Leipzig University
Faculty of Mathematics and Computer Science

Hyperparameter Optimization Methods for the H-SPPBO Metaheuristic

Applied to the Dynamic Traveling Salesperson Problem

Master's Thesis

Author:
Daniel Werner
3742529
Computer Science

Supervisor:
Prof. Dr. Martin Middendorf

Swarm Intelligence and Complex Systems Group
March 14, 2023

Contents

1 Introduction

1.1 Motivation

The applications of metaheuristics in a world constantly striving for optimization are vast. From finding the shortest path for the vehicle transporting an online purchase [75], to routing the traffic from "Internet of Things" (IoT) devices [64], these algorithmic problem solvers are unknowingly omnipresent. As systems become more complex, a demand for metaheuristics has emerged, as they are able to find solutions to underdetermined functions [34], computationally intensive systems, or NP-hard problems. Even when looking at more mainstream technology topics, especially in data mining or machine learning (ML), metaheuristics play an important role in so-called *hyperparameter optimization* [79]. Metaheuristic algorithms like Particle Swarm Optimization (PSO) and Genetic Algorithms (GAs) are used to find the ideal combination of parameters needed for a ML model to perform its best.

According to the "No free lunch theorem" [77] there cannot exist an optimization algorithm which is perfectly suited for all kinds of problems. Therefore, tackling these pressing research topics requires not one *perfect*, but multiple different metaheuristics. That is especially true, considering the emergence of new problems and challenges, requiring or even imposing a metaheuristic to solve this very optimization problem.

As a result, this increasingly growing scientific field is not only becoming more convoluted [69], the algorithmic contexts, not necessarily the algorithms themselves, are also becoming more complex. A streamlined metaheuristic framework, like the Simple Probabilistic Population-Based Optimization (SPPBO) proposes [49], can have multiple parameters to choose from and then, ideally, tune to the problem class and instance it shall solve. While problems like the Traveling Salesperson Problem (TSP) or the Quadratic Assignment Problem (QAP) have the benefit of being an abstracted version of real-world applicable problems (coming with their own benchmarking packages to boot [9, 61]),

there are plenty of other problems with a multitude of factors to consider, when configuring the parameters for your metaheuristic algorithm. Besides, the real-world is rarely static, therefore, implying the need for solving dynamically changing problems as well.

1.2 Problem and Scope

Knowing the context of modern metaheuristic research, this thesis focuses on the problem arising from feature- and parameter-rich metaheuristic frameworks, exemplified by the aforementioned SPPBO framework [49]. It combines and generalizes aspects from popular swarm intelligence algorithms, namely the Population-Based Ant Colony Optimization (PACO) and the Simplified Swarm Optimization (SSO). And while the SPPBO framework reduces algorithmic complexity, draws similarities to existing metaheuristics and therefore, holds the possibility of solving a greater problem space, it also needs to be configured correctly to perform its best. Solving this task manually, changing the parameters each iteration and looking at the results, is not only inefficient and tedious, but also error-prone, bearing the risk of getting stuck in a local optimum of an multi-dimensional parameter space.

This is also the case for Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO) [42], an algorithm derived using the SPPBO framework and incorporating aspects from Hierarchical Particle Swarm Pptimization (H-PSO) [35]. The hierarchical tree structure organizes a population of Solution Creating Entities (SCEs), which, as the name suggests, each create a solution to the presented problem per iteration - similar to the ants in PACO. The tree root represents the SCE with the best solution found so far, branching out into its sibling SCEs and their less good solutions, and so forth. This structure is subject to change with every new iteration of solutions, establishing a clear hierarchy of influence between the SCEs. By observing specific swap-patterns of this tree and its SCEs, the H-SPPBO algorithm is able to detect dynamic changes within the problem instance it solves and reacts accordingly to improve the solution, analyzed similarly in [36].

This opens up the scope of this thesis to dynamically changing problems, like the Dynamic Traveling Salesperson Problem (DTSP) [58]. Whereas for the 'normal' symmetrical TSP the solution for a given instance of a list or grid of cities would be the shortest path that visits each node ("city") exactly once, resulting in a Hamiltonian cycle, in practical applications an exact problem description is often not given in advance. Thus, the DTSP

is needed to model behavior corresponding with, for example, destinations changing during the routing of vehicles or new cities needing to be visited while the process is already underway.

1.3 Approach

To summarize, we want to solve the TSP, and its dynamic version, using the H-SPPBO algorithm. Furthermore, we want to detect dynamic changes that occur within the problem instances during runtime and react accordingly, to create a newly adapted solution as quickly as possible. And all this with the best available set of parameters. For this last crucial step we take a page from the field of machine learning, where optimizing a model's hyperparameters was already a research topic in the 1990s [21]. Since then, Hyperparameter Optimization (HPO) has evolved into an important staple in that research community, being implement into almost every modern ML training software and having multiple open-source standalone packages, written in most common programming languages, the most popular being *Python*. It is precisely this knowledge of optimizing parameters for functions that are often expensive to execute - be it a nondeterministic algorithm or a complex artificial neural network - that we want to leverage for our problem.

In this context, the two arising main research questions are:

1. What is the ideal HPO method for the H-SPPBO algorithm?
2. Which sets of parameters yield the best results for a given DTSP instance?

This thesis provides a complete software package written in *Python*, containing all the necessary parts needed to answer the research question outlined above. Every aspect of this package is modular (allowing for easy replacement), highly configurable (being able to adapt for other algorithms than H-SPPBO) and well-documented (increasing comprehensibility and replicability of the findings described here).

1.4 Outline

Chapter 2 continues with references to related work and solutions to similar problems, especially concerning dynamic problem solving and parameter tuning for metaheuristics. Chapter 3 explains the theoretical foundations and knowledge needed to fully understand the methods described. Complementing this, Chapter 4 provides insight into the software implementation, details about the libraries used and makes the algorithms and control flow more understandable in a programmatically oriented way. Progressing to the research part of the thesis, chapter 5 lays out the design of experiments carried out and explains in detail the reasoning behind the selection of problem instances and parameter spaces. Chapter 6 presents the results and discusses them with regard to the two main research questions. Lastly, a summary of the work and an outlook on further questions and methods to proceed are given in chapter 7.

2 Related Work

2.1 Metaheuristics on Dynamic Problems

One of the first publications to propose the DTSP as a potential and relevant problem was the work of Psaraftis [57] in 1988, mentioning "interchange methods", like the 2-opt [10], 3-opt [47] or Lin-Kernighan [48] methods, for solving slow dynamic changes occurring in the TSP. Using metaheuristics for dynamic problems began its early development starting in the 2000s. The theoretical concept of the DTSP was discussed by Huang et al. [33]. Following this, the work of Angus and Hendtlass [1] indicated, that adapting the Ant Colony Optimization (ACO) to a dynamic change of the TSP is faster than restarting the algorithm, while Guntsch and Middendorf [29] already proposed a general method for the ACO using modified pheromone diversification strategies to counteract the random insertion or deletion of cities in a TSP instance. Similar methods were suggested by Eyckelhof and Snoek [20]. Unlike changing the node topology of the TSP, Silva and Runkler [65] applied dynamic constrains to the nodes and left the evaluation to the ants. More recent reviews regarding the usage of ACO on different versions of the DTSP were done by Mavrovouniotis and Yang [52], [51].

Examining other metaheuristic categories, Li et al. [45] solve a version of the DTSP by using an evolutionary algorithm that applies genetic-like operations of inversion and recombination. Another proposal in the field of GA is the work of Simoes and Costa [66], using an algorithm based on CHC ("Cross-generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation" [19]) to solve a dynamic version of the TSP with changes to edge weights and insertions, deletions and swapping of city nodes.

PSO was also successfully used to react to dynamic changes in works by Janson and Middendorf [36], [37]. Several variants of the PSO - H-PSO and a partitioned version (PH-PSO) - were used to not only solve, but additionally detect changes within the dynamic problem instances presented.

2.2 Parameter Optimization for Metaheuristics

The choice of parameters was always an important consideration in research of metaheuristics. Eiben et al. [18] give an in-depth review and analysis over the different options in the field of parameter settings. Although focusing on GAs, they create a useful taxonomy for parameter optimization. They propose a distinction between *parameter tuning*, where the parameters are fixed before runtime, and *parameter control* strategies, where the values vary during algorithm runtime. *parameter control* are then distinguished by their type of value modification. Eiben et al. come to the conclusion, that a *parameter control* strategy usually results in better solutions compared to *parameter tuning*. Talbi [72] did a similar classification, including further distinction between (offline) parameter tuning methods. Lastly, In their review of parameter optimization, Wong et al. [78] conclude that parameter tuning plays an important role in the exploratory and exploitative behavior of ACO.

The most popular reference in manual parameter tuning for ACO is the original work by Dorigo et al. [14, 15]. The resulting values of several experiments with varying parameter combinations performed on one TSP instance serve as a baseline for many following studies. An updated version of 'good parameters' including variations of the original ACO was published by Dorigo and Stützle [16]. These parameter combinations were then challenged by Gaertner and Clark [24], who classified the TSP instances by certain properties, and then tried out wide ranges of parameters for these new TSP categories. They found optimal parameter values that, in some cases, differ greatly from the originally proposed values. More recent work in the field of parameter tuning is done by Tuani et al. [74], where, for a heterogeneous ACO, each ant is initialized with different random distributions for the α and β values. The algorithm was tested on multiple TSP instances and compared against different ACO variant and their heterogeneous counterparts.

A thorough review, classification and research on online parameter control was done by the aforementioned [71]. Building up on the taxonomy proposed by Eiben et al., they modify and apply these categories to other approaches in the field of ACO parameter control. Furthermore, computational time and any-time behavior of algorithms are also factored in for their experiments done with deterministic parameter control strategies. Further research on this type of parameter optimization was done by Neyoy et al. [54], where fuzzy logic statements are used to improve the solution diversification behavior. A self-adaptive control scheme was proposed by Hao et al. [30], constructing a combinatorial problem of the parameter search and applying PSO to optimize them in each iteration.

Benchmarks using the TSP promise good results. Similar studies were done by Li and Zhu [46], with a version of the ACO having its parameters controlled by a bacterial foraging algorithm and compared against parameter control through GA and PSO. Other interesting developments include the work of Randall [60], who construct an almost parameter free version of the ACO.

One of the first analogies drawn between optimizing metaheuristics and ML can be found in *Tuning Metaheuristics: A Machine Learning Perspective* by Birattari and Kacprzyk [3]. They analyze the similarities to problems that supervised learning also faces and propose guidelines for sampling parameter sets. Finally, they apply their findings using a version of the Hoeffding race algorithm [50] (originally used to select good models, of which HPO is a subset) to, among other examples, tune the parameters of a MAX–MIN Ant System to solving the TSP. A different approach in the context of applying ML concepts to parameter optimization was proposed by Dobslaw [13], explaining the possibility to train an artificial neural network (ANN) on the relation between the characteristics of a problem instance and a parameter set that resulted in good solutions. The necessary training data is to be acquired using the Design of Experiment (DoE) framework. On that note, several other authors also proposed some form of DoE variant, in this case the Taguchi method, to optimize or select ML models and/or their hyperparameters, see [40, 55, 73].

Lastly, a similar approach to this thesis, albeit more narrow in scope, was researched by Yin and Wijk [82]. They used two hyperparameter optimization methods, random search and Bayesian Optimization (BO), for tuning the parameters of a classic ACO algorithm on several instances of the asymmetric TSP. Their results promise great potential for tuning parameters this way, without requiring a priori knowledge of the problem.

3 Theoretical Background

3.1 Traveling Salesman Problem

3.1.1 A Brief History

Having its basis in the mathematical theory around the Hamiltonian cycle in the 19th century, one of the first publications mentioning the term **Traveling Salesperson Problem (TSP)** was by Robinson in 1949 as part of an United States research company, a think tank called "RAND Corporation", which offered their services to the U.S. armed forces [63]. Alongside Robinson's proposed solution, the scientific community at that time was very interested in the TSP, applying all sorts of mathematical graph operations and often using branch cutting algorithms to solve it [44]. The beauty of this problem lied in its simple, brief description, which made it easy to understand, but also its non-trivial and engaging solutions.

With advances in computer science also came more computational applicable algorithms like the Kernighan–Lin heuristic [48]. At the same time, the TSP was found to be NP-complete and therefore, NP-hard [56] - a problem category that still remains a very interesting research topic in computer science. Through the work by Dorigo et al. [14] and the "ant system", metaheuristics began to be a valid choice for solving the TSP in the early 1990s. Alongside, with emergence of the *TSPLIB* benchmarking suite [61] began a vast adoption of said test instances to compare and rank new algorithms and (meta-)heuristics. Applications for the solutions are numerous, from the apparent routing of travel routes to frequency assignment [59].

