, $p = \min \{ \exp(\Delta\mathcal{L}/T_i), 100\% \}$.³⁰ For maximization problems, a positive sign for the change in the objective function, $\Delta\mathcal{L} > 0$, indicates an improvement, $\Delta\mathcal{L} < 0$ indicates an impairment. T_i is the analogue for the temperature in iteration i and serves as an adjustment parameter to make the criterion more or less tolerant to impairments: High temperatures push the argument of the $\exp(\cdot)$ expression towards zero and hence the acceptance probability towards 100%, and changes with $\Delta\mathcal{L} \ll 0$ might still be accepted; low temperatures have the adverse effect and make even small impairments unlikely. Improvements, however, are accepted whatever the temperature: as $\exp(\cdot) > 1$ (and therefore exceeds the min-function's limit of 100%) when the argument is positive, the Metropolis function will return an acceptance probability of 100% whenever $\Delta\mathcal{L} > 0$.

Finding good values for the temperature is strongly dependent on what are “typical” impairments. This can be achieved by performing a series of modifications, evaluating the resulting changes in the objective function, $\Delta\mathcal{L}$, and determining the quantiles of the distribution of negative $\Delta\mathcal{L}$'s.

Solving the Metropolis function for the temperature yields $T_i = \Delta\mathcal{L}/\ln(p)$. Hence, during the early iterations, T_i should have values that allow most of the impairments to be accepted with reasonable probability; T_i should therefore be chosen such that a relatively large impairment is accepted with high probability. In the last iterations, only few of the impairments ought to be accepted; here, T_i should have a value such that even a relatively small impairment is accepted with low probability. Once the temperatures for the first and last iterations, T_1 and T_I , have been

³⁰ See also the discussion in section 6.3.1.

u	95%	90%	75%	50%	25%	10%	5%
0.05	-17.70569	<i>-13.42215</i>	-8.13720	-3.59098	-0.78051	-0.21308	-0.08924
0.025	-17.60417	<i>-13.09995</i>	-7.92351	-3.46938	-0.52063	-0.13813	-0.05751
0.01	-17.60372	<i>-13.15639</i>	-8.06564	-3.58483	-0.35556	-0.06503	-0.02704
0.005	-16.94350	-13.06744	-8.07430	-3.41242	-0.19775	-0.02877	-0.01147
0.0025	-17.65643	-13.24645	-8.00821	-3.44650	-0.11675	-0.01526	-0.00588
0.001	-17.59644	-13.15123	-8.05048	-3.45760	-0.12361	-0.00617	-0.00245
0.0005	-17.77629	-13.09759	-7.96761	-3.50134	-0.22632	-0.00315	-0.00127
0.00025	-17.63221	-13.09240	-7.93378	-3.41761	-0.17682	-0.00161	-0.00066
0.0001	-17.65130	-13.12136	-7.97900	-3.51114	-0.10407	-0.00063	-0.00025
0.00005	-17.05599	-12.85940	-7.7959	-3.43883	-0.1256	-0.00029	-0.00011
0.000025	-17.49037	-12.95163	-7.91984	-3.45329	-0.11150	-0.00016	-0.00006
0.00001	-18.15359	-13.32715	-8.06117	-3.57329	-0.23852	-0.00007	-0.00003

Tab. 2.1: Quantiles for modifications with negative $\Delta\mathcal{L}$ for different values of u , based on 10 000 replications each (*italics and boldface as explained in the text*)

found, the *cooling parameter* $\gamma_T = \sqrt[3]{T_1/T_1}$ can be determined where \mathbf{I} is the chosen number of iterations per run. In each iteration, the temperature is then lowered according to $T_{i+1} = T_i \cdot \gamma_T$.

Finding Proper Values In order to find suitable values for the temperature, the distribution of the potential impairments due to one local neighborhood search step has to be found. This can be done by a Monte Carlo approach where first a number of candidate solutions (that might occur in the search process) are randomly generated and for which the effect of a modification is evaluated. As this distribution depends on what is considered a local neighborhood, Table 2.1 summarizes the quantiles for impairments for the different values of u in the first and the last iteration.

As stated above, the initial values for u were selected from the alternatives [0.05, 0.025, 0.01]. For these three alternatives, the 90% quantiles of the impairments were approximately -13 (see figures in *italics* in Table 2.1). Hence, if in the beginning, 90% of all impairments should be accepted with a probability of at least $p = 0.5$, then temperature should be set to $T_1 = -13/\ln(0.5) \approx 20$.

