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
April 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 [98], to routing the traffic from "Internet of Things" (IoT) devices [84], 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 [50], 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* [103]. 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" [101] 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 optimization problem.

As a result, this increasingly growing scientific field is not only becoming more convoluted [89], 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 [65], 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 [12, 81]), 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 [65]. 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) [58], an algorithm derived using the SPPBO framework and incorporating aspects from Hierarchical Particle Swarm Optimization (H-PSO) [51]. 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 [52].

This opens up the scope of this thesis to dynamically changing problems, like the Dynamic Traveling Salesperson Problem (DTSP) [78]. 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 [32]. 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 [77] in 1988, mentioning “interchange methods”, like the 2-opt [19], 3-opt [63] or Lin-Kernighan [64] 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. [49]. Following this, the work of Angus and Hendtlass [2] 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 [40] 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 [31]. Unlike changing the node topology of the TSP, Silva and Runkler [85] 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 [70], [69].

Examining other metaheuristic categories, Li et al. [61] 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 [86], using an algorithm based on CHC (“Cross-generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation” [30]) 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 [52], [53]. 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. [29] 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 [93] did a similar classification, including further distinction between (offline) parameter tuning methods. Lastly, In their review of parameter optimization, Wong et al. [102] 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. [24, 25]. 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 [26]. These parameter combinations were then challenged by Gaertner and Clark [35], 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. [96], 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 [92]. 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. [73], where fuzzy logic statements are used to improve the solution diversification behavior. A self-adaptive control scheme was proposed by Hao et al. [42], 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 [62], 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 [80], 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 [5]. 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 [68] (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 [23], 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 [56, 74, 95].

Lastly, a similar approach to this thesis, albeit more narrow in scope, was researched by Yin and Wijk [106]. 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 [83]. 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 [60]. 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 [64]. At the same time, the TSP was found to be NP-complete and therefore, NP-hard [75] - a problem category that still remains a very interesting research topic in computer science. Through the work by Dorigo et al. [24] 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 [81] 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 [79].

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 $E = \left\{(v_i, v_j) | v_i, v_j \in V, i \neq j\right\}$. Staying in the traveling salesperson analogy, the vertices are often being referred to as "cities", while the edges between vertices represent the

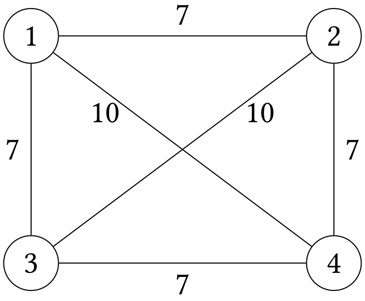


Figure 3.1: Example of a symmetric TSP instance with $n = 4$ cities

distance. This can be expressed by a distance function $d : V^2 \longrightarrow \mathbb{R}_0^+$, 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:

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

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 complexity and optimization potential for these instances [54]. 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 optimal 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 [88]. 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 [8]. 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" [88] 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 [36]. 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 [89].

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 [88]
- ...which analogy they belong to:
  Bio-Inspired vs. Physics-Based vs. Evolutionary vs. Swarm-Based

Although all of these classifications are justified, none of them will be used on its own, as it would serve no purpose to limit the discussion to one category. When necessary, algorithms are placed within these taxonomies, to explain their purpose respectively.

3.2.1 Overview

The first classification mentioned (Single-Solution Based vs. Population-Based) is 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 [93]. 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

```
$s \leftarrow$ GenerateInitialSolution( )
repeat
    $s' \leftarrow$ Generate($N(s)$)
    if Select($s'$) then
        $s \leftarrow s' \in N(s)$
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 [93]. 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 [6].

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 [6].

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) [93].

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 [6]. 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 [93].

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. [25]) 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 [38]. 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 [25]. 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 [93]. 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 [26].

[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 [93].

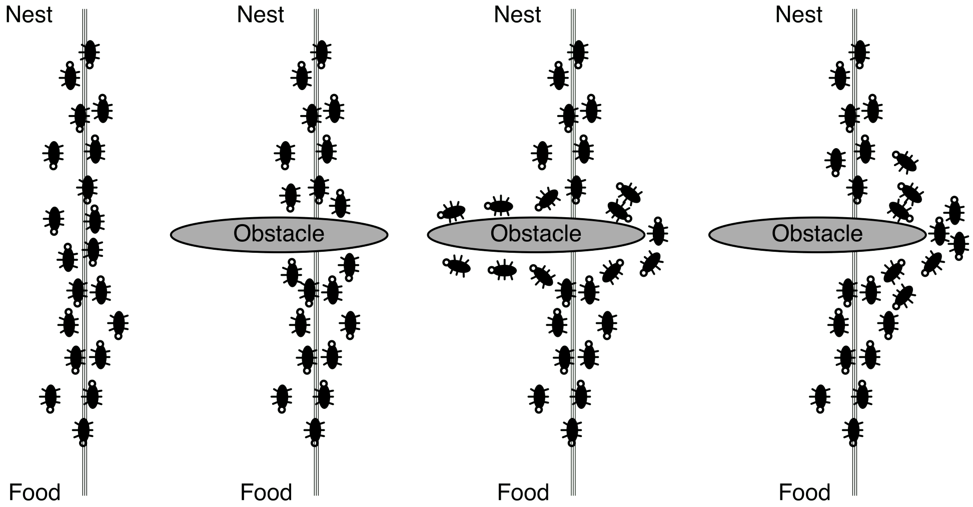


Figure 3.2: The process of ants following a pheromone trail between a food source and their nest affected by an obstacle. Modified from Talbi [93].

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 [27].

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 and 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 [93].

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:

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

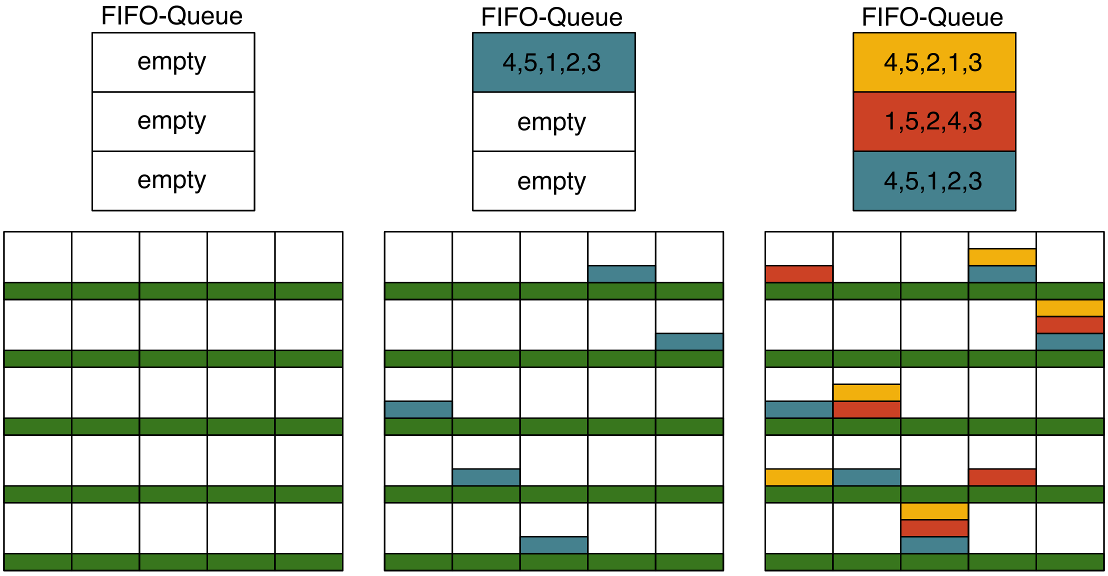
In the following reinforcement phase different strategies can be applied to select how the solutions selected by the ants influence the pheromone matrix. There are also versions of the ACO where this procedure is called after each step of the solution construction or at least after a single ant, but 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" [25], 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 $O(\mathrm{NC} \cdot n^2 \cdot m)$ [25].

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



(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).

Population-Based Ant Colony Optimization

The Population-Based Ant Colony Optimization (PACO) was first proposed by Guntsch and Middendorf [39] 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 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 [39].

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 [57] 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 $\vec{g_k}$ [93]. Putting both aspects together, the particles update their velocity based on personal success (*cognitive aspect*) and their neighborhoods success (*social aspect*) [51].

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
```

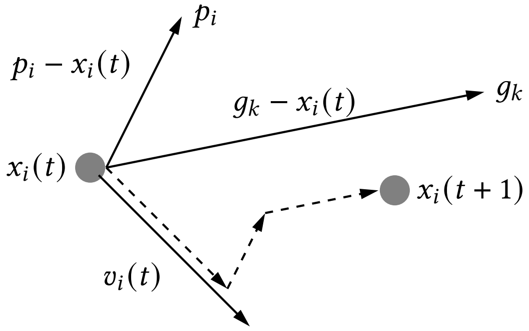


Figure 3.4: The vector summation of a PSO particle i during velocity and position update.