3.1.2 Theory

The problem description of the symmetric TSP can best be modeled by an undirected weighted graph $G = (V, E)$, with a set of n vertices $V = \{v_1, ...v_n\}$ and a set of edges $E = \{(v_i, v_j)|v_i, v_j \in V, i \neq j\}$. Staying in the traveling salesperson analogy, the vertices are often being referred to as "cities", while the edges between vertices represent the distance. This can be expressed by a distance function $d : V^2 \rightarrow \mathbb{R}_0^+$, which, in the case of Euclidean TSP, is just the Eucledian distance between two points in a two dimensional space. Each city can only be visited once and since its symmetric, it does not matter which direction the abstract salesperson travels. The problem is to find the shortest (meaning smallest total distance) tour that visits each city exactly once, starting and ending at the same city. The solution resembles a permutation of V, written as $\mathbf{s} = (s_1, ..., s_n) \in V^n$, that minimizes the sum over its distances resulting in a solution quality function over s

$$f(s) = \sum_{i=1}^{n} d(s_i, s_{i+1}) \tag{3.1}$$

with the city node s_{i+1} coming after s_i and $s_{n+1} = s_1$ to complete the full tour. For a more precise problem description, the following applies:

> Given a weighted undirected graph, find the Hamilton cycle that minimizes the weight of all edges traversed.

In most cases, the graph is also complete, having each vertex connected with each other. Especially for computational tasks, its often easier to view the TSP description as a distance matrix defined as follows:

$$\mathbf{D} = \{d(v_i, v_j)\}_{nxn} \tag{3.2}$$

An example of a simple TSP instance can be seen in Fig. 3.1, its distance matrix would then be

$$\mathbf{D} = \begin{bmatrix} 0 & 7 & 7 & 10 \\ 7 & 0 & 10 & 7 \\ 7 & 10 & 0 & 7 \\ 10 & 7 & 7 & 0 \end{bmatrix}.$$

There are multiple modifications to this standard problem. The asymmetric TSP (ATSP) has the added complication of the weights between the vertices being different depending on the direction the edge is being traversed, so $d(v_i, v_j) \neq d(v_j, v_i)$, which increases the

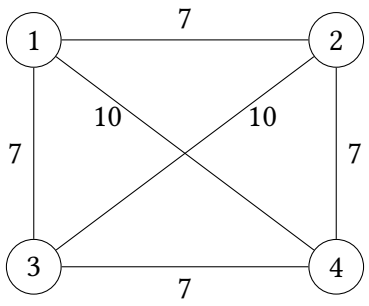


Figure 3.1: Example of a symmetric TSP instance with $n = 4$ cities

complexity and optimization potential for these instances [38]. The Dynamic Traveling Salesperson Problem (DTSP) incorporates certain dynamic aspects like changing edge weights or the insertion, deletion or swapping of city nodes into the problem sphere. It is different in the sense that it has neither a specific problem description nor generally valid solutions, since that all depends heavily on the implementation. The basis often is a traditional symmetric TSP instance with some form of the above mentioned dynamic changes applied deterministically or randomly over time t. Therefore, the distance matrix in Eq. (3.2) becomes time dependent $\mathbf{D}(t)$. Searching for solutions to an ever changing problem increases the importance of a good anytime behavior of the algorithm or metaheuristic, with the ultimate goal of finding the best possible solution at any given moment, so Eq. (3.1) becomes time dependent $f(s, t)$ as well.

3.2 Metaheuristics

Solving a hard optimization problem with an exact (deterministic) method may yield the very best solution to the problem, but often at the cost of high computational intensity. And, although, often giving mathematical proof that an optimal solution is achieved in reasonable time, that duration can be considerably large. A metaheuristic does not deal in exact solutions, but rather tries to find the *best* solution in a given, often small enough, time frame. Furthermore, a heuristic algorithm needs specific knowledge about the problem it solves, where on the other hand, metaheuristics are generally adaptable to a larger space of optimization problems and do not expect the problem's formulation to of that strict a format [68]. Algorithms of that kind often contain strategies to balance the exploratory and exploitative search behavior. The exploration aspect tries to find promising, areas within the (complex) search space containing good solutions, while the exploitation feature tries to specify the exact solution in those promising areas, also to

accumulate experience. This principle is one of the main distinctions between different metaheuristics and their configuration [5]. Therefore, coming from the Greek prefix *meta*, roughly translated as *high-level*, a metaheuristic can be understood as a "high-level problem-independent algorithmic framework" [68] that is capable of employing strategies to generate processes of heuristic nature that are able to escape local optima as well as robustly search a solution space. The latter is especially true for population-based metaheuristics [25]. The framework perspective is particularly important, because the general descriptions of metaheuristics often include certain operations that are combined to achieve the above mentioned functionality. Therefore, a metaheuristic can be more of a concept for designing an algorithm, rather than a strict specification of an implementation. This understanding also gave rise to so-called "hybrid metaheuristics", which mix and match ideas from several frameworks into one algorithm [69].

Giving a definitive taxonomy of metaheuristics is unpractical, because of the many characteristics they can be distinguished by. One of the more common ones are based on...

- ...solution creation quantity: Single-Solution Based vs. Population-Based
- ...solution creation process: Deterministic vs. Probabilistic/Stochastic
- ...how solutions are manipulated:
  Local Search vs. Constructive vs. Population-Based [68]
- ...which analogy they belong to:
  Bio-Inspired vs. Physics-Based vs. Evolutionary vs. Swarm-Based

Although all of these classifications are justified, I will not rely on any one of them, as it would serve no purpose to limit the discussion to one category. When necessary, algorithms will be placed within these taxonomies, to explain their purpose respectively.

3.2.1 Overview

The first classification mentioned (Single-Solution Based vs. Population-Based) will be used to give a short overview of the field, because of the intuitive separation it creates.

Single-Solution Based Metaheuristics

Single-Solution Based Metaheuristics (S-metaheuristics) improve on a single, initial solution and describe a trajectory while moving through the search space. Hence, often also referred to as *trajectory methods*. The dynamics applied to each new iteration of solutions depend heavily on the specific strategy. However, they generally can be described by a generation procedure, where new candidate solutions are generated from the current one, and a replacement procedure, where one of the new solutions is selected based on some criteria [72]. A very important aspect in finding new solutions during the first phase is the neighborhood structure. It represents the accepted area in the search space around the current solution, and the definition is usually very much dependent on the problem it is associated with. A larger neighborhood may increase the chance of "jumping" over local optima but at the expense of more computational effort.

Algorithm 3.1 Basic Local Search

```
s ← GenerateInitialSolution( )
repeat
    s′ ← Generate(N(s))
    if Select(s′) then
        s ← s′ ∈ N(s)
until termination criterion met
```

Algorithm 3.1 shows a high-level description for one of the first and simplest S-metaheuristic, the Basic Local Search. The functions (generate, select) and neighborhood N vary depending on the implementation. The generation of new solutions can, for example, be of deterministic or stochastic nature, while the selection of a candidate may be based on the best improvement found in the entire neighborhood or based on the first improvement that occurs [72]. The greatest issue with Basic Local Search is the convergence into local optima and most other implementations of a S-metaheuristic enhance this algorithm to counteract that flaw in some way [4].

One of these is Simulated Annealing (SA), based on the physical process of annealing. If a metal is heated above its recrystallization temperature and is then cooled in a controlled manner, its atoms are able to reside into a lower energy state, altering the metals properties. This analogy is used to simulate a temperature that controls the magnitude of change between possible solutions. A higher temperature allows for worse solutions than the

current one, which, hopefully, allows for "climbing" out of local optima. As the algorithm progresses, the virtual temperature cools down, decreasing the chances for such uphill moves and, eventually, SA converges into a simple local search algorithm [4].

Tabu Search (TS), on the other hand, makes use of a list (*memory*) to explicitly utilize the search history to its benefit. In the simplest form, TS uses a *short term memory* to remember the most recent solutions, limiting the neighborhood to solutions not present in the list. Therefore, larger lists force the algorithm to explore larger search spaces. Other, more recent S-metaheuristics algorithms include *Iterated Local Search* (ILS), *Greedy Randomized Adaptive Search Procedure* (GRASP) and *Variable Neighborhood Search* (VNS) [72].

Population-Based Metaheuristics

The common aspect with Population-Based Metaheuristics (P-metaheuristics) is that they maintain an entire set (*population*) of solutions. This presence of multiple solutions results in an intrinsic drift toward a more exploratory algorithmic behavior [4]. Although they do not share the same algorithmic background as many S-metaheuristics, they have similarities when it comes to improving their populations. Starting with a population generation phase, new solution sets are iteratively created by each member of the population (generation phase) and then incorporated into or even replaced with the current population (replacement phase) until a stopping criterion is satisfied. The last two of these three phases may even use some sort of solution history to build their functions, or operate completely memoryless. The choice of initial population also plays a significant role in the diversification behavior and computational cost of the algorithm [72].

Algorithms with analogies to swarm intelligence, often found in animals in nature, are an example of P-metaheuristics. These types of algorithms usual work via agents which, individually, are not complex and employ simple operations to build a solution. However, they exchange information and move cooperatively through the problem space. Two examples of swarm-intelligence based algorithms are described in the next two subsections. Other popular examples of P-metaheuristics are Evolutionary Algorithms (EAs), that refer to biology-inspired operations, like mutation and recombination, to manipulate their population, or Scatter search (SS).

3.2.2 Ant Colony Optimization

The Ant Colony Optimization (ACO) in its original form ("Ant System" as proposed by Dorigo et al. [15]) is a multi-agent metaheuristic inspired by the foraging behavior of real ants. Through ethological studies it was found that ants, although nearly blind [1], are able to find very short paths between a food source and their nest [27]. This is achieved by a chemical substance produced by the ant called pheromone, which is deposited along the path it has traveled. Without any pheromone information to guide them, ants travel mostly at random. However, when an ant encounters a pheromone trail, there is a high probability of following it and thus reinforcing this path with its own pheromone. This autocatalytic (positive feedback) behavior is counteracted by the pheromones volatility, which dissipated over time, weakening path reinforcement [15]. This results in the shorter paths accumulating higher amounts of pheromone, as shown in Fig. 3.2. This example also shows the reaction to a dynamic change in the path, which is an obstacle placed directly on the shortest route. Although the left path is at first equally probable as the right path, the reinforcement on the shorter right path is greater, causing the ants to adapt to the obstacle [72]. That said, the removal of the obstacle in this example would not lead to a return to the old path, because of the already reinforced pheromone trail [16].

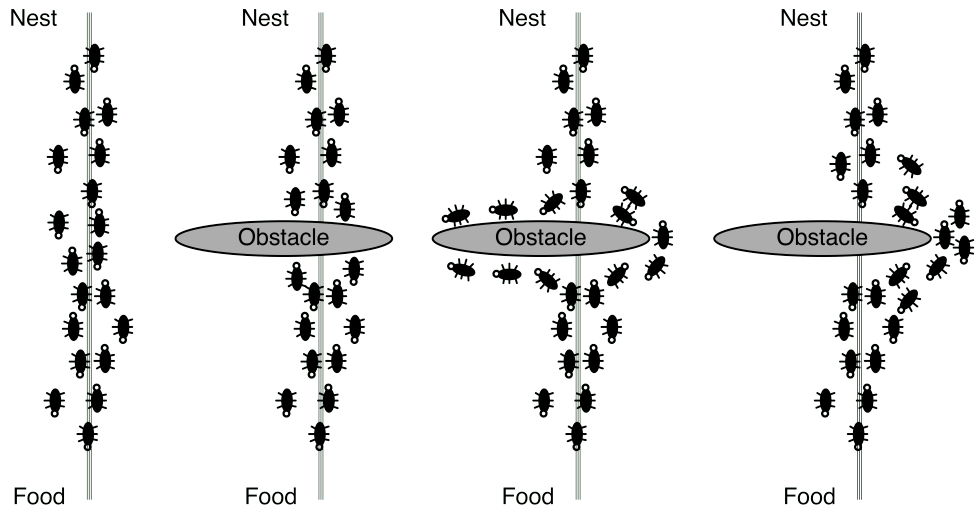


Figure 3.2: The process of ants following a pheromone trail between a food source and their nest affected by an obstacle. Modified from Talbi [72].

[1] The visual ability of ants varies according to species and function in the colony. The studied Argentine ant (Linepithema humile) has very poor vision [72].

The artificial ants in ACO are inspired by that behavior, iteratively building a solution by probabilistically choosing the next part based on heuristic, problem-specific information, and, analogous to the real ants, artificial pheromone information. The use of multiple agents also gives the algorithm more robustness and diversification in solving a problem [17].