The 10% quantiles of the impairments depend strongly on the value of u ; they can be approximated by $-6 \cdot u$ (see figures in **boldface** in Table 2.1). Hence, if only the

10% of impairments that are smaller than this critical value shall be accepted with a probability of more than $p = 0.1$ in the last iteration, I , the temperature should be set to $T_I = -6 \cdot u_I / \ln(0.1) \approx 2.6 \cdot u_I$. As this shall be the case in the last iteration, the cooling factor is set to $\gamma_T = \sqrt[I]{u_I \cdot 2.6/20}$ where I is the number of iterations.³¹

2.4.3 Results

Based on the above considerations, there are three candidate values for the initial value of u ($u_1 = 0.05, 0.025$ or 0.01) and four alternatives for the terminal value of u ($u_{I_T} = 1/1, 1/10, 1/100$, and $1/1000$; the values for γ_u follow directly according to $\gamma_u = \sqrt[I]{u_1/u_{I_T}}$). In addition, we test four different alternatives to use the conceded 50 000 guesses (ranging from a single run with $I = 50\,000$ iterations to 50 000 independent runs with $I = 1$, i.e., without subsequent iterations³²), there are 48 different parameter settings to be tested. With each of these combinations, the algorithm was applied to the optimization problem several hundred times, and from each of these experiments, the best of the 50 000 candidate solutions was reported and the distributions of the reported solutions are evaluated. Implemented in Delphi (version 7), the CPU time per experiment (i.e., per 50 000 candidate solutions) was approximately 15 seconds on a Centrino Pentium M 1.4 GHz. Table 2.2 summarizes the medians and 10% quantiles for the deviations between reported solutions and the optimum.

When an adequate parameter setting has been chosen, the algorithm is able to find good solutions with high probability: when the number of iterations is sufficiently high (e.g., $I = 50\,000$) and the parameters for the neighborhood search are well chosen (e.g., $u_1 = 0.025$ and $u_{I_T} = 0.001$), then half of the reported solutions will have a \mathcal{L} which is at most 0.00001 below the optimum. The best 10% of the solutions generated with this parameter setting will deviate by just 0.000001 or less.

³¹ A more sophisticated consideration could take into account that during the last iteration, the algorithm has converged towards the optimum and that a random step in this region might have a different effect on $\Delta\mathcal{L}$ as the same modification would cause in a region far from the optimum. Also, the values for p were chosen based on experience from implementations for similar problems; more advanced considerations could be performed. However, for our purpose (and for many practical applications), the selection process as presented seems to generate good enough results.

³² Note that in listing 2.6, the loop of iterations is not entered when $I = 1$: To assure that the initial values, too, count towards the total number of guesses, the loop is performed only $I - 1$ times.

		runs \times number of guesses per run (I)			
u_1	u_2/u_1	$1 \times 50\,000$	$5 \times 10\,000$	$50 \times 1\,000$	$50\,000 \times 1, MC$
median	1	-0.011240	-0.016400	-0.347450	-17.895114
	0.1	-0.000795	-0.000985	-0.233348	
	0.01	-0.000055	-0.000062	-0.425277	
	0.001	-0.000007	-0.000078	-2.224412	
	1	-0.004913	-0.005810	-0.489114	
	0.1	-0.000328	-0.000359	-0.650558	
	0.01	-0.000024	-0.000103	-5.468503	
	0.001	-0.000008	-1.144997	-23.693378	
	1	-0.001681	-0.001865	-1.288883	
	0.1	-0.000114	-0.000572	-16.841008	
	0.01	-0.000012	-6.891969	-65.365018	
	0.001	-7.289898	-32.970728	-99.662400	
10% quantile	1	-0.003348	-0.005210	-0.067568	-8.625649
	0.1	-0.000309	-0.000363	-0.010878	
	0.01	-0.000020	-0.000025	-0.011645	
	0.001	-0.000002	-0.000008	-0.054592	
	1	-0.001790	-0.002157	-0.047682	
	0.1	-0.000123	-0.000127	-0.020024	
	0.01	-0.000008	-0.000024	-0.090048	
	0.001	-0.000001	-0.008738	-2.413633	
	1	-0.000614	-0.000749	-0.067869	
	0.1	-0.000041	-0.000112	-0.611584	
	0.01	-0.000004	-0.022991	-15.653601	
	0.001	-0.193161	-2.774787	-37.478838	

Tab. 2.2: 10% quantiles and medians of differences between reported solutions and optimum solution for different parameter settings from several hundred independent experiments (typically 400; MC: 7 800) with 50 000 candidate solutions each

If, on the other hand, the heuristic search part is abandoned and all of the allowed 50 000 guesses are used on generating independent random (initial) values for the vector of decision variables, then the algorithm performs a sheer Monte Carlo (MC) search where there is no neighborhood search (and, hence, the values for u_i are irrelevant) and where, again, only the best of the 50 000 guesses per experiment is reported. The results are by magnitude worse than for SA with a suitable set of parameters (see last column, labeled MC). This also supports that the use of the search heuristic leads to significantly better results – provided an appropriate parameter setting has been selected.