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 [51].

One such variant is the Hierarchical Particle Swarm Optimization (H-PSO) proposed by Janson and Middendorf [51], 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 [104]. 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 [105]. 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 [65].

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. [65], 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 $A \in \mathcal{A}$ do
        CREATE($A$)
    for all population $P \in \mathcal{P}$ 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 [65].

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. [58], 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 $\text{parent}(A)$, and a "root SCE" $A*$ defined as its own parent ($\text{parent}(A*) = A*$). 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 $P^{A}_{\text{parentbest}}$

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 is not discussed in this thesis.

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

[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.

Algorithm 3.5 H-SPPBO

```
Initialize the SCE tree
Initialize SCEs with random populations $P^A_{\text{persprev}}$,$P^A_{\text{persbest}}$
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 ← swapNum + 1
    if swapNum > $[\theta \cdot |\mathcal{A}|]$ and no change in $L$ previous iterations then
        ChangeHandlingProcedure($H$)
until termination criterion met
```

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_{persprev}, P^A_{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 is randomly drawn from that distribution, added to the solution vector $\mathbf{s}$ and removed from the unvisited set $U \longleftarrow j \setminus U$. 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 $f : V^n \longrightarrow \mathbb{R}_0^+$, 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 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).

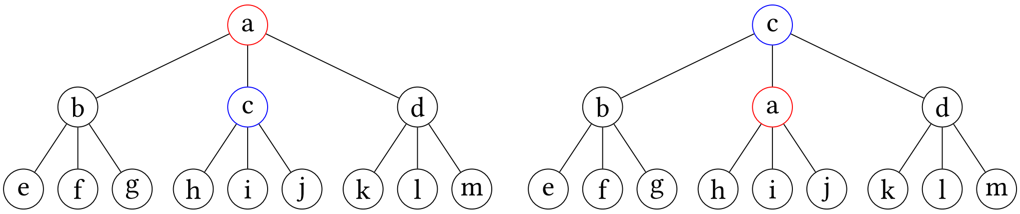


Figure 3.5: Example of a ternary SCE tree ($m = 3$) with height $h = 3$ showing the SWAP(A,C) operation before (left) and after (right) the execution. Modified from Kupfer et al. [58].

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 [58].

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.

This thesis applies methods from machine learning research to address said challenge. Therefore, the following review of parameter optimization methods classifies this technique and identifies other possible strategies. Figure 3.6 shows a taxonomy of parameter optimization methods by Eiben et al. [29], with additions from Talbi [93] and Stützle et al. [92]. 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.

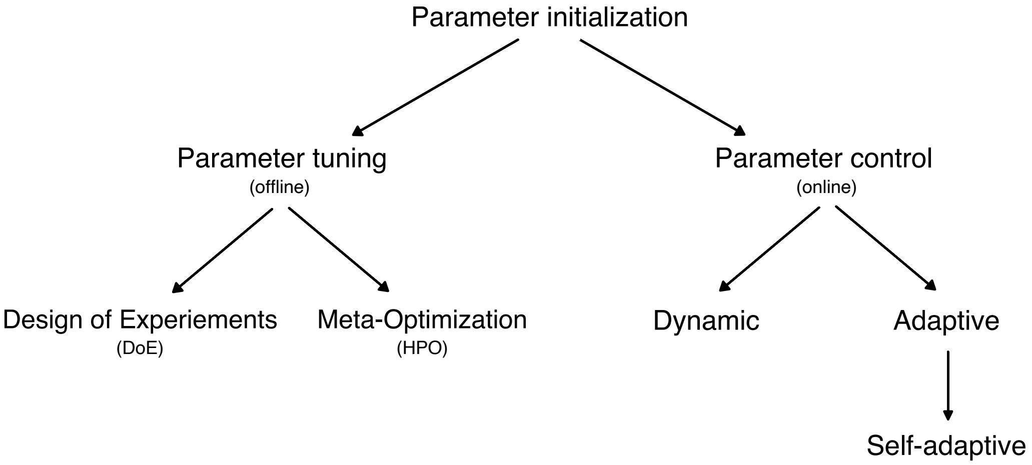


Figure 3.6: Taxonomy of parameter optimization. Modified from Eiben et al. [29].

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 $D = \{x \in X | g_i(x) \leq 0, h_j(x) = 0\}$, 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 [103]. 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 [32].

The HPO problem statement can be formulated as follows [32]: 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 $\mathbf{\Lambda} = \Lambda_1 \times \Lambda_2 \times \ldots \times \Lambda_N$, 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 $f : \mathbf{\Lambda} \longrightarrow \mathbb{R}$ 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 [32]. 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. Other strategies for HPO involve evolutionary algorithms or population-based methods, such as PSO [103]. Although, this approach of using metaheuristics to tune a metaheuristic is similar to the process proposed by [93] (see Section 3.3.1), it is not used in later stages of the thesis, due to its conflicting nature in the context of parameter optimization applied to a complex metaheuristic such as H-SPPBO.

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 [4]. 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 [103].

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 [4]. 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) [4]. 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 [103]. Nevertheless, it still serves as a well-performing baseline for many ML benchmarks, with hyperparameters relatively close to the optimum, given sufficient resources [32].

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 $\lambda^*$
```

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 $\mathcal{D}_t = \{(\lambda_{\text{i}}, f(\lambda_{\text{i}}))|i \in [1, t]\}$ 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 [11]. 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 [100].

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 [11]. With the most promising parameter input λ_{t} of iteration t specified, the (potentially) noisy objective function $f(\lambda_{\text{t}}) + \epsilon_t$ gets sampled. The noise

Algorithm 3.7 Bayesian Optimization