Algorithm 3.2 Ant Colony Optimization

```
Initialize pheromone information
repeat
    for each ant do
        ConstructSolution
    procedure UpdatePheromones
        Evaporation
        Reinforcement
until termination criterion met
```

The algorithmic implementation is quite simple. And since its original use was often adapted for the TSP, and this thesis solves that problem as well, the following description is slightly modified to fit it. Algorithm 3.2 shows the basic template of the ACO. First, the pheromone information is initialized evenly, so that every path is equally likely to be chosen at first. With a total of n cities to visit, the resulting matrix is of dimension $n \times n$, with each entry τ_{ij} representing the amount of pheromone being present on the edge (i, j). For every complete cycle, each of m ants probabilistically creates a solution with this pheromone information τ_{ij} and heuristic information η_{ij}. Starting from a randomly selected city i, the probability to visit the next possible city j from a set S of not yet visited cities is given by

$$p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{h \in S} \tau_{ih}^{\alpha} \cdot \eta_{ih}^{\beta}} \tag{3.3}$$

where η_{ij} holds the problem specific heuristic value, which is the inverse distance $\eta_{ij} = \frac{1}{d_{ij}}$ between cities i and j in case of the TSP. The constants $\alpha, \beta \geq 0$ control the influence of either the stochastic pheromone or the heuristic value respectively. The denominator normalizes the fraction into a probability $0 \leq p_{ij} \leq 1$, with the set of all probabilities from the unvisited cities S effectively creating a probability distribution [72].

The pheromone information is then updated based on the generated solutions. First, every pheromone entry is subject to evaporation controlled by a constant value $\rho \in (0, 1]$ and defined in the following equation:

$$\forall i, j \in [1, n] : \tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} \tag{3.4}$$

In the following reinforcement phase different strategies can be applied to select how the solutions selected by the ants influence the pheromone matrix. There are also versions of the ACO where this procedure is called after each step of the solution construction or at least after a single ant, but not necessarily every ant is finished constructing their solution. More common, however, are offline strategies called after every ant, is finished. One of the easier implementations in this category is the "Elitist pheromone update", where updates to the pheromone matrix are heavily influenced by the global best solution. Another approach is the "Quality-based pheromone update" or "ant-cycle" [15], where every ant $k = 1, 2, ..., m$ updates the pheromone matrix relative to the length of solution L_k they found in that iteration, which is further controllable with a parameter $Q \geq 1$. The following two equations define this strategy:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{3.5}$$

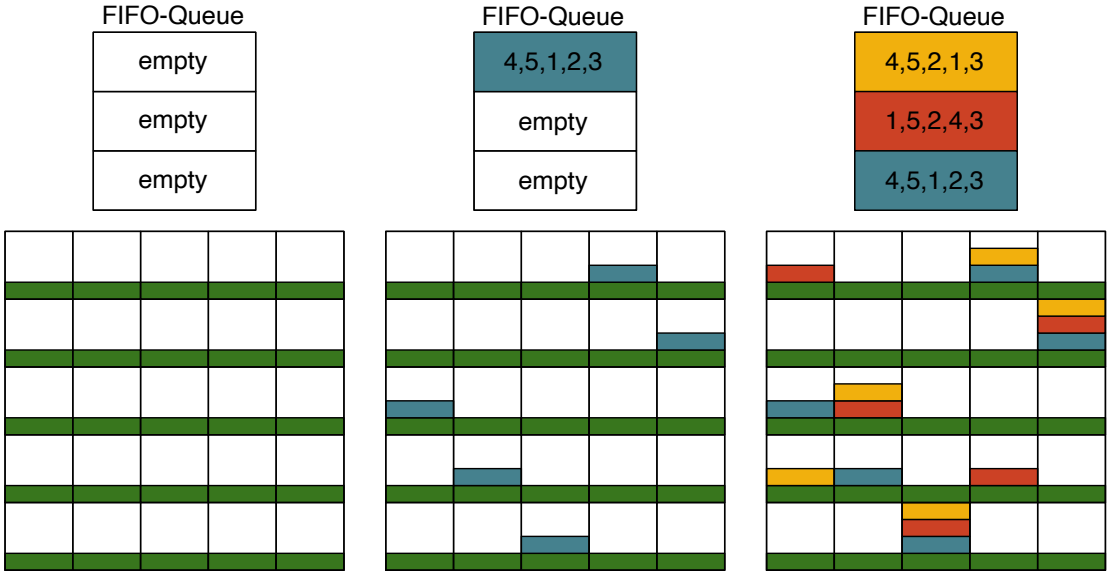
$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k}, & \text{if edge } (i, j) \text{ is in } k\text{-th ant tour,} \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

This loop is repeated until a termination criterion is met. This can be a fixed number of iterations $\mathrm{NC}_{\mathrm{MAX}}$ or until the solution quality stagnates for a certain period of iterations. With the pheromone update implemented as "ant-cycle" the time complexity of this algorithm is $O(\mathrm{NC} \cdot n^2 \cdot m)$ [15].

Based on this algorithm, a large number of variants were created. One of them is a population-based approach, which will be discussed in the following.

Population-Based Ant Colony Optimization

The Population-Based Ant Colony Optimization (PACO) was first proposed by Guntsch and Middendorf [28] as a simplification of the ACO, effectively reducing the operations necessary to update the pheromone information by quantifying and limiting the values added to the matrix. Their motivation was to speed up the process and decrease the



(a) Matrix at $i = 0$ **(b)** Matrix at $i = 1$ **(c)** Matrix at $i = 3$

Figure 3.3: Example of a PACO matrix being updated over multiple iterations for a solution population size of $k = 3$. The rows represent the location of the solution ranging from 1 to n, while the columns depict the option chosen (e.g., a city node for the TSP).

influence of older solutions in order to apply the PACO to the DTSP. Instead of saving all updates to the pheromone matrix, the approach keeps track of the solutions that updated the solution in a queue, referred to as *population*. Therefore, every solution in that queue is actually present in the matrix and vice versa. In order to limit the influence of old solutions, the population size is set to a value k. When the queue is full after k iterations, multiple actions are possible in iteration $k + 1$, with the most obvious being to implement a FIFO (first in, first out) behavior, discarding the oldest solution. This also eliminates the need for evaporation. The rest of the algorithm is analogue to the ACO presented before. The matrix is initialized with constant amount τ_{init}. The pheromone update is done by the ant with the iteration best solution. With a weight $w_e \in [0, 1]$ controlling the amount of pheromone deposited and a maximum set to $\tau_{\max}$, the amount of pheromone added is defined by $(1 - w_e) \cdot (\tau_{\max} - \tau_{\text{init}})/k$. This reduces the number of pheromone updates per generation, for a TSP instance of n cities, from n^2 operations to at most $2n$ operations [28].

Fig. 3.3 shows an example of a pheromone matrix with solution population of $k = 3$ being updated over the course of three iterations for a problem of size n. In Fig. 3.3a the population is empty and the matrix is initialized with τ_{init}, visualized by green bars. The first iteration's best solution has the city referenced at position 4 as its start, continuing

with position 5, and so on. This update is visualized in blue colored bars. Eventually, after two more iterations (Fig. 3.3c), the population queue is full. In a next iteration the solution visualized in blue would leave the matrix, with a new one being placed on top of the queue.

3.2.3 Particle Swarm Optimization

The Particle Swarm Optimization (PSO) method was introduced by Kennedy and Eberhart [41] as a "concept for optimization of nonlinear functions" by simulating swarms of birds or fish in their search for food. The individuals, referred to as *particles*, iteratively explore a given problem space of dimension d. Therefore, each of the N particles in a swarm represent a candidate solution to the problem, evaluated by an objective fitness function $f : \mathbb{R}^d \rightarrow \mathbb{R}$. Furthermore, each particle i is defined by three vectors and two values:

- the current position $\vec{x_i} \in \mathbb{R}^d$
- the current velocity $\vec{v_i} \in \mathbb{R}^d$
- the best solution found so far $\vec{p_i} \in \mathbb{R}^d$
- the fitness values $f(\vec{x_i})$ and $f(\vec{p_i})$

In order to let the particles influence each other, a neighborhood rule needs to be defined. The most straightforward way is the global best method (**gbest**), where all particles influence each other without restrictions. Another, potentially more complex, strategy to let the particles exchange information is the local best method (**lbest**). In this method, particles interact based on a given topology, such as a ring, on which only direct neighbors exchange information. Thus, regardless of the strategy chosen, each neighborhood k has a leader with the best solution $\vec{g_k}$ [72]. Putting both aspects together, the particles update their velocity based on personal success (*cognitive aspect*) and their neighborhoods success (*social aspect*) [35].

Algorithm 3.3 shows a high-level description of the PSO procedures. Typically, the swarm is randomly initialized, having each particle assigned a velocity and position in the search space. The resulting solutions are set as $\vec{p_i}$ and, for each particle's neighborhood k, the leader solution $\vec{g_k}$ is determined. After the initialization, each particle i updates its velocity $\vec{v_i}$ per iteration t according to the following equation:

$$\vec{v_i}(t+1) = w \cdot \vec{v_i}(t) + c_1 \cdot r_1 \cdot (\vec{p_i} - \vec{x_i}(t)) + c_2 \cdot r_2 \cdot (\vec{g_k} - \vec{x_i}(t)) \tag{3.7}$$

Algorithm 3.3 Particle Swarm Optimization

Initialize swarm
repeat
  for all particles $i \in [1, N]$ **do**
    UpdateVelocities
    UpdatePosition
    if $f(\vec{x}_i) < f(\vec{p}_i)$ **then** $\vec{p}_i = \vec{x}_i$
      if $f(\vec{x}_i) < f(\vec{g}_k)$ **then** $\vec{g}_k = \vec{x}_i$
until termination criterion met

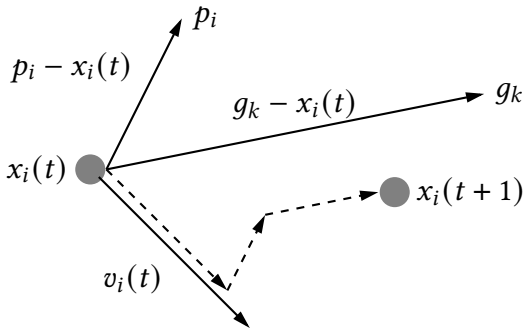


Figure 3.4: The vector summation of a PSO particle i during velocity and position update.

with an inertia weight $w > 0$ controlling the influence of the particles velocity. The parameters $c_1, c_2 > 0$ define the impact of the personal best $\vec{p}_i$ and neighborhood best solution $\vec{g}_k$, respectively and are additionally subject to a random factor due to values r_1, r_2 drawn from a uniform distribution of $[0, 1]$. Afterwards, the particle's position is updated:

$$\vec{x}_i(t+1) = \vec{x}_i(t) + \vec{v}_i(t+1) \tag{3.8}$$

Lastly, the best solutions are potentially updated if they are better than the current ones. To ensure convergence of the swarm, many implementations also limit the velocity to a certain value or reduce the inertia weight over time. This whole process is visualized in Fig. 3.4, showing one particle i being updated according to all three possible influences.

Research regarding the behavior of the algorithm and variations have been proposed. Especially the choice of neighborhood topology has a significant impact on performance. The **gbest** method seems to perform better on unimodal problems, where one distinct function optimum is present, while the **lbest** method results in better performance for multimodal problems, with many optimal function values [35].

One such variant is the Hierarchical Particle Swarm Pptimization (H-PSO) proposed by Janson and Middendorf [35], which uses a hierarchical tree topology (indicating solution quality) to define a neighborhood. The resulting (almost) regular tree has a total number of m nodes over a height h, where each parent node has at most d children. The particles are represented by the tree's nodes, and, therefore, the neighborhood of each particle i is defined only by its direct parent j in that hierarchy, so $\vec{g}_i = \vec{p}_j$. The updating of velocity and position remains the same as per the standard approach but the comparison and eventual update of the neighborhood best solution $\vec{g}_k$ can be seen as a tree restructuring process. If a child node i happens to find a better solution than its parent node j, so $f(\vec{p}_i) < f(\vec{p}_j)$, they swap their position in the hierarchy. This process is performed top-down, breadth-first, resulting in worse solutions may moving down several levels in an iteration, but good solutions moving up at most one level. The global best solution can then be located at the root position after a maximum of h iterations.

Another variant is called Simplified Swarm Optimization (SSO), which is a discrete version of the PSO, able to solve combinatorial problems like the *MMRAP* (multi-level, multi-state redundancy allocation problem), as proposed by Yeh [80]. The particle's solution is no longer represented by the position and the velocity in multi-dimensional problem space, but is instead encoded as a finite-length string and an additional fitness value. The update mechanism has also been simplified by probabilistically selecting the next position string based on a random number that may lie in multiple tuneable intervals [81]. These intervals are the probabilities for the personal best solution p_{pbest}, the personal previous solution p_{pprev} and the global best solution p_{gbest}. The resulting solution vector $\mathbf{s}_j^t$ for particle j at iteration t is therefore built according to

$$s_{ij}^{t+1} = \begin{cases} s_{ij}^t & \text{with propability } p_{\text{pprev}}, \\ p_{ij}^t & \text{with propability } p_{\text{pbest}}, \\ g_i^t & \text{with propability } p_{\text{gbest}}, \\ v & \text{with propability } p_r \end{cases} \tag{3.9}$$

with s_{ij} being the i-th solution component of particle j and $v \in V$ being a value from a set of all feasible values chosen with probability p_r [49].