A closer look at the results also underlines the importance of suitable parameters and that inappropriate parameters might turn the algorithm's advantages into their exact opposite. When there are only few iterations and the neighborhood is chosen too small (i.e., small initial value for u which is further lowered rapidly), then the step size is too small to get anywhere near the optimum within the conceded number of search steps. As a consequence, the algorithm virtually freezes at (or near) the initial solution.

However, it also becomes apparent that for most of the tested parameter combinations, the algorithm performs well and that preliminary considerations might help to quickly tune a heuristic optimization algorithm such that it produces good results with high reliability. Traditional methods are highly dependent on the initial values which might lead the subsequent deterministic search to the ever same local optimum. According to Brooks, Burke, and Persand (2001) the lack of sophisticated initializations is one of the reasons why the tested software packages found solutions for the considered problem that sometimes differ considerably from the benchmark. Table 2.3 reproduces their parameter estimates from different software packages³³ together with the benchmark values and the optimum as found by the SA algorithm

³³ Brooks, Burke, and Persand (2001) report only three significant figures for the estimates from the different software packages, also the packages might use alternative initializations for σ_0^2 . (Our implementation uses the popular approach $\sigma_0^2 = e_0^2 = \frac{1}{T} \sum_{t=1}^T e_t^2$ with e_t coming from equation (2.1a).) Reliable calculations of the respective values for \mathcal{L} that would allow for statistically sound tests on the estimation errors are not possible.

Method	$\psi_0 = \mu$	$\psi_1 = \alpha_0$	$\psi_2 = \alpha_1$	$\psi_3 = \beta_1$	
Benchmark	-0.00619041	0.0107613	0.153134	0.805974	
Heuristic optimization	-0.00619034	0.0107614	0.153134	0.805973	
Software packages	E-Views	-0.00540	0.0096	0.143	0.821
	Gauss-Fanpac	-0.00600	0.0110	0.153	0.806
	Limdep	-0.00619	0.0108	0.153	0.806
	Matlab	-0.00619	0.0108	0.153	0.806
	Microfit	-0.00621	0.0108	0.153	0.806
	SAS	-0.00619	0.0108	0.153	0.806
	Shazam	-0.00613	0.0107	0.154	0.806
	Rats	-0.00625	0.0108	0.153	0.806
	TSP	-0.00619	0.0108	0.153	0.806

Tab. 2.3: Results for the GARCH estimation based on the benchmark provided in Bollerslev and Ghysels (1996), the results from the software packages (with default settings) as reported in Brooks, Burke, and Persaud (2001)

– which, by concept, uses perfectly random initial values.³⁴ Unlike with traditional deterministic optimization techniques, this reliability can arbitrarily be increased by increasing the runtime (which is the basic conclusion from convergence proofs for HO algorithms).

2.5 Conclusion

In this chapter, some basic concepts of optimization in general and heuristic optimization methods in particular were introduced. The heuristics presented in this chapter differ significantly in various aspects: the varieties range from repeatedly modifying one candidate solution per iteration to whole populations of search agents each of them representing one candidate solution; from neighborhood search strategies to global search methods, etc. As diverse these methods are, as diverse are

³⁴ In practice, HO techniques do not always benefit when the initial values come from some “sophisticated guess” or another optimization as this often means that the optimizer first and prime task is to overcome a local optimum. On the contrary, heuristically determined solutions might sometimes be used as initial values for traditional methods. Likewise, it might be reasonable to have the fine-tuning of the heuristic’s last iterations done by a strict up-hill search.

also their advantages and disadvantages: Simulated Annealing and Threshold Accepting are relatively easy to implement and are good general purpose methods, yet they tend to have problems when the search space is excessively large and has many local optima. Other methods such as Genetic Algorithms or Memetic Algorithms, on the other hand, are more complex and their implementation demands some experience with heuristic optimization, yet they can deal with more complicated and highly demanding optimization problems. Hence, there is not one best heuristic that would be superior to all other methods. It is rather a “different courses, different horses” situation where criteria such as the type of optimization problem, restrictions on computational time, experience with implementing different HO algorithms, the programming environment, the availability of toolboxes, and so on that influence the decision which heuristic to choose – or eventually lead to new or hybrid methods.

The following chapters of this contribution make use of heuristic optimization techniques for approaching problems, merely from the area portfolio management, that cannot be answered with traditional models. The diversity of the problems leads to the application of different methods as well as the introduction of a new hybrid approach. Though the main focus of these applications shall be on the financial implications that can be drawn from the results, there will also be some comparisons of these methods together with suggestions for enhancements.



<http://www.springer.com/978-0-387-25852-2>

Portfolio Management with Heuristic Optimization

Maringer, D.G.

2005, XIV, 223 p., Hardcover

ISBN: 978-0-387-25852-2