```
Initialize parameters: $\lambda^* \longleftarrow F(\mathbf{\Lambda})$
$y_{min} = f(\lambda^*) + \epsilon$
InitializeModel($\mathcal{D}_{\text{init}}$)
for $t \in [1, \text{n\_calls}]$ do
    $\lambda_{\text{t}} = \text{argmax}_{\lambda}\, u(\lambda|\mathcal{D}_{t-1})$                // acquisition function
    $y_t = f(\lambda_{\text{t}}) + \epsilon_t$
    $\mathcal{D}_t = \{\mathcal{D}_{t-1}, (\lambda_{\text{t}}, f_t)\}$
    UpdateModel($\mathcal{D}_t$)
    if $y_t < y_{min}$ then
        $y_{min} = y_t$
        $\lambda^* = \lambda_{\text{t}}$
return $\lambda^*$
```

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 [100]. 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 [32]. 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 [103]. The following subsections briefly explains some popular acquisition functions and then discusses three surrogate models in more detail. These models were selected based on their variance with respect to the BO process. A more detailed justification is given in Section 5.2.

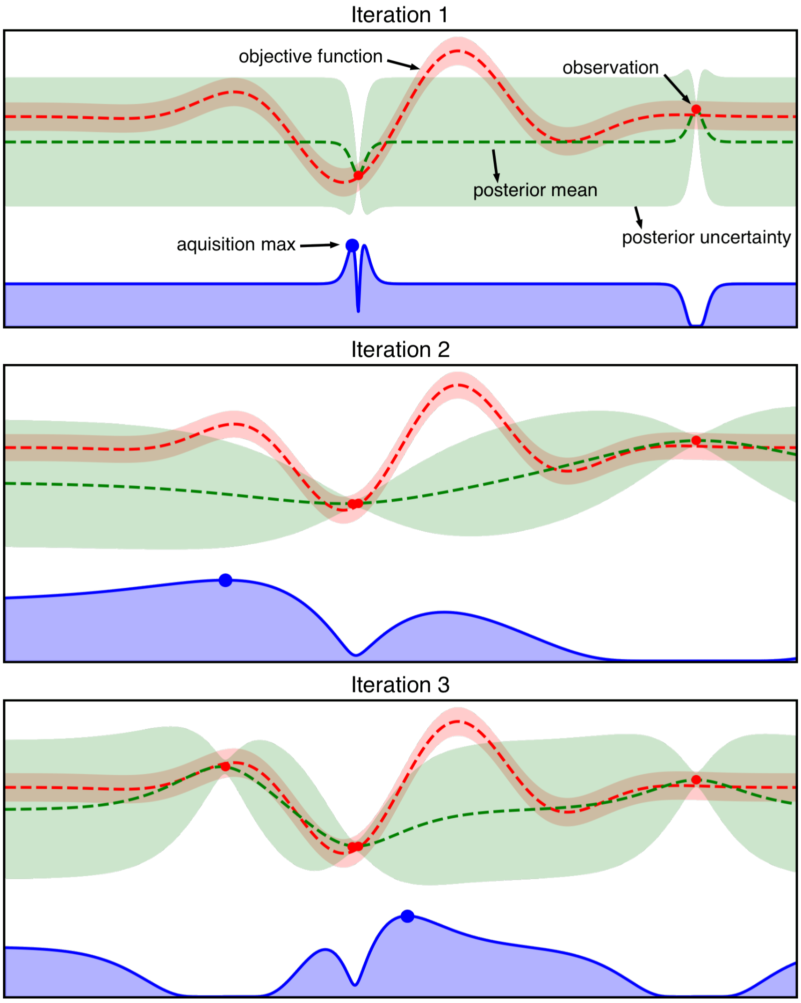


Figure 3.7: An example of BO using a GP surrogate (mean prediction as green dotted line, uncertainty as green tube) and an acquisition function (lower blue curve) on a noisy 1D toy function (red dotted line with red tube as noise). The figure shows three different iterations of the BO process using two 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.

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 $u : \mathbf{\Lambda} \rightarrow \mathbb{R}^+$ [87]. Two common choices for that function are the probability of improvement (PI) and the expected improvement (EI).

Probability of Improvement PI as proposed by Kushner [59] 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:

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

This process is greedy in nature, but offers guaranteed improvement of at least ξ [11].

Expected Improvement As proposed by Jones et al. [55], 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

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

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:

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

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

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 [11].

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 [100]. 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 $\lambda_\mathrm{i}, \lambda_\mathrm{j}$ as:

$$k(\lambda_\mathrm{i}, \lambda_\mathrm{j}) = \exp\left(-\frac{1}{2}||\lambda_\mathrm{i} - \lambda_\mathrm{j}||^2\right) \tag{3.20}$$

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

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

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 [11, 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 [90]. Instead, a Matérn kernel is often proposed as a substitution. It uses a parameter v to control smoothness and common settings for ML applications are $v = 3/2$ and $v = 5/2$ [100].

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 [32].

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 [20]:

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

RF is one of these methods, and was first introduced under that name by Breiman [10] 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"

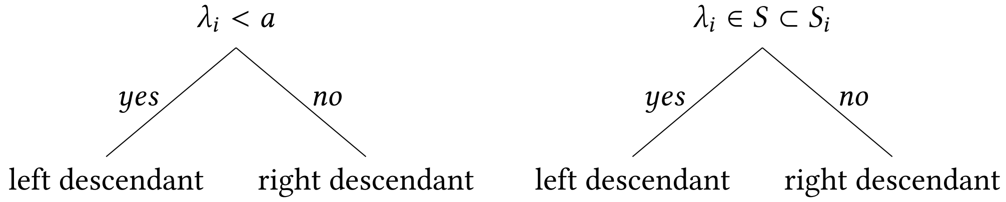


Figure 3.8: Example of two decision trees. (Left) Splitting a continuous variable λ_i at point a. (Right) Splitting a categorical variable λ_i via subset S.

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 [20].

The bagging algorithm is now improving upon the foundation of decision trees [9]. 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 [47]. The resulting implementation template is described in Algorithm 3.8. The process can also easily be parallelized, since each base learner is constructed individually [20].

Extremely Randomized Trees As the name suggests, the extremely randomized trees approach proposed by Geurts et al. [37] 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 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

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)
```

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 [33]. 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

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

where $\upsilon \in (0, 1]$ is a “shrinkage” parameter controlling the learning rate leading to better generalization [34]. 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 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 $y_i = f(\lambda_i)$ [33]:

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

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

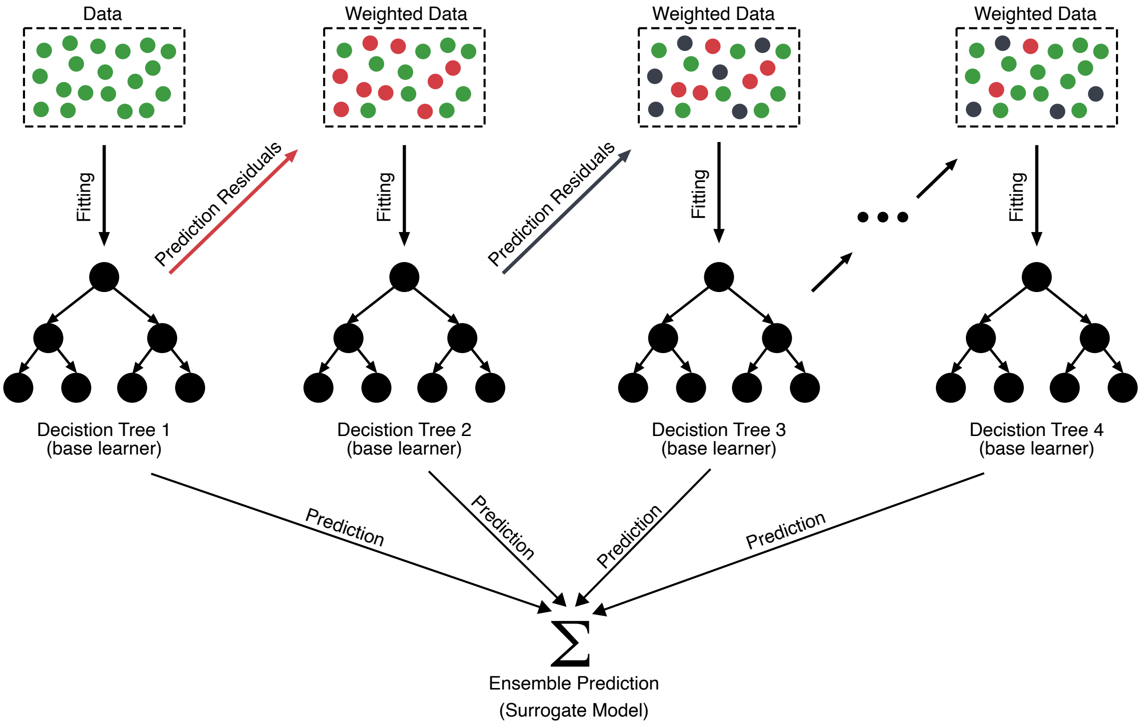


Figure 3.9: Schematic view of GBRT. Modified from Deng et al. [22].

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:

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

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 $r_{ij} = -g_{ij} = y_i - \hat{f}_{j-1}(\lambda_i)$ [44, 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) [72]. 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
    $h_j \leftarrow \textsc{CreateTree}(\{(\lambda_1, r_1), ..., (\lambda_n, r_n)\}, d)$
    for $i = 1$ to $n$ do
        $r_i \leftarrow r_i - \upsilon \cdot h_j(\lambda_i)$
$\hat{f} = \upsilon \cdot \sum_{j=1}^{J} h_j$
return $\hat{f}$
```

3.3.5 Example: Bayesian Optimization for the H-SPPBO

To better understand how the Hyperparameter Optimization process works with a metaheuristic such as the H-SPPBO, a theoretical example using the mathematical symbols and formula from above is explained in the following. This example focuses on BO using a RF surrogate model and the underlying problem to solve will be the TSP.

Let the H-SPPBO algorithm, as explained in Section 3.2.4, be denoted by $\mathcal{A}_\lambda$ with its parameters initialized by the parameter vector λ. This parameter vector is an element of the parameter configuration space $\mathbf{\Lambda} = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_N = w_{\text{persprev}} \times w_{\text{persbest}} \times w_{\text{parentbest}} \times \alpha \times \beta \times \theta \times H \times L$. Furthermore, let each solution constructed by $\mathcal{A}_\lambda$ be defined as a vector $\mathbf{s} = (s_1, ..., s_n) \in V^n$, with V as the set containing all possible city nodes of the TSP problem description. Assuming, that random influences can be ignored, the function describing this solution construction depends only on the parameters and is denoted as $g(\lambda) = \mathbf{s}$. The quality of each of these solutions $f : V^n \rightarrow \mathbb{R}_0^+$ is then defined as the tour length L given by the weights/distances between each of the nodes, described by Equation (3.1), so $f(\mathbf{s}) = L$. Since these solutions are depend only on the parameters λ with which the H-SPPBO algorithm $\mathcal{A}_\lambda$ was initialized, again assuming no random influences, the function definition $\lambda \rightarrow_g V^n \rightarrow_f \mathbb{R}_0^+$ can be simplified to directly output the length L for any given parameter vector λ. Thus, the only function necessary for the metaheuristic is now defined as $f(\lambda) = L = y$, with y used to conform to the previous notation for HPO.

Now, given the Hyperparameter Optimization goal from Equation (3.13), we have defined the solution quality function f and the algorithm $\mathcal{A}_\lambda$, as well as a configuration space $\mathbf{\Lambda}$. In addition, let $F(\mathbf{\Lambda})$ denote a sampling function, that chooses a parameter set λ under a uniform distribution, let $u : \mathbf{\Lambda} \rightarrow \mathbb{R}^+$ be an arbitrary acquisition function, and let $\mathcal{D}_t = \{(\lambda_i, y_i) | i \in [1, t]\}$ be the data set consisting of parameter pairs and their corresponding solution qualities, sampled so far during the HPO process.

Starting with the BO process, we first acquire an initial data set $\mathcal{D}_{\text{init}}$ using our sampling function $F(\mathbf{\Lambda})$. For example, the first entry of this set might contain, among the other parameters, the two values $\alpha = 8, \beta = 2$, resulting in a solution quality (tour length) of $y = 100$, and a second entry might contain the values $\alpha = 7, \beta = 3$, resulting in a solution quality of $y = 90$, so $\{((..., 8, 2, ...), 100), ((..., 7, 3, ...), 90)\} \subset \mathcal{D}_{\text{init}}$. Based on this data, the first RF model is built as explained in Section 3.3.4. For example, the first base learner $h_j(\lambda)$ built in the ensemble of all J learners could select $\alpha < 7.5$ as a split node for the decision tree with a predicted solution quality of $y = 100$ for the side where the condition is true, and $y = 90$ otherwise. This split was chosen because following the prediction of this decision tree using the parameters from the data set $\mathcal{D}_{\text{init}}$ resulted in the lowest mean squared error of all possible splits. Note, that this explanation is an oversimplification of the actual process.

From this split, all further child nodes are also split until some termination criterion is met, e.g. the mean squared error is below a certain threshold. Now that we have trained our base learners, we average over their predictions, resulting in our predictor $\hat{f}(\lambda) = \hat{y}$. This predictor, or surrogate model in terms of the BO, is ideally as close as possible to the actual objective function $f(\lambda) = y$ that we defined earlier. However, the predictor only learns a relationship between an input parameter set and the resulting solution quality (tour length L), not the actual solution path $\mathbf{s}$. Ignoring possible ML-related effects such as overfitting, the more parameter pairs and corresponding solution qualities we have, the more accurate our predictor will be. Nevertheless, since we do not want to run hundreds or thousands of expensive real objective functions calls, we use the BO process, to acquire only new data points that maximize our benefit for training this predictor - quality over quantity of data.

As we progress in the BO process, we can use our trained surrogate model for the acquisition function $u(\cdot)$ to determine the most beneficial new point in the parameter configuration space $\mathbf{\Lambda}$. To do this, we need to compute the mean and variance over our surrogate model and its parameter inputs (see Section 3.3.4). The resulting new point from u, which is a parameter set λ, is evaluated by the real objective function, the H-SPPBO, and this new data point (λ, y) is added to $\mathcal{D}_t$ in the t-th iteration of the BO algorithm (see Algorithm 3.7). The RF surrogate model is then trained again, but now with this new data point, which should increase its accuracy. This process is repeated until a certain number of predetermined objective calls `n_calls` have been evaluated. Finally, the BO algorithm returns the parameter set λ^* from the iteration that produced the best solution quality y. In addition, we also get the final trained surrogate model $\hat{f}(\lambda)$.

4 Implementation

A tool that does everything is optimized for nothing.

4.1 Modules

The implementation used for all the experiments in this thesis combines several theoretical aspects from Chapter 3. The foundation, of course, is the Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO) algorithm implemented as explained in Section 3.2.4. Then a Hyperparameter Optimization (HPO) framework using Bayesian Optimization (BO) was built around this algorithm, along with several other modes of operation, that make use of the metaheuristic algorithm. Although, the package is mostly adaptable to other combinatorial problem types, it has some aspects to it, that were designed with the DTSP in mind. These are emphasized as such.

The entire program package was written in *Python*, with some modules completely imported from established libraries (especially for ML and statistical functionality), some modules consisting of modified libraries that did not quite meet the requirements, and some completely new modules. Figure 4.1 shows the dependency graph starting from the `main` function, with a maximum depth of three references. These modules, their dependencies, and their general functionality are individually explained in the following subsections to provide a better understanding of the experiments and the research process.

4.1.1 H-SPPBO Module

The `hsppbo` module implements a multiprocessing version of the H-SPPBO algorithm (see 3.5). It is initialized using all the parameters discussed in Section 3.2.4:

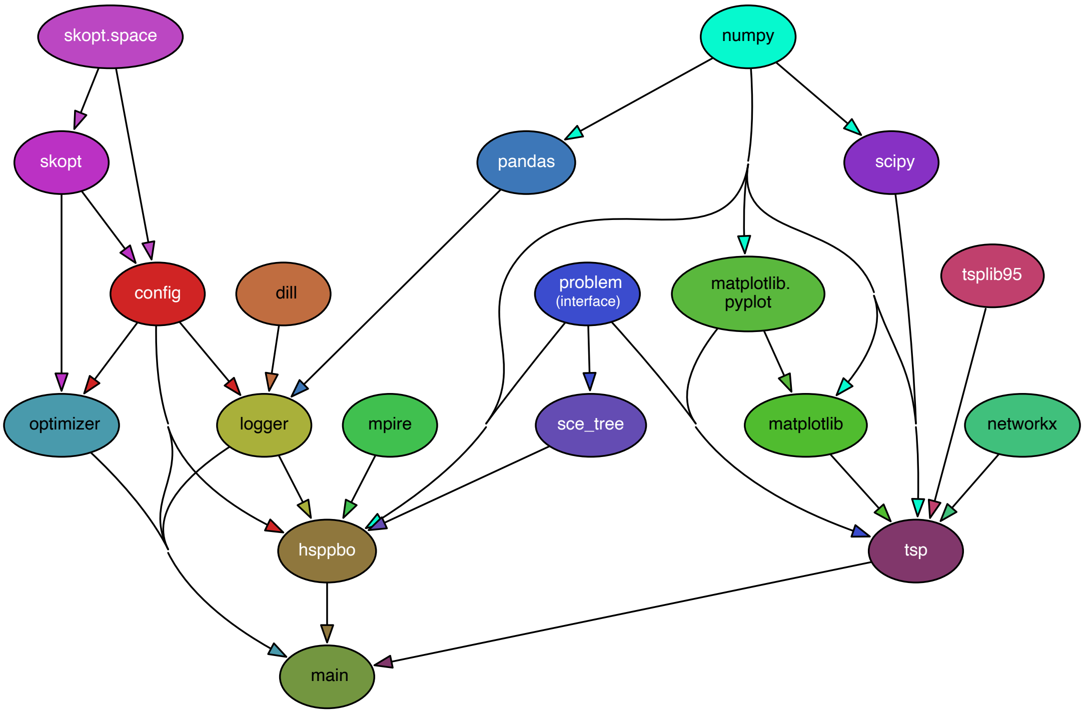


Figure 4.1: Dependency graph of the *XF-OPT/META* python software package

- The three weights $w_{\text{persprev}}, w_{\text{persbest}}, w_{\text{parentbest}} \geq 0$ controlling the influence of their respective populations
- $\alpha, \beta \geq 0$ limiting the influence of the stochastic and the heuristic importance
- A detection threshold $\theta \in [0, 1]$
- A categorical dynamic reaction type $H = \left\{H_{\text{full}}, H_{\text{partial}}, \emptyset\right\}$
- A number of iterations to pause detections for $L > 0$
- A maximum number of iterations $i_{\max}$ (used as a termination criterion).

In addition, some parameters have been fixed in the code using the suggestions from the original paper [58]. The number of SCEs is set to $|\mathcal{A}| = 13$ with three children per SCE. The weight for the random influence w_{rand} is set to $\frac{1}{n-1}$ with n being the dimension of the TSP instance[1].

Since the algorithm is influenced by many random processes, it explicitly provides a function to manually set a random seed, i.e. every time a random number is drawn or something is chosen from a probability distribution, we can expect the same outcome. This

[1] This value might not be ideal for other problem types.

not only makes personal benchmarking more comparable, but also allows reproducible solutions, which is a very important aspect in ML research. The use of this feature for the experiments is discussed further in Section 5.4.

The solution construction process is implemented in a sequential, iterative manner, similar to the description around Equation (3.11). An attempt to convert this task into a matrix computation problem, using the capabilities of the popular *NumPy* [43] library, resulted in worse or similar performance at best. Hence, this approach was not followed in order to reduce complexity. As a result, the computational performance of the solution construction process relies heavily on the calculations of $s_{ik}(P)$, which is basically a lookup of a subset in an ordered set (see Example 3.2.1). For this reason, a key-value hash-map (called `dictionary` in *Python*) was used as the solution population P and a write-once, read-many tuple was used for the potential subset. The crucial function is the following:

Listing 4.1 The *Python* code for the check, if a set is a subset of an ordered subset.