3.2.4 SPPBO Framework and H-SPPBO Algorithm

Simple Probabilistic Population-Based Optimization

The Simple Probabilistic Population-Based Optimization (SPPBO) is a metaheuristic scheme combining generalized aspects from population based approaches, like the PSO and SSO, and strategies effectively creating probability distributions, like the ACO and PACO. As discussed by Lin et al. [49], the framework can be used to classify and virtually recreate many of these metaheuristics for solving discrete combinatorial problems by using two simple operations:

- *SELECT+COPY*: Selecting a solution from the population and copying (parts of) this solution when certain conditions apply.
- *RANDOM*: Random selection of a value from the set of possible values.

These operations can be used to create multiple variants of an SPPBO algorithm as well as de facto implementations of PACO and SSO. However, all of them share the same schematic foundation. First, a distinction between population and Solution Creating Entities (SCEs) needs to be made. Reminiscent of an ant (ACO) or a particle (PACO), a set of SCEs $\mathcal{A}$ create the solutions through application of probabilistic rules Prob_V and, optionally, some form of heuristic information η. These solutions may then enter some set of populations $\mathcal{P}$ (cmp. PACO). V denotes the set of possible values $v_i \in V$ that may appear in a solution vector $\mathbf{s} \in V^n$ of length n. The high-level structure of these SPPBO metaheuristics can be found in Algorithm 3.4.

Algorithm 3.4 SPPBO

```
Initialize random solutions
repeat
    for all SCE A ∈ 𝒜 do
        CREATE(A)
    for all population P ∈ 𝒫 do
        UPDATE(P)
until termination criterion met
```

After the populations are initialized with random solutions, each SCE creates one solution, resulting in $k_{\text{new}} = |\mathcal{A}|$ new solutions per iteration. The underlying probability distribution is influenced by three aspects:

- The populations in the SCE's neighborhood ($\text{Range}_p \subseteq \mathcal{A}$), realizing the *SELECT+COPY* operation.
- The set of feasible values V, realizing the *RANDOM* operation.
- The problem-specific heuristic information η.

Additionally, the influence of the *SELECT+COPY* and *RANDOM* operations of the population can be further controlled by the weights w_p and w_r. Afterwards, the populations are updated based on a set of rules specific to the implementation. For example, in an SPPBO version with only one global best population, the update procedure may insert the iteration best solution, and saving a total of k iterations [49].

Hierarchical Simple Probabilistic Population-Based Optimization

Building on the schematic foundation of the SPPBO, the Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO) was designed using its principles. Combining the hierarchical aspect of H-PSO with a probabilistic solution creating approach similar to SSO, while maintaining a population of solutions, like PACO, the goal of this algorithm, as stated by Kupfer et al. [42], was to not only solve the DTSP, but also detect these dynamic changes to react accordingly.

Similar to the descriptions regarding SPPBO, there is a set $\mathcal{A}$ of SCEs and a set of populations $\mathcal{P}$. The important distinction here is, that these populations $P \in \mathcal{P}$ each belong to an SCE $A \in \mathcal{A}$, which is described by a function $P \in \text{range}(A) \subseteq \mathcal{P}$, that dynamically adapts to changing neighborhood relations. In order to use these parent-children relation in the population, the hierarchical aspect is implemented similar to H-PSO (see 3.2.3). The SCEs are organized in an m-ary tree[2], with every "child SCE" influenced by their parent(A), and a "root SCE" $A*$ defined as its own parent (parent($A*$) = $A*$). This hierarchy allows for a clear definition of the following populations $P \in \mathcal{P}$ for each SCE A:

- the personal previous solution P^A_{persprev}
- the personal best solution P^A_{persbest}
- the parent best solution $P^A_{\text{parentbest}}$

[2] A tree structure in which every node has at most m children, with no restriction on height. For example, a value of $m = 2$ would result in a binary tree.

Each population contains exactly one solution vector $\mathbf{s} \in V^n$ (e.g., for a TSP instance of size n) from the set of all feasible values V. Also, note that due to the tree structure $P^A_{\text{parentbest}} = P^{\text{parent}(A)}_{\text{persbest}}$. Each of these populations have a corresponding weight $w_{\text{persprev}}, w_{\text{persbest}}, w_{\text{parentbest}} \geq 0$ to control their respective influence. Now, to create a solution $\mathbf{s}$ the probabilistic rule is very similar to the one used by SSO seen in Eq. (3.9), with p_{gbest} referring to the parent's best solution and a distinct random influence p_r through a random weight w_{rand}.

The following description of the algorithm and equations are adapted to fit the TSP, because the H-SPPBO was initially created to solve the TSP and its dynamic variant. A modification to solve other combinatorial problems, e.g., the Quadratic Assignment Problem (QAP), would be straightforward, but will not be discussed in this thesis.

Algorithm 3.5 H-SPPBO

Initialize the SCE tree
Initialize SCEs with random populations P^A_{persprev},P^A_{persbest}
repeat
  for all SCE $A \in \mathcal{A}$ **do**
    CreateSolution(A) // using (3.10) and (3.11)
    UpdatePopulations($P^A_{\text{persprev}}, P^A_{\text{persbest}}$)
  swapNum = 0
  for all SCE $A \in \mathcal{A}_{\text{tree}}$ **do**
    if $f(\mathbf{s}^A_{\text{persbest}}) < f(\mathbf{s}^{\text{parent}(A)}_{\text{persbest}})$ **then**
      Swap(A, parent(A))
      swapNum $\leftarrow$ swapNum + 1
  if swapNum $> [\theta \cdot |\mathcal{A}|]$ **and** no change in L previous iterations **then**
    ChangeHandlingProcedure(H)
until termination criterion met

Algorithm 3.5 shows the process to solve a dynamic problem. It is similar to the template given for SPPBO (see Algorithm 3.4), with a solution creation and population update phase. However, because the populations are directly related to the SCEs, the update phase is executed in the same loop. First, the SCE tree is initialized with a number of $|\mathcal{A}|$ randomly set SCEs and their two solution populations. Then, every iteration, each SCE creates one solution using the following procedure: Begin with a set of all unvisited

nodes $U \subseteq V$ and set a random start node i. Now, calculate the following term τ_{ik} for all possible nodes $k \in U$ by

$$\tau_{ik}(A) = \left(w_{\text{rand}} + \sum_{P \in \text{range}(A)} w_P \cdot s_{ik}(P) \right)^{\alpha} \cdot \eta_{ik}^{\beta} \tag{3.10}$$

$$s_{ik}(P) = \begin{cases} 1 & \text{if } (i,k) \subset \mathbf{s}_P, \\ 0 & \text{otherwise} \end{cases}$$

with range(A) giving all three populations assigned to the SCE as mentioned above, and a heuristic component η_{ik} set as the inverse distance $1/d_{ik}$ between nodes i and k, which is typical for the TSP. This τ-term effectively accumulates all the different weights, by using $s_{ik}(P)$ as an activation function for checking, if the current, possible edge (i,k) was also visited previously by the SCE ($P^A_{\text{persprev}}, P^A_{\text{persbest}}$) or its parent ($P^A_{\text{parentbest}}$), i.e., formally, if the ordered set (i,k) is a subset of the solution vector $\mathbf{s}_P$ of population P. The following example is given for clarification:

Example 3.2.1 (A subset of an ordered set)
Using the TSP instance from Fig.3.1, a previous solution of a SCE might be $\mathbf{s}_{persprev} = (4,2,3,1) \in V^4$. *Then,* $(4,2)$ *would be an ordered subset of that solution,* $s_{4,2}(P_{persprev}) = 1$.

As with the ACO metaheuristic, $\alpha, \beta \geq 0$ are parameters to control the stochastic and heuristic influence respectively. The probability for visiting node j after node i can now be defined by

$$p_{ij}(A) = \frac{\tau_{ij}(A)}{\sum_{k \in U} \tau_{ik}(A)} \tag{3.11}$$

where the denominator is used to normalize this term into a probability distribution over all unvisited nodes U. Finally, a node j will randomly be drawn from that distribution, added to the solution vector $\mathbf{s}$ and removed from the unvisited set $U \leftarrow j \setminus U$. With this new node being the next current node, the process is repeated until the set of unvisited nodes is exhausted. And eventually, the populations are updated.

With new solutions calculated, the hierarchy is now subject to change. The SCE tree is iterated in a top-down, breadth-first manner ($\mathcal{A}_{\text{tree}}$) and every SCE compares its personal best solution with its parent by using an evaluation function $f : V^n \to \mathbb{R}_0^+$, which in case of the TSP, is just the length of the tour L. If the child has a better solution quality than its parent, so $f(\mathbf{s}^A_{\text{persbest}}) < f(\mathbf{s}^{\text{parent}(A)}_{\text{persbest}})$, they swap their places. Thus, making the range function dynamically changing by also swapping the $P^A_{\text{parentbest}}$ population. An example of said swap operation is given in Fig. 3.5. By using this top-down approach, comparatively

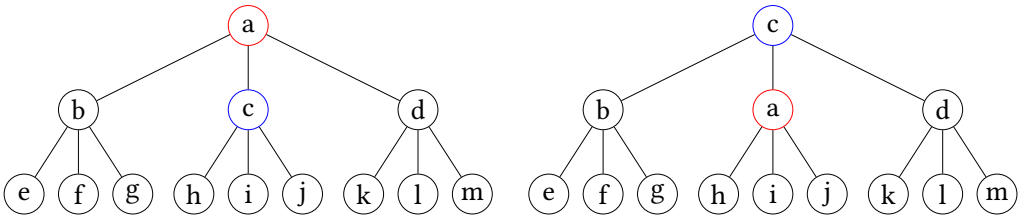


Figure 3.5: Example of a ternary SCE tree ($m = 3$) with height $h = 3$ showing the SWAP(A,C) operation before (left) and after (right) the execution. Modified from Kupfer et al. [42].

worse solutions can descend all the way to the bottom level in one iteration, while good solutions may only be able to move up one tier. Nevertheless, if no new personal best solutions have been found in at least as many iterations as the number of levels of the tree h, the global best solution is able to move to the root of the SCE tree.

Since the H-SPPBO should also detect and react to dynamic changes in the TSP instance, the last part of the algorithm is executed, if a certain threshold of rearrangements in the SCE tree is exceeded. Specifically, the number of swaps from the previous part is being compared against a percentage of SCEs in the tree $|\mathcal{A}|$. This is controlled by a constant $\theta \in [0, 1]$, with a higher value reducing the detection sensitivity and an extreme of $\theta = 1$ needing the whole tree to fully rearrange to render the condition true. However, this condition may be met without the problem instance actually changing, leading to false detections. Regardless, if the change handling procedure is triggered by "enough" change, one of two H mechanisms is executed to alter the SCEs, ideally, aiding in creating better solutions to this (possibly) modified problem. The strategies are the following:

- H_{full} resets the P^A_{persbest} population for all SCEs $A \in \mathcal{A}$ to a random solution.
- H_{partial} resets only the P^A_{persbest} population of the SCEs starting from the third level down, leaving the the root and its children unaltered.

Whereas H_{full} can be understood as a complete reset of the algorithm, H_{partial} tries to apply some of the best "knowledge" to solve this changed problem (exploitation), while the lower performing SCEs may instead concentrate on finding new solutions (exploration). Additionally, due to the possibly major rearrangements in the tree as a result to these change handling procedures being executed, the parameter $L > 0$ acts as a guardrail to prevent the procedure from virtually triggering itself, providing the hierarchy some iterations to settle [42].

3.3 Parameter Optimization for Metaheuristics

3.3.1 Overview

Every metaheuristic has at least a few parameters to control its behavior. This is not only a byproduct of ambivalent algorithm design, but more often to allow more flexibility in solving multiple problems with different qualities. For example, looking at the ACO (see 3.2.2), there is the number of ants m, the trail persistance rate ρ, the initial amount of pheromone τ_0, the relative quantity of pheromone added Q, and the importance of stochastic aspects α and heuristic information β. And as mentioned in Section 2.2, the performance of these algorithms depends heavily on optimal parameters, without a priori knowledge of which settings to choose.

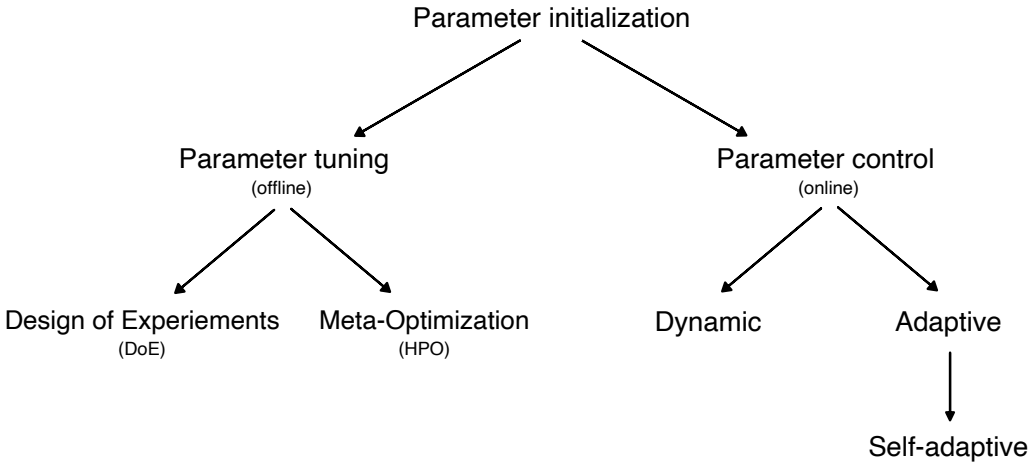


Figure 3.6: Taxonomy of parameter optimization. Modified from Eiben et al. [18].

This thesis applies methods from machine learning research to address said challenge. Therefore, the following review of parameter optimization methods will classify this technique and identify other possible strategies. Figure 3.6 shows a taxonomy of parameter optimization methods by Eiben et al. [18], with additions from Talbi [72] and Stützle et al. [71]. Two main distinctions can be made here: "Parameter Tuning" (offline initialization) tries to find good parameter settings before the algorithm is even applied, and these settings remain fixed afterwards. "Parameter Control" (online initialization) modifies the parameters during algorithm runtime, allowing for potentially better adaptation to the problem.

Parameter Control

Parameter control methods can be beneficial because they are able to adapt to the problem instance over time. They can also be used to encourage certain search phases of the algorithm, increasing either the exploratory or the exploitative behavior after a certain time. Several ways to implement such online parameter initialization methods have been proposed. *Deterministic* strategies alter the values by a specific deterministic rule, while also allowing for minor random influences to the parameters. *Adaptive* optimization, on the other hand, takes additional feedback from the metaheuristic algorithm to control the weight for its change, but still relies on predefined functions for that. Lastly, *self-adaptive* parameter control implements the parameter change into the metaheuristic algorithm itself, making them part of its search space and, therefore, the solution.

Parameter Tuning

The simplest and oldest method of parameter tuning is a manual trial-and-error approach. By tuning one parameter at a time, each parameter is evaluated on its own, but without considering the interactions between them. This also becomes a very time consuming and error prone process as the parameter space grows. Two methods address these issues with offline tuning. The first is a Design of Experiment (DoE) approach to determine the minimum required scope of the experiments needed and to arrange the test points within the parameter space based on an optimality criterion. The result of these experiments should reveal a good set of parameters, with possibility for further statistical significance tests. The other method uses meta-optimization, that is, an algorithmic level on top of the metaheuristic to find optimal parameter values. This can be either a metaheuristic algorithm (like PSO) or a machine learning based approach, which is discussed in the following subsection.

3.3.2 Hyperparameter Optimization

In machine learning (ML) applications hyperparameters are used to configure a ML model. Much like with parameters found in metaheuristics, tuned hyperparameters can greatly improve the performance of a ML model over the default settings. Especially state-of-the-art deep learning techniques, which have a vast amount of tunable parameters, can only really be sensibly deployed, if the hyperparameters are specifically tuned for their problem.

Traditionally, optimization problems often used simple gradient descend-based approaches. In its simplest form, the objective function $f(x)$ is to be minimized by $\min_{x \in \mathbb{R}} f(x)$, with a region of feasible values $D = \left\{x \in X | g_i(x) \leq 0, h_j(x) = 0\right\}$, and possible range and equality constrains $g_i(x), h_j(x)$ from a set of all values X. Following the negative gradient direction, a global optimum could be obtained for convex functions [79]. However, ML models pose some challenges for these tried-and-tested techniques:

1. The underlying objective function of a ML model is usually not convex nor differentiable. Therefore, methods like gradient descend often result in local optima.
2. The hyperparameters of ML models are in the domain of continuous (real-valued), discrete (integer-valued), binary, categorical and even conditional values. This results in a complex configuration space with sometimes non-obvious value ranges.
3. Objective function evaluations can be very expensive, which necessitates methods for quicker, more efficient sampling.

Parallel to the research on optimal parameters for metaheuristics (see 2.2), the manual tuning of hyperparameters in ML applications began to be unfeasible in light of more complex and feature-rich models. Therefore, interest in the automated tuning of hyperparameters, called Hyperparameter Optimization (HPO), began in the 1990s. With important use cases being the reduction of human effort, the improved performance of these algorithms and, especially in scientific research, to help reproducibility, much progress has been made since the above mentioned gradient descent-based methods were initially applied [21].

The HPO problem statement can be formulated as follows [21]: Let $\mathcal{A}$ be a ML algorithm with N hyperparameters, with Λ_n denoting the n-th hyperparameter. The complete hyperparameter configuration space can then be defined as $\mathbf{\Lambda} = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_N$, with a vector of a possible hyperparameter configuration $\lambda \in \mathbf{\Lambda}$. Therefore, a ML algorithm initialized with hyperparameters λ is denoted by $\mathcal{A}_\lambda$. Based on that, the process of HPO consists of four main components: 1) an estimator (most often a regressor, but a classifier is possible too), 2) a search space $\mathbf{\Lambda}$, 3) a method to select configurations from the search space, and 4) a validation protocol $\mathcal{V}$ and its loss function $\mathcal{L}$ to evaluate the configurations performance (e.g., error rate or root mean squared error). The goal is then to find the optimal hyperparameter set λ^* on a given data set $\mathcal{D}$ that is split into training data D_{train} and validation data D_{valid} that minimizes the error $\mathbb{E}$ from the validation protocol:

$$\lambda^* = \underset{\lambda \in \mathbf{\Lambda}}{\operatorname{argmin}} \, \mathbb{E}_{(D_{\text{train}}, D_{\text{valid}}) \sim \mathcal{D}} \left[\mathcal{V}(\mathcal{L}, \mathcal{A}_\lambda, D_{\text{train}}, D_{\text{valid}}) \right] \tag{3.12}$$

Almost all of this also applies to the parameter optimization for metaheuristic algorithms. Here, the loss function $\mathcal{L}$ is often closely tied the objective function itself, e.g., the resulting tour length in solutions for the TSP. Therefore, HPO for metaheuristics has no need for any supervised data sets or ground truths. Applied to metaheuristics solving the TSP, with the solution quality function $f(\mathbf{s})$ from Equation (3.1), the HPO goal from Equation (3.12) can be defined as follows:

$$\lambda^* = \underset{\lambda \in \mathbf{\Lambda}}{\operatorname{argmin}} \, \mathbb{E}\left[\mathcal{V}(f, \mathcal{A}_\lambda)\right] \tag{3.13}$$

with $\mathcal{A}_\lambda$ now being the metaheuristic algorithm initialized with parameters λ. To simplify all further explanations, let the objective function $f : \mathbf{\Lambda} \to \mathbb{R}$ be defined directly by its parameter space $\mathbf{\Lambda}$, which is mapped into the real solution quality space $\mathbb{R}$.

There are multiple ways to solve the above mentioned problem statements. A common distinction between these methods is their usage of the objective function and its loss landscape. An HPO algorithm can either treat this as a black-box function, using full evaluations of the objective function to model its behavior, or approach the problem with so-called "multi-fidelity optimization", where the model is too complex and computationally expensive to use for evaluation and a cheap (possibly noisy) proxy is used instead [21]. Another important distinction, specifically in the field of black-box HPO, is whether or not to use a statistical model, e.i., an estimator. Model-free techniques solely rely on function evaluation and factual improvement for their optimization process without using any prior knowledge to sample the search space. Model-based methods, however, employ a more sophisticated strategy, often implemented using so-called Bayesian Optimization (BO), which is explained in the next subsection.

The following subsections give four examples of black-box HPO, with one using model-free and the remaining using model-based, BO techniques. All of these methods were also used for the experiments discussed in latter sections of this thesis.

3.3.3 Grid Search and Random Search

A very simple and straightforward approach to HPO is Grid Search (GS). With its basic functionality similar to brute-force methods, all possible combinations of hyperparameter values are evaluated. GS only requires finite sets of value ranges to be defined for each hyperparameter. It then iterates over the whole search space by creating the Cartesian product of these sets. This approach is easily interpretable and repeatable, while also

being trivially implemented and parallelized [2]. However, each new parameter causes the Cartesian product to grow exponentially, making GS suffer greatly from the "curse of dimensionality". Another problem is its inability to explore promising value ranges on its own. Since these need to be pre-defined, a user would have to manually adjust these ranges prior to each run [79].

Random Search (RS) was proposed to overcome the limitations of GS, by sampling the hyperparameters from a probability distribution over the configuration space $F(\mathbf{\Lambda})$, eliminating the need for trying out all possible combinations [2]. In most cases, this probability distribution is simply uniform for all hyperparameters, but certain applications may warrant for a higher density in some regions of a hyperparameter. Algorithm 3.6 presents the pseudo-code for the RS algorithm. It basically compares each new function evaluation $f(\lambda_{\text{new}})$ on a randomly sampled parameter λ_{new}. The termination criterion is usually implemented as a fixed budget of function evaluations B, therefore giving each of N hyperparameters B different evaluations, as opposed to $B^{1/N}$ with GS (would it also operate on a fixed budget) [2]. This gives parameters with a higher partial dependency on the solution quality a much higher chance of finding a global optimum. It also shares the same advantages as GS: easy implementation, parallelization and reproducibility (given a fixed random number generator). On the other hand, RS (possibly) still evaluates unimportant search areas, since it has no guidance in its exploratory behavior, like a model-based HPO algorithm might have [79]. Nevertheless, it still serves as a well-performing baseline for many ML benchmarks, with hyperparameters relatively close to the optimum, given sufficient resources [21].

Algorithm 3.6 Random Search

Require: Probability distribution over parameter space $F(\mathbf{\Lambda})$
  Initialize parameters: $\lambda^* \leftarrow F(\mathbf{\Lambda})$
  repeat
    $\lambda_{\text{new}} \leftarrow F(\mathbf{\Lambda})$
    if $f(\lambda_{\text{new}}) < f(\lambda^*)$ **then**
      $\lambda^* \leftarrow \lambda_{\text{new}}$
  until termination criterion met
  return λ^*

3.3.4 Bayesian Optimization

BO is not only an algorithm used for HPO, but a complete framework for the global optimization of (expensive) black-box functions. It differs from methods like the aforementioned RS by incorporating prior knowledge about the objective function into the sampling procedure. The prior, i.e., the analyzed objective function $f : \mathbf{\Lambda} \rightarrow \mathbb{R}$, under assumption of the evident data it samples $\mathcal{D}_t = \{(\lambda_\mathbf{i}, f(\lambda_\mathbf{i}))|i \in [1, t]\}$ yields a likelihood $P(\mathcal{D}_t|f)$, which can then be combined with the prior distribution $P(f)$ leading to the posterior distribution through application of Bayes' theorem:

$$P(f|\mathcal{D}_t) \propto P(\mathcal{D}_t|f)P(f) \tag{3.14}$$

Essentially, this expresses the likelihood of the sampled data under the assumptions made for the objective function [8]. Applying this in practice, means that more sampled data points result in the posterior function to adjust its mean for these points, reducing its uncertainties and enhancing its predictive power [76].

Algorithm 3.7 Bayesian Optimization

```
InitializeModel(D_init),
Initialize parameters: λ* ← F(Λ)
y_min = f(λ*) + ε
for t ∈ [1, n_calls] do
    λ_t = argmax_λ u(λ|D_{t−1})                    // acquisition function
    y_t = f(λ_t) + ε_t
    D_t = {D_{t−1}, (λ_t, f_t)}
    UpdateModel(D_t)
    if y_t < y_min then
        y_min = y_t
        λ* = λ_t
return λ*
```

A common interpretation of this theory for effective implementations of BO is an iterative algorithm, given with Algorithm 3.7, consisting of two main parts: a surrogate model analogous to the posterior function over the objective and an acquisition function guiding the sampling process to the optimum. After initializing the model to an optional number of initial samples $\mathcal{D}_{\text{init}}$, a maximum of *n_calls* are made to the objective function f, with the following process for each iteration: First, the acquisition function $u(\lambda|\mathcal{D}_{t-1})$ uses the probability distribution of the surrogate and all previously sampled data points to assess

the search space, in order to find the most beneficial next point to sample. High acquisition corresponds to potentially optimal objective function values. Through choosing widely unsampled parameter areas, the function realizes exploratory behavior, while further samples in already well performing areas employs exploitative strategies. The specific choice of acquisition function always defines a (customizable) trade-off between these two [8]. With the most promising parameter input λ_t of iteration t specified, the (potentially) noisy objective function $f(\lambda_t) + \epsilon_t$ gets sampled. The noise is often modeled as an independent Gaussian distribution with zero mean and variance σ_n^2

$$\epsilon = \mathcal{N}(0, \sigma_n^2) \tag{3.15}$$

following the original proposition of Williams and Rasmussen [76]. Afterwards, the probabilistic surrogate model is fitted to the observations $\mathcal{D}_t$, realizing the update procedure from the algorithm. This surrogate model ideally represents the actual objective function as close as possible while also being mathematically advantageous and easy to compute [21]. A standard choice for that would be a Gaussian process (GP), although many other models can be used, e.g., Random Forests (RF) or Gradient Boosted Regression Trees (GBRT). An example of the process can be seen in Figure 3.7, where the three first iterations of BO are shown applied to a 1D toy function, using a GP surrogate with two initial random samples.