```python
def is_solution_subset(subset: tuple, solution: dict) -> bool:
  try:
    return solution.get(subset[0]) + 1 == solution.get(subset[1])
  except:
    return False
```

This code works well for two important reasons. First, in the case of the TSP, the solution is essentially just a list of node identifiers of size n. Since these are the keys, the values are just their indices in this list. Second, `dictionaries` have a great key lookup performance. The `get`-method returns the value of the given key. Thus, using the first node of the subset as the dictionary key returns its index in the solution sequence. Therefore, adding one to this index and checking again for the index of the second node in the subset should result in equality, if the subset exists. Many other solutions have been tried, some not specific enough for this use case, others not focused on performance. This small function was the result of many optimization efforts and the complete creation process scales linearly with n. Every iteration, a total number of $|\mathcal{A}| = m$ solutions are created, which gives a worst case time complexity of $O(n^m)$.

Other performance improvements are achieved through parallelization. During the solution creation and population update procedure, each SCE operates individually, with only the parent best solution as a reference. Therefore, this process is parallelized using the `mpire` library [1] for problem dimensions $n > 100$. Smaller instances of the TSP were sequentially fast enough to outperform their parallelized counterparts because the overhead to do so was greater than the gain in performance.

SCE Tree Module

The SCEs, their populations P^A_{persprev},P^A_{persbest} and the tree structure are encapsulated in their own module `sce_tree`, which is an extensions of the k-ary tree package `treelib` [14]. We have separate classes for the tree (`SCETree`) and its nodes (`SCENode`). Each node is essentially an SCE $A \in \mathcal{A}$ and holds four variables:

- P^A_{persprev} (tuple): Previous solution of the SCE node
- $f(\mathbf{s}^A_{\text{persprev}})$ (float): Quality of the personal pervious solution
- P^A_{persbest} (tuple): Personal best solution of the SCE node
- $f(\mathbf{s}^A_{\text{persbest}})$ (float): Quality of the personal best solution

The output of the solution quality function $f(\cdot)$ depends on the provided problem type and module, as well as the initialization of the solutions for the populations. This module and the specific case of the TSP is explained in Section 4.1.2. After each SCE node is initialized, they are ordered into an k-ary tree, where $k = 3$ is the number of children each parent has. The structure is similar to Figure 3.5. The `treelib` base package already provides much of this functionality, but the swapping of the SCEs had to be implemented separately. In addition, the algorithm-specific change handling procedures are also present in this module, either resetting the personal best solution of the whole tree (H_{full}) or only from the third level down (H_{partial}).

4.1.2 Problem Interface

The `problem` module is implemented as an interface for all kinds of problem realizations. It uses *Python*'s abstract methods to facilitate future development of other (dynamic) problem types, and provide guidance on all important methods, their parameters and return values. All other modules only use their problem instance through this interface, which further simplifies development. Although the problem module should also contain the functions to validate, randomly generate, and evaluate its corresponding solutions, it does not store these solutions.

One of these problem types using the interface is the `tsp` module, which implements the symmetrical TSP with optional dynamic capabilities to turn every instance into a DTSP problem.

TSP Module

The tsp module is a realization of the problem described in Section 3.1.2. Programmatically, it is based on the tsplib95 library, which provides read, write, and transform functions for files in the .tsp file format proposed by Reinelt [81]. It works very well with TSP instances of the types *EUC_2D* (cities in a two-dimensional Euclidean space), *GEO* (cities as geographic coordinates), and *ATT* (special pseudo-Euclidean distance function), thus limiting the capabilities to these types.

The specific TSP instance is initialized by its name (e.g., rat195) and then loaded by tsplib95. From this instance, the dimension n is stored and the distance matrix D is calculated using the package's Euclidean distance method and numpy [43]. Since all solutions generated by the tsp module are of the same type, they share a solution quality function $f : V^n \longrightarrow \mathbb{R}_0^+$. It works as described in Section 3.2.4, where each solution vector $\mathbf{s}$ is given as a n-dimensional combination of the solution space V, where the positive real value is the length of the traveled tour L.

The DTSP is implemented by enabling the positional swap of two randomly selected city nodes. This dynamic part of the problem is optional and can be initialized separately. The settings correspond to those of Kupfer et al. [58] and are as follows:

- A percentage $C \in [0, 1]$ of how many cities n are changing per dynamic turn
- A dynamic period $T_d \in \mathbb{N}$ defining how often the change is triggered
- A number of minimum iterations i_{min} before the dynamic starts to trigger

That means, that starting from iteration i_{min}, every T_d iterations ($i_{\text{min}} + k \cdot T_d < i_{\text{max}}, k \in \mathbb{N}$) a number of $\frac{n \cdot C}{2}$ distinct pairs of cities are randomly selected and swap their entries in the distance matrix D. After this procedure, the distance matrix is recalculated to reflect the changes.

To evaluate the problem instance itself, some statistical methods have been implemented. First, the length of the optimal solution to each symmetric *TSPLIB* problem is stored in a metadata file. In addition to this information, the module can compute the mean and median distance of a problem, the standard deviation and the coefficient of variation for the distance, as well as the first eigenvalue, the coefficient of quartile variation (CQV), and a so-called regularity index according to [15, 18, 28]. More on the use of these values in Section 5.1.2. Finally, the TSP instances and their solutions can also be visualized using the networkx package [41] (for an example, see Figure 5.2).

4.1.3 Optimizer Module

Several options for the HPO module were considered, but the requirements called for an adaptable library, that was not too closely tied to ML application. Self-implementation was omitted early on to reduce complexity and potential for error. The libraries compared by Yang and Shami [103] were reviewed and checked for potential use with metaheuristics. Ultimately, the `scikit-optimize` library [46] was chosen as the most versatile and adaptable option, while still providing reasonable performance. In addition to the basic RS method, it also gives an implementation of Bayesian Optimization with several options for surrogate models: GP, RF, ET, and GBRT (see Section 3.3.4 for details). And since `scikit-optimize` is built on top of the popular ML library `scikit-learn` [76], it allows other regression models from that library to be used instead. Furthermore, it also provides three popular acquisition functions - probability of improvement (PI), expected improvement (EI) and lower confidence bound (LCB) - of which the two explained in Section 3.3.4 were used for the experiments of this thesis. Overall, `scikit-optimize` provides the mature interfaces and development foundation of `scikit-learn`, while also implementing a customizable Bayesian Optimization workflow.

Because of this already good base library, the actual implementation of the `optimizer` module only contains some interfaces to streamline the interaction between the metaheuristic (in this case H-SPPBO) and the HPO process. On initialization, the optimizer class needs only three things: First is the optimization algorithm to use. As mentioned earlier, these are Random Search, Random Forests, Gaussian process, Random Forests, Extra-Trees, and Gradient Boosted Regression Trees. However, the choice of acquisition function and other algorithm-specific parameters have been preconfigured or left default, depending on the algorithm, which is explained in Section 5.2.1. The second parameter for the optimizer is a reference to the execution object of the objective function $f(\lambda)$. *Python* allows for complete methods to be used as function parameters. Therefore, the `hsppbo` module needed only a special wrapper function that accepts a variable array of parameters λ, executes the algorithm with these parameters initialized for all $i_{\max}$ iterations, and then returns only the quality of the best solution. Mathematically, the entire `hsppbo` module has been reduced to the function $f : \mathbf{\Lambda} \rightarrow \mathbb{R}$, as explained in Section 3.3.4. The third and final parameter is the configuration space $\mathbf{\Lambda}$. It consists of a list of tuples, where each tuple contains the name of the parameter in the `hsppbo` module, and its domains and value ranges to be optimized.

After this initialization, the `optimizer` instance can be invoked to perform any number r_{opt} of repeated optimizer runs. Each optimizer run consists of a number of calls to the objective function f (denoted as `n_calls`), where each `n_call` uses a different parameter set bounded by the specified configuration space $\mathbf{\Lambda}$ and chosen by the acquisition function u (see Section 3.3.4). The random state can also be fixed with the same reasoning as for the hsppbo module. After these n_calls of BO execution, the optimizer module returns a result object containing, among other things, the best parameter set it obtained and the corresponding solution quality, the complete parameter history, and, if used, the trained, underlying regression model. These results, for each optimizer run $i = 1, ..., r_{\text{opt}}$, are collected in a set C.

4.1.4 Logger Module

The logger module captures all the intermediate data and results from the H-SPPBO algorithm and the optimizer module. It is initialized according to the operating mode (see Section 4.3) and automatically creates the necessary folder structure. Then, it outputs an info log about all the environment data concerning the hsppbo, sce_tree, problem, and the optimizer module to precisely capture the runtime conditions of the program. In the case of an optimizer run, it also save the complete results, including the trained regression model, in a so-called "pickle" file using the dill package [71]. This package is an extension of the popular pickle library for serializing and deserializing *Python* objects and adds support for more complex data types to be stored. This way, the various results of the optimizer module can be loaded and used in their entirety at any time, rather than after the run with the in-memory object still present. This greatly improves the analysis process and allows for more complex, flexible post-processing of the data (see Section 5.5 for more details).

4.2 Framework View and Workflows

In addition to the actual module implementation, different possible combinations of HPO and ML libraries and their corresponding configuration options for use with a metaheuristic were explored. This resulted in an optimizer pipeline capable of adapting to multiple problems and metaheuristics, while also logging various aspects of the results and runtime environment. To analyze and present this data in an accurate and meaningful way, a sophisticated analysis pipeline with statistical tests and graphs was also developed. Furthermore, due to the many potential influences on the H-SPPBO algorithm and the HPO process, much thought has been given to ensuring that every aspect is explainable and reproducible to make further research easier and more reliable.

All of these different aspects, modules, and workflows come together in the software called *EXperimentation Framework and (Hyper-)Parameter Optimization for Metaheuristics (XF-OPT/META)* used in this thesis. Since each major part of the framework is modularly implemented, they can be easily exchanged or extended. Let us first look at the optimizer pipeline, where we have three different workflows, each introducing a new aspect[2]:

1. Implement a new optimizer, the problem and metaheuristic remain the same
2. Implement a new problem, the optimizer and metaheuristic remain the same
3. Implement a new metaheuristic, the problem and optimizer remain the same

1.) The optimizer is built on a versatile BO base, that can accept most *scikit-learn* estimators as a surrogate model. The package also includes a general-purpose `skopt.BayesSearchCV` method, that can be used with any estimator returning a score for the provided configuration space. With BO as a state-of-the-art Hyperparameter Optimization method, the possibilities for trying out new modules are vast. 2.) The `problem` interface already guides developers through all the necessary methods and class variables to consider, when implementing a new problem type. However, these have been influenced by the H-SPPBO algorithm for solving the DTSP. Therefore, new problems, especially those of a non-combinatorial nature, may require more customization in their implementation. The documentation for the `tsp` problem class should be of great help for that. 3.) Since the H-SPPBO algorithm is based on the SPPBO framework for metaheuristics, the implementation of the *Python* module was performed with these general principles in mind. This means, that all metaheuristic algorithms, that can be designed using SPPBO can also be easily implemented in *XF-OPT/META*. As with 2), a good starting point would be the well-documented `hsppbo` module.

In order for the analysis pipeline to be fully utilized for all of the above cases, the `logger` module needs to operate correctly. Therefore, it is initialized and called in the main function, instead of being deeply integrated into each module itself. As long as every major module (`problem, optimizer, metaheuristic`) implements the necessary information-providing methods, the `logger` module can adapt to any new integration. From then on, the analysis module (`analyzer`) can be used with any of the results generated by a *XF-OPT/META* mode of operation (explained in the next section).

[2] Of course, it is also possible to implement all three of these modules at the same time.

4.3 Modes of Operation

The *XF-OPT/META* package has three different modes of operation, each of which uses some part of the above workflows: 1) run, 2) experimentation and 3) optimization. Each of these modes serves a different purpose for the thesis, especially with the latter two, because of their relevance to the following chapters. Despite their differences, they all have in common, that they take parameters to describe their problem instance. These inputs are the problem type t (e.g., symmetric TSP, asymmetric TSP, QAP), the instance name p, with a file of that name present in the problem folder, and the optional dynamic intensity C. Thus, each (dynamic) problem instance $\mathcal{P}$ can be described by these three parameters $\mathcal{P} = (t, p, C)$.

4.3.1 Run Mode

The run mode is just a single execution of the metaheuristic algorithm $\mathcal{M}$ on a certain given problem instance $\mathcal{P}$, i.e. a run of the hsppbo module solving the DTSP problem. Besides the problem description explained before, the run mode only requires the parameter configuration λ for the hsppbo module as input. It returns a solution vector $\mathbf{s}$ and the quality of the solution $f(\lambda)$, e.g., for the DTSP it returns the ordered list of city nodes and the length of this tour. This mode also logs the complete run history including the absolute runtime, function evaluations, swaps in the SCE tree, a potentially triggered response mechanism, and the current best solution for each iteration. A high-level template for the run mode is shown in Algorithm 4.1. Primarily, this mode is used to quickly test parameter configurations, new problem instances or other changed aspects of the software workflow.

Algorithm 4.1 XF-OPT/HSPPBO: Run Mode

```
Require: Parameter configuration $\lambda$, problem parameters $(t, p, w_{di})$
  $\mathbb{L} \leftarrow \textsc{InitLogger}(\,)$
  $\mathcal{P} \leftarrow \textsc{InitProblem}(t, p, w_{di})$
  $\mathcal{M} \leftarrow \textsc{Init}\text{HSPPBO}(\mathcal{P}, \mathbb{L}, \lambda)$
  $\mathbf{s} \leftarrow \textsc{Execute}(\mathcal{M}, \mathcal{P})$
  $\textsc{LogResults}(\mathbb{L}, \mathbf{s})$
  return $\mathbf{s}, f(\lambda)$
```

4.3.2 Optimizer Mode

The optimizer mode realizes the Hyperparameter Optimization workflow using Bayesian Optimization. Unlike the run mode, we do not need to explicitly provide any parameters for the metaheuristic. Instead, we provide a parameter configuration space $\mathbf{\Lambda}$ that specifies, for each parameter λ_i of the `hsppbo` module (see Section 4.1.1), which range or categorical values are allowed during the optimization run. For example, the value controlling the stochastic influence α may be any natural number (integer) between 0 and 10. Note that it is possible to set a parameter to a fixed value instead, effectively excluding it from the optimization process. Another new input to this mode is the number of consecutive runs r_{opt} and the number of calls to the objective function `n_calls` for each of these runs (see Section 4.1.3).

Algorithm 4.2 shows the complete workflow of this mode. An impotent step after all the modules have been initialized is to set the random seed for the `hsppbo` and `problem` modules. As explained before, this allows for reproducible results. The random seed for the `optimizer` is set to be equal to the run counter. This way, each optimizer run gets a new randomly-initialized surrogate model with different results, but also ensures that these results are obtained repeatedly. Next, a special execution wrapper is created that acts as a mapping function $f : \mathbf{\Lambda} \rightarrow \mathbb{R}$, giving each parameter configuration a score that the optimizer can decide on. Furthermore, the `logger` module is especially important in this mode, since the `optimizer` returns a variety of information and objects that are useful for later analysis. Each run aggregates the optimal parameters into a set C, to easily view the resulting best configuration. Figure 4.2 also illustrates this workflow.

Algorithm 4.2 XF-OPT/HSPPBO: Optimizer Mode