Different combinations of surrogate model and acquisition function are possible in realizing a BO algorithm. The relatively fast convergence to near-global optima makes this versatile algorithm a viable choice for HPO, heavily improving on model-free methods. However, the sequential reliance on previously sampled data makes parallelization difficult [79]. The following subsections will briefly explain some popular acquisition functions and then discuss three surrogate models in more detail.

Acquisition Functions

As mentioned before, an acquisition function guides the sampling process while balancing exploration and exploitation. Given the distribution of the surrogate model, including its predictive mean $\mu(\lambda)$, and its variance function $\sigma(\lambda)$, and the previously sampled data $\mathcal{D}$, with $f_{min} = \text{argmin}_{\lambda \in \mathcal{D}} f(\lambda)$ denoting the best observed value so far, the acquisition function is defined as $u : \mathbf{\Lambda} \rightarrow \mathbb{R}^+$ [67]. Two common choices for that function are the probability of improvement (PI) and the expected improvement (EI).

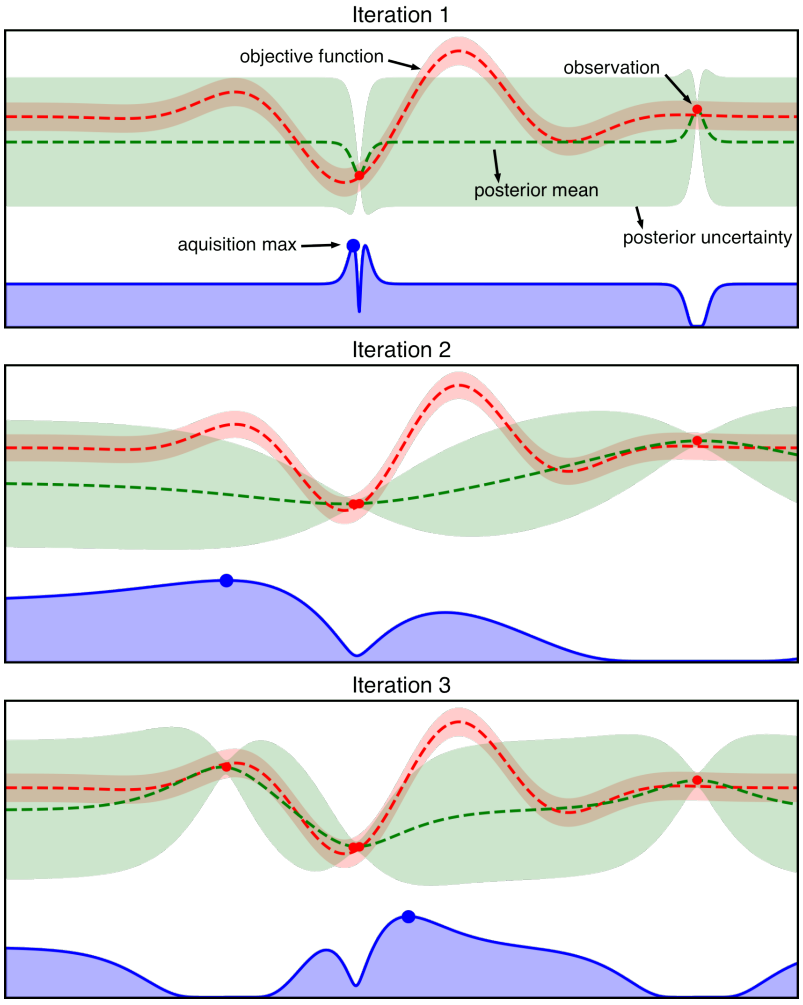


Figure 3.7: An example of BO using a GP surrogate (mean prediction as green dotted line, uncertainty as green tube) and an acquisition function (lower blue curve) on a noisy 1D toy function (red dotted line with red tube as noise). The figure shows three different iterations of the BO process using two initial samples: The top shows the first iteration with only the samples to build the surrogate on and low acquisition values around these points. The middle shows the following iteration, including the newly sampled point, while reducing the uncertainty. The bottom shows that only after three iterations, almost half of the acquisition space has lost its value, and the surrogate model gains more accuracy around the samples.

Probability of Improvement PI as proposed by Kushner [43] tries to maximize the probability of improving over the best current value f_{min}, with only small regard to the comparative amount of improvement in less certain, but more promising environments. With a trade-off parameter $\xi \geq 0$ to control this shortcoming, and $\Phi(\cdot)$ denoting the

cumulative distribution function (CDF) of the standard normal, the resulting acquisition function is:

$$\mathrm{PI}(\lambda) = P(f(\lambda) < f_{min} + \xi) = \Phi\left(\frac{f_{\min} - \mu(\lambda) - \xi}{\sigma(\lambda)}\right) \tag{3.16}$$

This process is greedy in nature, but offers guaranteed improvement of at least ξ [8].

Expected Improvement As proposed by Jones et al. [39], EI improves upon PI by also considering the magnitude of potential improvement a sample may yield. The goal is to calculate the expected deviation from a potential sample $f(\lambda)$ and the current minimum $f_{\min}$, so

$$\mathrm{EI}(\lambda) = \mathbb{E}[\max(f_{\min} - f(\lambda), 0)]. \tag{3.17}$$

This allows for choosing the sample giving a maximum improvement over the previous best observation. EI can also be expressed in a closed form with an additional trade-off parameter $\xi \geq 0$ for balancing exploration and exploitation. Using the standard normal CDF $\Phi(\cdot)$ and standard normal probability density function (PDF) $\phi(\cdot)$ the equation is defined as follows:

$$\mathrm{EI}(\lambda) = (f_{\min} - \mu(\lambda) - \xi) \cdot \Phi(Z) + \sigma(\lambda) \cdot \phi(Z) \tag{3.18}$$

$$Z = \frac{f_{\min} - \mu(\lambda) - \xi}{\sigma(\lambda)}$$

The first addend of Equation (3.18) realizes the exploitation of the function, with high means being preferred, while the second addend handles the exploration, choosing points with large surrogate variances [8].

Surrogate Model: Gaussian Process

A standard choice for a surrogate model is the GP. As an extension of the multivariate Gaussian distribution, a GP is defined by any finite number N of variables (parameters) λ, or, in this case, by its mean $m(\lambda)$ and a covariance function $k(\lambda, \lambda')$:

$$f(\lambda) \sim \mathcal{GP}(m(\lambda), k(\lambda, \lambda')) \tag{3.19}$$

with the mean function most often taken to be zero, therefore, making the process solely rely on the covariance function, with its specification assuming a distribution over the objective function [76]. The default choice for this so-called *kernel* (implying the usage of

the *kernel trick*) in the original proposition is a squared exponential function specifying the covariance between pairs of random variables λ_i, λ_j as:

$$k(\lambda_i, \lambda_j) = \exp\left(-\frac{1}{2}\|\lambda_i - \lambda_j\|^2\right) \tag{3.20}$$

Then, using the Sherman-Morrison-Woodbury formula, a predictive distribution for the function value at next iteration $t + 1$ can be expressed by:

$$P(f_{t+1}|\mathcal{D}_t, \lambda_{t+1}) = \mathcal{N}(\mu_t(\lambda_t + \mathbf{1}), \sigma_t^2(\lambda_t + \mathbf{1})) \tag{3.21}$$

where μ and σ^2 denote the mean and variance of the model, and can be calculated using the kernel matrix over the covariance function (see [8, p.8]). A disadvantage of the squared exponential kernel is that it assumes a very smooth objective function, which is unrealistic considering real physical processes [70]. Instead, a Matérn kernel is often proposed as a substitution. It uses a parameter υ to control smoothness and common settings for ML applications are $\upsilon = 3/2$ and $\upsilon = 5/2$ [76].

The standard GP scales cubically with the number of data points, limiting the amount of function evaluations. Another issue is poor scalability to higher dimensions, which is tried to be overcome by using other kernels [21].

Surrogate Model: Random Forest

Very different to GP are the next two approaches, coming from the realm of ensemble methods often used in ML. The basic idea of these methods is to train multiple so-called "base learners" $h_1(\lambda), ..., h_J(\lambda)$ (sometimes also referred to as "weak learners"), which are easy to fit and infer on, but have poor individual generalization performance due to very high variance. The base learners are then combined to give a final predictor of the objective function $\hat{f}(\lambda)(\approx f(\lambda))$. In case of regression, which is applicable for HPO, this means a simple average of all base learners [11]:

$$\hat{f}(\lambda) = \frac{1}{J}\sum_{j=1}^{J} h_j(\lambda) \tag{3.22}$$

RF is one of these methods, and was first introduced under that name by Breiman [7] as an improvement upon his bagging algorithm for decision trees. Each base learner, as described above, is a binary partitioned regression tree, with each partition or "split"

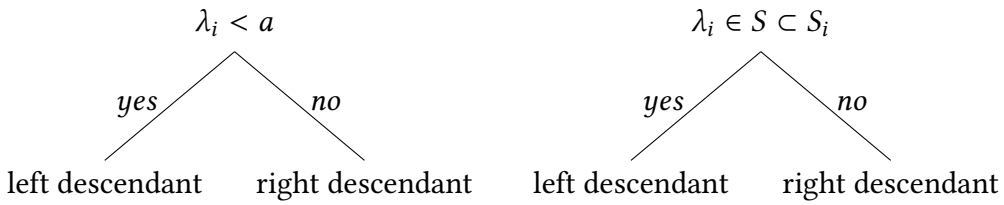


Figure 3.8: Example of two decision trees. (Left) Splitting a continuous variable λ_i at point a. (Right) Splitting a categorical variable λ_i via subset S.

being based on a predictor variable $\lambda_i \in \lambda$ (e.i., the hyperparameters). A split may be done by value range for a continuous variable or by choosing a subset S from the set of all categories S_i for categorical variables (see Figure 3.8). A splitting criterion is used to evaluate all possible splits among all variables, and then the most descriptive split is chosen. For regression, the mean squared error is often used for that task, whereas classification typically uses the Gini Index to calculate the "purity" of each potential class [11].

The bagging algorithm is now improving upon the foundation of decision trees [6]. Given a data set $\mathcal{D}$ of size n, consisting of pairs of N parameters $\lambda = \{\lambda_1, ..., \lambda_N\}$ and their corresponding function value $y = f(\lambda)$, bagging repeatedly chooses a random subset of the same size n as the original data set, but with replacement, therefore allowing for duplicates and possible unused values or "out-of-bag" data. This process is called "bootstrap sampling" and reduces overfitting, while also allowing the out-of-bag data to be used for validation of the estimator. The tree is then fitted using that sample as described above. RF introduces more randomization into the decision tree building process, by not only taking a subset of the available data for each base learner, but also taking a random sample (with replacement) of size m, with $m < N$ of the predictor values for each split. As a result, the base learners do not overfit on presumably highly predictive variables, which reduces correlation among the sub-sampled data sets and increases accuracy for the ensemble prediction [32]. The resulting implementation template is described in Algorithm 3.8. The process can also easily be parallelized, since each base learner is constructed individually [11].

Extremely Randomized Trees As the name suggests, the extremely randomized trees approach proposed by Geurts et al. [26] introduces another step of randomization into the process. While the main part of this algorithm, often called Extra-Trees (ET), is based on random forests, the split points for each node of the decision tree are now chosen

Algorithm 3.8 Random Forests

for $j = 1$ to J **do**
  $\mathcal{D}_j \subseteq \mathcal{D}$ // sample with replacement, $|\mathcal{D}_j| = |\mathcal{D}|$
  procedure CreateTree($\mathcal{D}_j, m$) $\rightarrow h_j(\lambda)$
    Start with all observations in root node
    for all unsplit nodes, recursivley **do**
      Randomly select m predictor variables
      Split the node according to the best binary split among these m variables
return $\hat{f}(\lambda)$ // using Equation (3.22)

completely randomly, as opposed to being based on the best split among all available predictive variables. To further explain, the sampled m predictor variables are each randomly split once, evaluated (e.g., using the mean squared error) and then the best performing split is chosen for the current node. In addition, each base learner is now built using the entire data set, rather than just a bootstrap sample. The motivation behind ET is to reduce variance through random splits and minimize bias by using the full data set, while also having the potential to improve the computational time needed to build the estimator.

Surrogate Model: Gradient Boosted Trees

GBRT is also an ensemble method using decision trees as base learners to create an ensemble prediction, and was originally proposed by Friedman [22]. However, in contrast to RF, where many full-depth decision trees are averaged, with GBRT, many small, high-bias decision trees (depth $d \approx 4$) are built sequentially, improving upon each other by using the residuals from the last iteration. The schematic architecture of this approach is outlined in Figure 3.9. For simplification, let $f_j = f_j(\lambda)$. The ensemble regressor $\hat{f}_j$ is the j-th of all J estimators in an additive sequence

$$\hat{f}_j = \hat{f}_{j-1} + v \cdot h_j \tag{3.23}$$

where $v \in (0, 1]$ is a "shrinkage" parameter controlling the learning rate leading to better generalization [23]. Let $\mathcal{L}(f, \hat{f})$ be the loss function for the estimator. Now, in each iteration j a new base learner h_j is added to the ensemble by minimizing over its sum of

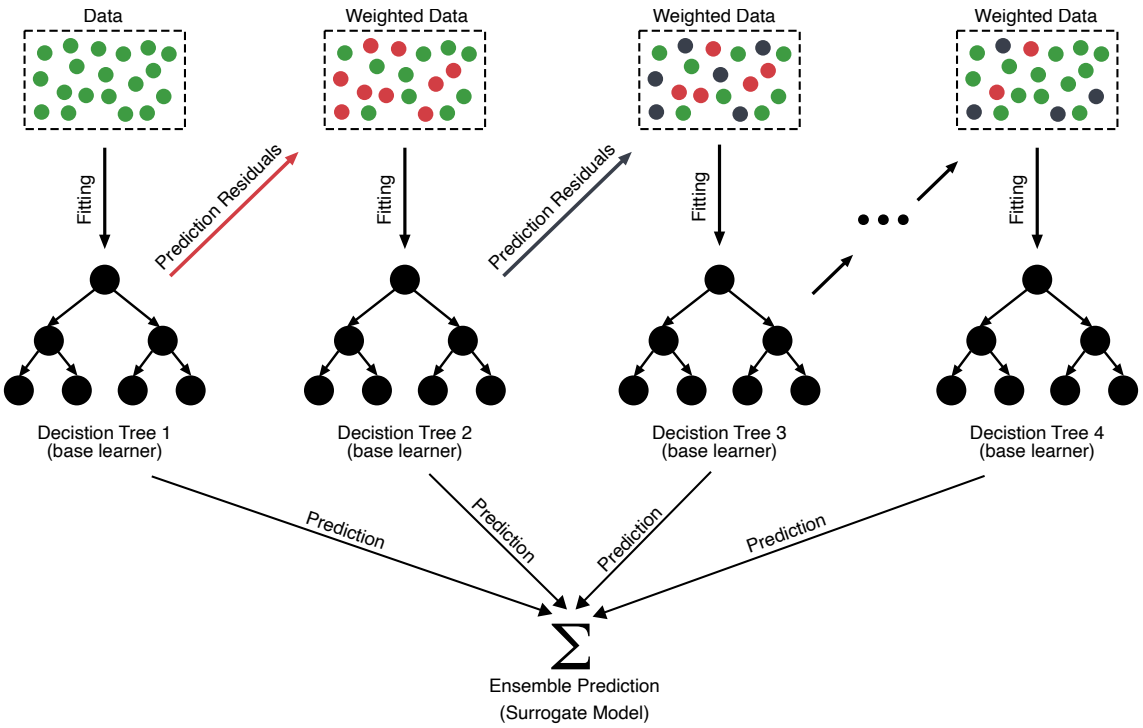


Figure 3.9: Schematic view of GBRT. Modified from Deng et al. [12].

losses for the whole data set $\mathcal{D}$ ($|\mathcal{D}| = n$), with (λ_i, y_i) being the i-th element of $\mathcal{D}$[3] and $y_i = f(\lambda_i)$ [22]:

$$h_j = \underset{h}{\operatorname{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, \hat{f}_{j-1}(\lambda_i) + h(\lambda_i)) \tag{3.24}$$

In order to bring this into computationally closed-form, a first-order Taylor approximation is used on the loss-function to get the following term:

$$\mathcal{L}(y_i, \hat{f}_{j-1}(\lambda_i) + h(\lambda_i)) \approx \mathcal{L}(y_i, \hat{f}_{j-1}(\lambda_i)) + h_j(\lambda_i) \left[\frac{\partial \mathcal{L}(y_i, \hat{f}(\lambda_i))}{\partial \hat{f}(\lambda_i)} \right]_{\hat{f}=\hat{f}_{j-1}} \tag{3.25}$$

The derivative of this equation can be understood as gradient g_{ij} to the loss function, where a steepest descent following $-g_{ij}$ is desired. To summarize, in each of the J iterations, a decision tree of fixed depth d, using all predictor variables, is fitted to minimize the negative gradient of the all n data points:

$$h_j \approx -\underset{h}{\operatorname{argmin}} \sum_{i=1}^{n} (h(\lambda_i) - g_{ij})^2 \tag{3.26}$$

[3] Please note, that this is an exception to the previously established notation of λ_i being the i-th parameter in the configuration.

Given a typical squared error loss function $L(f, \hat{f}) = \frac{1}{2}(f - \hat{f})^2$, the negative gradient can be simplified to an ordinary residual $r_{ij} = -g_{ij} = y_i - \hat{f}_{j-1}(\lambda_i)$ [31, Chapter 10].

Lastly, Algorithm 3.9 describes the high-level implementation of GBRT using a squared loss function and a decision tree building process, as described with RF in Section 3.3.4, using a continuous update for the residuals ($r_i \leftarrow r_{ij}$, for current iteration j) [53]. Although this algorithm uses a constant initialization for the residuals, other methods could be used as well.

Algorithm 3.9 Gradient Boosted Regression Trees (Squared Loss)

Initialization: $\forall i \in [1, n] : r_i = y_i$
for $j = 1$ to J **do**
  $h_j \leftarrow$ CreateTree($\{(\lambda_\mathbf{1}, r_1), ..., (\lambda_\mathbf{n}, r_n)\}, d$)
  for $i = 1$ to n **do**
    $r_i \leftarrow r_i - v \cdot h_j(\lambda_\mathbf{i})$
$\hat{f} = v \cdot \sum_{j=1}^{J} h_j$
return $\hat{f}$

4 Implementation

4.1 Modules

4.2 Workflow

4.3 Modes of Operation

5 Experimental Design and Tests

5.1 Choice of Problem Instances

5.1.1 Statistical Measures for Analysis

5.1.2 Classification

Method I - Value Range

Method II - Value Range

Method III - K-Means Clustering

K-Means Cluster Analysis

5.2 Choice of Methods for Optimization

5.2.1 Optimizer Initialization

5.3 Choice of Parameters and Value Ranges

5.4 Testing Procedure

5.5 Analyzing Procedure

5.5.1 Statistical Tests

Kruskal–Wallis H Test

Conover–Iman Test

6 Results and Evaluation

6.1 Part I - Choosing the Optimization Algorithm

6.1.1 Convergence Behavior

6.1.2 Statistical Tests

6.2 Part II - Choosing the Parameter Sets

6.2.1 Robustness of Parameter Values

6.2.2 Correspondence with Problem Instances

6.2.3 Parameter Importance

6.3 Part III - Evaluating the Parameter Sets

7 Conclusion and Outlook

Outlook

Abstract

<Short summary of the thesis>

Bibliography

[1] D. Angus, T. Hendtlass. "Ant colony optimisation applied to a dynamically changing problem". In: *Developments in Applied Artificial Intelligence: 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE 2002 Cairns, Australia, June 17–20, 2002 Proceedings 15*. Springer. 2002, pp. 618–627 (cit. on p. 9).

[2] J. Bergstra, Y. Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012) (cit. on p. 35).

[3] M. Birattari, J. Kacprzyk. *Tuning Metaheuristics: A Machine Learning Perspective*. Vol. 197. Springer, 2009 (cit. on p. 11).

[4] C. Blum, A. Roli. "Metaheuristics in combinatorial optimization: Overview and conceptual comparison". In: *ACM computing surveys (CSUR)* 35.3 (2003), pp. 268–308 (cit. on pp. 17, 18).

[5] I. Boussaid, J. Lepagnot, P. Siarry. "A survey on optimization metaheuristics". In: *Information sciences* 237 (2013), pp. 82–117 (cit. on p. 16).

[6] L. Breiman. "Bagging predictors". In: *Machine learning* 24 (1996), pp. 123–140 (cit. on p. 41).

[7] L. Breiman. "Random forests". In: *Machine learning* 45 (2001), pp. 5–32 (cit. on p. 40).

[8] E. Brochu, V. M. Cora, N. De Freitas. "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning". In: *arXiv preprint arXiv:1012.2599* (2010) (cit. on pp. 36, 37, 39, 40).

[9] R. E. Burkard, S. E. Karisch, F. Rendl. "QAPLIB–a quadratic assignment problem library". In: *Journal of Global optimization* 10 (1997), pp. 391–403 (cit. on p. 5).

[10] G. A. Croes. "A method for solving traveling-salesman problems". In: *Operations research* 6.6 (1958), pp. 791–812 (cit. on p. 9).

[11] A. Cutler, D. R. Cutler, J. R. Stevens. “Random forests”. In: *Ensemble machine learning: Methods and applications* (2012), pp. 157–175 (cit. on pp. 40, 41).

[12] H. Deng, Y. Zhou, L. Wang, C. Zhang. “Ensemble learning for the early prediction of neonatal jaundice with genetic features”. In: *BMC medical informatics and decision making* 21 (2021), pp. 1–11 (cit. on p. 43).

[13] F. Dobslaw. “A parameter tuning framework for metaheuristics based on design of experiments and artificial neural networks”. In: *International conference on computer mathematics and natural computing*. WASET. 2010 (cit. on p. 11).

[14] M. Dorigo, A. Colorni, V. Maniezzo. “Ant system: An autocatalytic optimizing process”. In: *Dipartimento Di Elettronica, Politecnico Di Milano, Milan, Italy* (1991) (cit. on pp. 10, 13).

[15] M. Dorigo, V. Maniezzo, A. Colorni. “Ant system: optimization by a colony of cooperating agents”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1 (1996), pp. 29–41 (cit. on pp. 10, 19, 21).

[16] M. Dorigo, T. Stützle. *Ant Colony Optimization*. The MIT Press, June 2004. DOI: 10.7551/mitpress/1290.001.0001 (cit. on pp. 10, 19).

[17] M. Dorigo, T. Stützle. *Ant colony optimization: overview and recent advances*. Springer, 2019 (cit. on p. 20).

[18] Á. E. Eiben, R. Hinterding, Z. Michalewicz. “Parameter control in evolutionary algorithms”. In: *IEEE Transactions on evolutionary computation* 3.2 (1999), pp. 124–141 (cit. on pp. 10, 31).

[19] L. J. Eshelman. “The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination”. In: *Foundations of genetic algorithms*. Vol. 1. Elsevier, 1991, pp. 265–283 (cit. on p. 9).

[20] C. J. Eyckelhof, M. Snoek. “Ant systems for a dynamic TSP: Ants caught in a traffic jam”. In: *Ant Algorithms: Third International Workshop, ANTS 2002 Brussels, Belgium, September 12–14, 2002 Proceedings*. Springer. 2002, pp. 88–99 (cit. on p. 9).

[21] M. Feurer, F. Hutter. “Hyperparameter optimization”. In: *Automated machine learning: Methods, systems, challenges* (2019), pp. 3–33 (cit. on pp. 7, 33–35, 37, 40).

[22] J. H. Friedman. “Greedy function approximation: a gradient boosting machine”. In: *Annals of statistics* (2001), pp. 1189–1232 (cit. on pp. 42, 43).

[23] J. H. Friedman. “Stochastic gradient boosting”. In: *Computational statistics & data analysis* 38.4 (2002), pp. 367–378 (cit. on p. 42).

[24] D. Gaertner, K. L. Clark. "On Optimal Parameters for Ant Colony Optimization Algorithms." In: *IC-AI*. Citeseer. 2005, pp. 83–89 (cit. on p. 10).

[25] M. Gendreau, J.-Y. Potvin, et al. *Handbook of metaheuristics*. Vol. 2. Springer, 2010 (cit. on p. 16).

[26] P. Geurts, D. Ernst, L. Wehenkel. "Extremely randomized trees". In: *Machine learning* 63 (2006), pp. 3–42 (cit. on p. 41).

[27] S. Goss, S. Aron, J.-L. Deneubourg, J. M. Pasteels. "Self-organized shortcuts in the Argentine ant". In: *Naturwissenschaften* 76.12 (1989), pp. 579–581 (cit. on p. 19).

[28] M. Guntsch, M. Middendorf. "A population based approach for ACO". In: *Applications of Evolutionary Computing: EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN Kinsale, Ireland, April 3–4, 2002 Proceedings*. Springer. 2002, pp. 72–81 (cit. on pp. 21, 22).

[29] M. Guntsch, M. Middendorf. "Pheromone modification strategies for ant algorithms applied to dynamic TSP". In: *Applications of Evolutionary Computing: EvoWorkshops 2001: EvoCOP, EvoFlight, EvoIASP, EvoLearn, and EvoSTIM Como, Italy, April 18–20, 2001 Proceedings*. Springer. 2001, pp. 213–222 (cit. on p. 9).

[30] Z.-F. Hao, R.-C. Cai, H. Huang. "An adaptive parameter control strategy for ACO". In: *2006 International Conference on Machine Learning and Cybernetics*. IEEE. 2006, pp. 203–206 (cit. on p. 10).