```
Require: Number of runs $r_{\text{opt}}$, Number of objective call n_calls,
    parameter configuration space $\mathbf{\Lambda}$, problem parameters $(t, p, w_{di})$
  $\mathbb{L} \leftarrow$ InitLogger( )
  $\mathcal{P} \leftarrow$ InitProblem(problem type, instance $p$ and dynamic intensity $w_{di}$)
  $\mathcal{M} \leftarrow$ InitHSPPBO($\mathcal{P}, \mathbb{L}$)
  SetRandomSeed($\mathcal{M}, \mathcal{P}$)
  $f(\lambda) \leftarrow$ ExecuteWrapper($\mathcal{M}, \mathcal{P},$)
  InitOptimizer(optimization algorithm, $f(\lambda)$, $\mathbf{\Lambda}$)
  for $i = 1$ to $r_{\text{opt}}$ do
    results $\leftarrow$ OptimizeParameters(n_calls, random state $i$)
    LogResults($\mathbb{L}$, results)
    $\lambda^* \leftarrow$ GetOptimalParameters(results)
    $C \leftarrow C \cup \{\lambda^*\}$
  return $C$
```

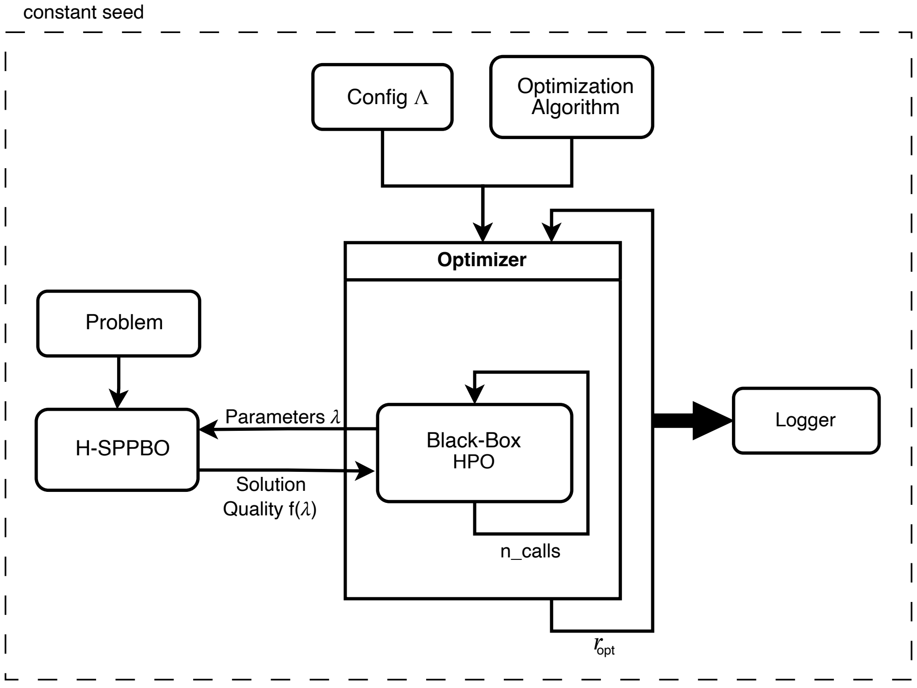


Figure 4.2: Visualization of the optimizer mode workflow.

4.3.3 Experimentation Mode

The main task of the experimentation mode is to repeat multiple runs of a fixed metaheuristic, i.e. using only one parameter configuration. It is essentially a version of the run mode with options for multiple runs. Therefore, the inputs of this mode are also similar, with the parameter configuration λ, problem parameters (t, p, w_{di}) and, additionally, the number of consecutive runs r_{exp}. The results for this mode are also logged in an averaged version, to easily plot the mean development of an algorithm across different random influences. That is why in this mode, the random seed is explicitly not set to a fixed value. See Algorithm 4.3 for an implementation template.

Algorithm 4.3 XF-OPT/HSPPBO: Experimentation Mode

```
Require: Number of runs $r_{\text{exp}}$, parameter configuration $\lambda$, problem parameters $(t, p, w_{di})$
  $\mathbb{L} \leftarrow$ InitLogger( )
  $\mathcal{P} \leftarrow$ InitProblem$(t, p, w_{di})$
  $\mathcal{M} \leftarrow$ InitHSPPBO$(\mathcal{P}, \mathbb{L}, \lambda)$
  for $i = 1$ to $r_{\text{exp}}$ do
      $\mathbf{s} \leftarrow$ Execute$(\mathcal{M}, \mathcal{P})$
      LogResults$(\mathbb{L}, \mathbf{s})$
```

5 Experimental Design and Tests

5.1 Choice of Problem Instances

This thesis builds on the foundation of the Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO) work of [58]. Therefore, the problem category was chosen analogously to the symmetric Traveling Salesperson Problem (TSP). In this way, we can refer to previous work, while also generalizing many different, relevant problems (see Section 3.1). The TSP instance test cases were taken from the popular *TSPLIB* benchmarking suite [81], since they have been tried and tested in many publications and also have the advantage that the optimal solution is known for each of the problems, which enables further comparison with other metaheuristics. Besides the standard 2D Euclidean weights, there are also instances of geographic distance problems or distance matrices. The thesis focuses on 2D Euclidean instances for simplicity, but the software itself can handle most types of TSP edge weights (see Section 4.1.2).

The thesis also aims at solving the Dynamic Traveling Salesperson Problem (DTSP). However, the dynamic part is implemented by the `problem` module itself, and is not a standard part of the *TSPLIB* library. All *TSPLIB* instances have a different number of cities n (called dimension in the following), and often certain characteristics by which the cities are placed in their space, sometimes described in the *TSPLIB* file (under `COMMENT`). An example of such a file is shown in Listing 5.1. To quantify these cases, several statistical values have been calculated for the corresponding distance matrices. These provide a way to select a meaningful, disjoint subset of problem instances without using too many, since the computational cost of running the larger instances can be quite significant.

The selection of problem instances was influenced by two metrics: dimension n and city placement characteristics. Since this implementation of the H-SPPBO algorithm scales linearly with n, and the Hyperparameter Optimization (HPO) process runs the algorithm multiple times (`n_calls`), with the optimization being repeated multiple times (r_{opt}) for each dynamic configuration and problem instance $\mathcal{P}$, the maximum dimension used is around 450 to keep the computation time within a reasonable limit. The lower bound for the dimension n is 50, since smaller instances make it difficult to detect any

Listing 5.1 The *TSPLIB* file for the `bier127` problem instance (node list shortened).