[31] T. Hastie, R. Tibshirani, J. H. Friedman, J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer, 2009 (cit. on p. 44).

[32] T. K. Ho. "A data complexity analysis of comparative advantages of decision forest constructors". In: *Pattern Analysis & Applications* 5 (2002), pp. 102–112 (cit. on p. 41).

[33] Z.-C. Huang, X.-L. Hu, S.-D. Chen. "Dynamic traveling salesman problem based on evolutionary computation". In: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*. Vol. 2. IEEE. 2001, pp. 1283–1288 (cit. on p. 9).

[34] R. S. Jamisola Jr, E. P. Dadiost, M. H. Ang Jr. "Using Metaheuristic Computations to Find the Minimum-Norm-Residual Solution to Lrnear Systerns of Equations". In: *Philippine Computing Journal* 4.2 (2009), pp. 1–9 (cit. on p. 5).

[35] S. Janson, M. Middendorf. "A hierarchical particle swarm optimizer". In: *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.* Vol. 2. IEEE. 2003, pp. 770–776 (cit. on pp. 6, 23–25).

[36] S. Janson, M. Middendorf. "A hierarchical particle swarm optimizer for dynamic optimization problems". In: *Applications of Evolutionary Computing: EvoWorkshops 2004: EvoBIO, EvoCOMNET, EvoHOT, EvoISAP, EvoMUSART, and EvoSTOC, Coimbra, Portugal, April 5-7, 2004. Proceedings*. Springer. 2004, pp. 513–524 (cit. on pp. 6, 9).

[37] S. Janson, M. Middendorf. "A hierarchical particle swarm optimizer for noisy and dynamic environments". In: *Genetic programming and evolvable machines* 7 (2006), pp. 329–354 (cit. on p. 9).

[38] D. S. Johnson, G. Gutin, L. A. McGeoch, A. Yeo, W. Zhang, A. Zverovitch. "Experimental analysis of heuristics for the ATSP". In: *The traveling salesman problem and its variations* (2007), pp. 445–487 (cit. on p. 15).

[39] D. R. Jones, M. Schonlau, W. J. Welch. "Efficient global optimization of expensive black-box functions". In: *Journal of Global optimization* 13.4 (1998), p. 455 (cit. on p. 39).

[40] J.-R. Jung, B.-J. Yum. "Artificial neural network based approach for dynamic parameter design". In: *Expert Systems with Applications* 38.1 (2011), pp. 504–510 (cit. on p. 11).

[41] J. Kennedy, R. Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95-international conference on neural networks*. Vol. 4. IEEE. 1995, pp. 1942–1948 (cit. on p. 23).

[42] E. Kupfer, H. T. Le, J. Zitt, Y.-C. Lin, M. Middendorf. "A Hierarchical Simple Probabilistic Population-Based Algorithm Applied to the Dynamic TSP". In: *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2021, pp. 1–8 (cit. on pp. 6, 27, 30).

[43] H. J. Kushner. "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise". In: (1964) (cit. on p. 38).

[44] E. L. Lawler. "The traveling salesman problem: a guided tour of combinatorial optimization". In: *Wiley-Interscience Series in Discrete Mathematics* (1985) (cit. on p. 13).

[45] C. Li, M. Yang, L. Kang. "A new approach to solving dynamic traveling salesman problems". In: *Simulated Evolution and Learning: 6th International Conference, SEAL 2006, Hefei, China, October 15-18, 2006. Proceedings 6*. Springer. 2006, pp. 236–243 (cit. on p. 9).

[46] P. Li, H. Zhu. “Parameter selection for ant colony algorithm based on bacterial foraging algorithm”. In: *Mathematical Problems in Engineering* 2016 (2016) (cit. on p. 11).

[47] S. Lin. “Computer solutions of the traveling salesman problem”. In: *Bell System Technical Journal* 44.10 (1965), pp. 2245–2269 (cit. on p. 9).

[48] S. Lin, B. W. Kernighan. “An effective heuristic algorithm for the traveling-salesman problem”. In: *Operations research* 21.2 (1973), pp. 498–516 (cit. on pp. 9, 13).

[49] Y.-C. Lin, M. Clauss, M. Middendorf. “Simple probabilistic population-based optimization”. In: *IEEE Transactions on Evolutionary Computation* 20.2 (2015), pp. 245–262 (cit. on pp. 5, 6, 25–27).

[50] O. Maron, A. Moore. “Hoeffding races: Accelerating model selection search for classification and function approximation”. In: *Advances in neural information processing systems* 6 (1993) (cit. on p. 11).

[51] M. Mavrovouniotis, F. M. Müller, S. Yang. “Ant colony optimization with local search for dynamic traveling salesman problems”. In: *IEEE transactions on cybernetics* 47.7 (2016), pp. 1743–1756 (cit. on p. 9).

[52] M. Mavrovouniotis, S. Yang. “Ant colony optimization with immigrants schemes for the dynamic travelling salesman problem with traffic factors”. In: *Applied Soft Computing* 13.10 (2013), pp. 4023–4037 (cit. on p. 9).

[53] A. Mohan, Z. Chen, K. Weinberger. “Web-search ranking with initialized gradient boosted regression trees”. In: *Proceedings of the learning to rank challenge*. PMLR. 2011, pp. 77–89 (cit. on p. 44).

[54] H. Neyoy, O. Castillo, J. Soria. “Dynamic fuzzy logic parameter tuning for ACO and its application in TSP problems”. In: *Recent advances on hybrid intelligent systems* (2013), pp. 259–271 (cit. on p. 10).

[55] M. Packianather, P. Drake, H. Rowlands. “Optimizing the parameters of multilayered feedforward neural networks through Taguchi design of experiments”. In: *Quality and reliability engineering international* 16.6 (2000), pp. 461–473 (cit. on p. 11).

[56] C. H. Papadimitriou, K. Steiglitz. “Some complexity results for the traveling salesman problem”. In: *Proceedings of the eighth annual ACM symposium on theory of computing*. 1976, pp. 1–9 (cit. on p. 13).

[57] H. N. Psaraftis. “Dynamic vehicle routing problems”. In: *Vehicle routing: Methods and studies* 16 (1988), pp. 223–248 (cit. on p. 9).

[58] H. N. Psaraftis. “Dynamic vehicle routing: Status and prospects”. In: *Annals of operations research* 61.1 (1995), pp. 143–164 (cit. on p. 6).

[59] A. P. Punnen. “The traveling salesman problem: Applications, formulations and variations”. In: *The traveling salesman problem and its variations* (2007), pp. 1–28 (cit. on p. 13).

[60] M. Randall. “Near parameter free ant colony optimisation”. In: *Ant Colony Optimization and Swarm Intelligence: 4th International Workshop, ANTS 2004, Brussels, Belgium, September 5-8, 2004. Proceedings 4*. Springer. 2004, pp. 374–381 (cit. on p. 11).

[61] G. Reinelt. “TSPLIB—A traveling salesman problem library”. In: *ORSA journal on computing* 3.4 (1991), pp. 376–384 (cit. on pp. 5, 13).

[62] J. Robinson. *On the Hamiltonian game (a traveling salesman problem)*. Tech. rep. Rand project air force arlington va, 1949 (cit. on p. 13).

[63] A. Schrijver. “On the history of combinatorial optimization (till 1960)”. In: *Handbooks in operations research and management science* 12 (2005), pp. 1–68 (cit. on p. 13).

[64] V. Sharma, A. K. Tripathi. “A systematic review of meta-heuristic algorithms in IoT based application”. In: *Array* (2022), p. 100164 (cit. on p. 5).

[65] C. A. Silva, T. A. Runkler. “Ant colony optimization for dynamic traveling salesman problems”. In: *ARCS 2004–Organic and pervasive computing* (2004) (cit. on p. 9).

[66] A. Simoes, E. Costa. “CHC-based algorithms for the dynamic traveling salesman problem”. In: *Applications of Evolutionary Computation: EvoApplications 2011: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Torino, Italy, April 27-29, 2011, Proceedings, Part I*. Springer. 2011, pp. 354–363 (cit. on p. 9).

[67] J. Snoek, H. Larochelle, R. P. Adams. “Practical bayesian optimization of machine learning algorithms”. In: *Advances in neural information processing systems* 25 (2012) (cit. on p. 37).

[68] K. Sörensen, F. Glover. “Metaheuristics”. In: *Encyclopedia of operations research and management science* 62 (2013), pp. 960–970 (cit. on pp. 15, 16).

[69] K. Sörensen, M. Sevaux, F. Glover. “A history of metaheuristics”. In: *Handbook of heuristics*. Springer, 2018, pp. 791–808 (cit. on pp. 5, 16).

[70] M. L. Stein. *Interpolation of spatial data: some theory for kriging*. Springer Science & Business Media, 1999 (cit. on p. 40).

[71] T. Stützle, M. López-Ibánez, P. Pellegrini, M. Maur, M. Montes de Oca, M. Birattari, M. Dorigo. *Parameter adaptation in ant colony optimization*. Springer, 2012 (cit. on pp. 10, 31).

[72] E.-G. Talbi. *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009 (cit. on pp. 10, 17–20, 23, 31).

[73] A. Tortum, N. Yayla, C. Çelik, M. Gökdağ. "The investigation of model selection criteria in artificial neural networks by the Taguchi method". In: *Physica A: Statistical Mechanics and its Applications* 386.1 (2007), pp. 446–468 (cit. on p. 11).

[74] A. F. Tuani, E. Keedwell, M. Collett. "H-ACO: A heterogeneous ant colony optimisation approach with application to the travelling salesman problem". In: *Artificial Evolution: 13th International Conference, Évolution Artificielle, EA 2017, Paris, France, October 25–27, 2017, Revised Selected Papers 13*. Springer. 2018, pp. 144–161 (cit. on p. 10).

[75] U. Vogel. "A flexible metaheuristic framework for solving rich vehicle routing problems: Formulierung, Implementierung und Anwendung eines kognitionsbasierten Simulationsmodells". PhD thesis. Köln, Universität zu Köln, Diss., 2011, 2011 (cit. on p. 5).

[76] C. K. Williams, C. E. Rasmussen. *Gaussian processes for machine learning*. Vol. 2. 3. MIT press Cambridge, MA, 2006 (cit. on pp. 36, 37, 39, 40).

[77] D. H. Wolpert, W. G. Macready. "No free lunch theorems for optimization". In: *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 67–82 (cit. on p. 5).

[78] K. Y. Wong et al. "Parameter tuning for ant colony optimization: A review". In: *2008 International Conference on Computer and Communication Engineering*. IEEE. 2008, pp. 542–545 (cit. on p. 10).

[79] L. Yang, A. Shami. "On hyperparameter optimization of machine learning algorithms: Theory and practice". In: *Neurocomputing* 415 (2020), pp. 295–316 (cit. on pp. 5, 33, 35, 37).

[80] W.-C. Yeh. "A two-stage discrete particle swarm optimization for the problem of multiple multi-level redundancy allocation in series systems". In: *Expert Systems with Applications* 36.5 (2009), pp. 9192–9200 (cit. on p. 25).

[81] W.-C. Yeh. "Simplified swarm optimization in disassembly sequencing problems with learning effects". In: *Computers & Operations Research* 39.9 (2012), pp. 2168–2177 (cit. on p. 25).

[82] E. Yin, K. Wijk. *Bayesian Parameter Tuning of the Ant Colony Optimization Algorithm: Applied to the Asymmetric Traveling Salesman Problem.* 2021 (cit. on p. 11).

Acronyms

ACO Ant Colony Optimization. 9, 19

ANN artificial neural network. 11

BO Bayesian Optimization. 11, 34

CDF cumulative distribution function. 39

DoE Design of Experiment. 11, 32

DTSP Dynamic Traveling Salesperson Problem. 6, 15

EA Evolutionary Algorithm. 18

EI expected improvement. 37

ET Extra-Trees. 41

GA Genetic Algorithm. 5

GBRT Gradient Boosted Regression Trees. 37

GP Gaussian process. 37

GS Grid Search. 34

HPO Hyperparameter Optimization. 7, 33

H-PSO Hierarchical Particle Swarm Pptimization. 6, 25

H-SPPBO Hierarchical Simple Probabilistic Population-Based Optimization. 6, 27

ML machine learning. 5, 32

PDF probability density function. 39

PI probability of improvement. 37

P-metaheuristic Population-Based Metaheuristic. 18

PACO Population-Based Ant Colony Optimization. 6, 21

PSO Particle Swarm Optimization. 5, 23

QAP Quadratic Assignment Problem. 5, 28

RF Random Forests. 37

RS Random Search. 35

SA Simulated Annealing. 17

S-metaheuristic Single-Solution Based Metaheuristic. 17

SCE Solution Creating Entity. 6, 26

SPPBO Simple Probabilistic Population-Based Optimization. 5, 26

SSO Simplified Swarm Optimization. 6, 25

TS Tabu Search. 18

TSP Traveling Salesperson Problem. 5, 13

List of Figures

List of Tables

List of Listings

List of Algorithms