```
NAME : bier127
COMMENT : 127 Biergaerten in Augsburg (Juenger/Reinelt)
TYPE : TSP
DIMENSION : 127
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1   9860  14152
2   9396  14616
[...]
127   3248  14152
EOF
```

placement characteristics. This results in a dimension bounded by the interval $[50, 450]$, which is then roughly divided into smaller instances (50-250 cities) and larger instances (250-450).

With the dimension partitioned, we are left with the statistical measures to analyze the placement characteristic. These measures have been chosen for their expressiveness in graph and distribution problems. Furthermore, it should be possible to calculate them as fast as possible. In order to justify the final choice of TSP instances, various literature was searched for similar procedures. Under the assumption that similarly structured TSP instances share common parameter values for their metaheuristic solvers, the following problem classification aims to be very thorough, so that all parameters obtained for the smaller instances can later be used for the larger instances as well.

5.1.1 Statistical Measures for Analysis

The city placement characteristic was determined using the following statistical values, which was computed for each *TSPLIB* instance over the corresponding distance matrix D using the *NumPy* package [43]:

- The mean μ, median $\tilde{d}$ and standard deviation σ
- The coefficient of variation c_{v}
- The coefficient of quartile variation (CQV)
- The regularity index R
- The first eigenvalue λ_1
- The “eigen gap” $\Delta\lambda_{1,2}$, i.e. the gap between the first two eigenvalues

The mean, median, and standard deviation are common choices for analyzing data sets. These three values already make it possible to give a first impression of how evenly the city nodes are distributed in Euclidean space. For example, if the mean distance between nodes differs greatly from the median with a high standard deviation, we can assume that the instance is somewhat unevenly distributed. However, since these values are absolute and therefore dependent on the problem and its distance scaling, they cannot be used for comparison across all instances. Since we want to identify distributions and clusters within the TSP instances, measures of statistical dispersion were preferred for further calculations. These provide insight into how compressed or stretched out a data set is. Furthermore, only dimensionless metrics were considered.

One possible measure of dispersion is the coefficient of variation, which improves on the standard deviation by effectively normalizing it by division with the mean: $c_\mathrm{v} = \frac{\sigma}{\mu}$. This results in a relative value that can be used comparatively. However, the coefficient of variation tends to overexpose outliers, which may be undesirable when classifying highly clustered instances. Another measure is the CQV, which is a robust version of the coefficient of variation and therefore less sensitive to outliers [7]. It is defined as $CQV = \frac{Q_3 - Q_1}{Q_3 + Q_1}$, where Q_1 and Q_3 are the first and third quartiles of the distance matrices distribution.

The regularity index R is a metric that was developed especially for the quantification of spatial distributions by Clark and Evans [15]. It is defined as the ratio between the median distance between each nearest neighbor r_A and the median distance between nearest neighbors under the assumption of a perfect random distribution r_E, specified by a density ρ, so that $R = r_A / r_E$. Thus, a value of $R = 1$ would indicate a completely random distribution, while $R = 0$ would suggest, that all nodes are located at the same position. In order to compute this measure effectively, some considerations had to be made. The numerator r_A is easily calculated by using the minimum function over all possible distances for each node [28]:

$$r_A = \frac{1}{n} \sum_{i \neq j}^{n} \min(d_{ij}) \tag{5.1}$$

In this case, however, the distance under random distribution depends on the area A and the number of nodes n, with a point density of $\delta = n/A$. The formula for r_E is described by a Poisson process for complete spatial randomness. By making some adjustments to incorporate a sense of absolute distance and taking the expectation of the resulting probability distribution, we get the following formula [28]:

$$r_E = \frac{1}{2} \sqrt{\frac{A}{n}} \tag{5.2}$$

The area A covered by the nodes, i.e. the convex hull of the graph, was obtained using the *SciPy* library [97], which provides algorithms for scientific computing in *Python*. Although not originally applied to the TSP, the works of Crişan et al. [18] and Dry et al. [28] give an insight into the suitability of the regularity index for this problem category, concluding that it is highly significant, even if not perfect. Finally, in an application of the research around spectral analysis of graph problems and the TSP, the first two (largest) eigenvalues were computed over the distance matrix D. While the first eigenvalue can be related to average length of the Hamilton cycle in a TSP instance [21], the "gap" between the first two eigenvalues could be used as a measure of connectivity [67].

To have a larger sample set, all of these values were calculated for all *TSPLIB* instances with a dimension less than 1000 and a valid edge weight type for the XF-OPT/META package. This excluded *ATT*, *EXPLICIT*, and *CEIL_2D* problems, but included *GEO* and *ATT*, which resulted in metadata for 118 problem instances.

5.1.2 Classification

Using these statistical measures, three different methods were employed to classify these 118 TSP instances. Since these statistics, except for the regularity index R, do not provide qualitative information about the type of class the instance belongs to, all of the graphs were also visualized using the *NetworkX* package. This made it possible to interpret the resulting problem groups by formulating the similarities suggested by the classification. The exploration and application of each of these methods has yielded mixed results, with one clear winner.

Method I - Regularity Index

The first method is to use certain value ranges of the regularity index R to discriminate between structures. As mentioned above, this procedure and the applicable ranges have already been validated for the application to the TSP by previous work. To further replicate the results obtained by Crişan et al. [18] and Dry et al. [28], additional TSP instances from the University of Bonn's *Tnm* test data set [48] and a triangle lattice generated using the aforementioned *NetworkX* package were used. The implementation of this thesis was able to successfully reproduce the R values from both papers, i.e., all the same values for the *Tnm* TSP and a value of $R = 2$, for a highly regular, uniformly distributed triangle lattice [28]. With this foundation established, the first method was applied to the selected TSP instances using the following groups and their ranges:

- Heavy clusters: $R < 0.3$

- Semi-clustered: $0.3 \leq R < 0.8$
- Random Distribution: $0.8 \leq R < 1.2$
- Semi-regular: $1.2 \leq R < 1.4$
- Regular: $1.4 \leq R$

This first method generally worked well and was able to classify each problem into a satisfactory group. However, it was heavily influenced by higher dimensions and artificial patterns, such as with *pcb442* or *ts225* (see Figures A.1 and A.2 for visualizations), where artificial is meant to describe that the placement of the cities is clearly influenced by a pattern. These instances were almost all incorrectly classified as randomly distributed ($R \approx 1$). This method alone would not be able to reliably and disjunctively classify the instances.

Method II - Eigenvalues of the Distance Matrix

The second method, inspired by Lovász [67], of using the gap between the first two eigenvalues of the distance matrix D proved to be impractical to implement, since it could not be calculated directly on the distance matrix, and would instead use its Laplacian, which is computationally infeasible, especially for larger instances. The paper also states its theory on a positive semi-definite matrix, which the distance matrix used here are not. However, applying the TSP-related approach of Cvetković et al. [21] of using only the first eigenvalue as a classifier for Hamiltonian path length yielded interesting results. While most of the resulting eigenvalue groups were very unsatisfactory, there was one group consisting of all the artificially structured TSP instances. Therefore, a combination of the two methods was the logical next step.

Method III - k-means Clustering

The first two methods already identified the regularity index and the first eigenvalue as expressive metrics for classification. With the coefficient of quartile variation (CQV) as an additional robust measure of dispersion, a successful classification should be possible. However, it is not practical to manually apply value ranges to three metrics. Therefore, a cluster analysis was performed using the k-means algorithm.

The problem description for the k-means algorithm is given for a set of data points n in $\mathbb{R}^d$. Given a variable number of k centers, select the position of each center that minimizes the total squared distance from each point to its nearest center [3]. This NP-hard problem was originally solved by Lloyd [66], resulting in “Lloyd’s algorithm”, or “Voronoi iteration”.

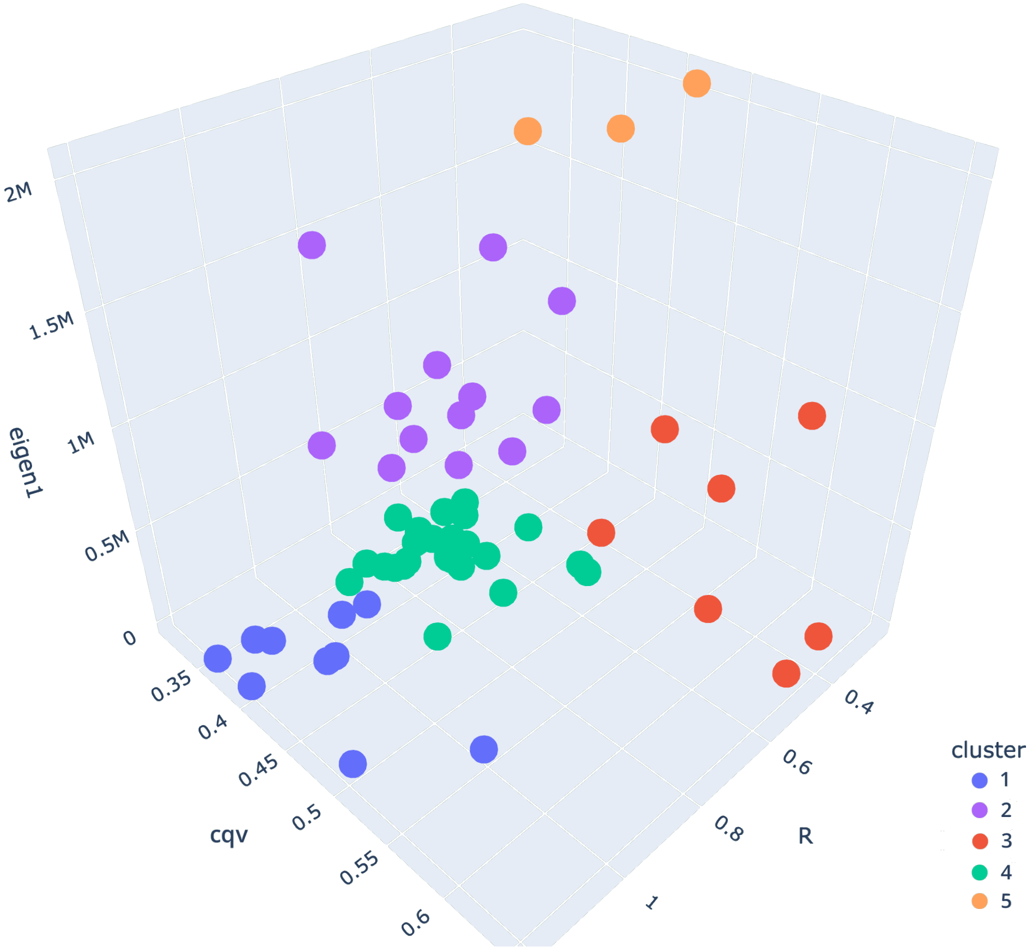


Figure 5.1: Visualization of the k-means cluster analysis applied to 118 TSP instances using $k = 5$ clusters. Each data point is an instance placed into three-dimensional space using its regularity index R, its first eigenvalue (eigen1), and its coefficient of quartile variation (CQV).

It randomly places k centers in Euclidean space and assigns each point to its nearest center. It then computes a Voronoi diagram over the k sites, integrates these cells, and moves the center to the centroid of each cell. This process is repeated until a stagnation condition is registered. The algorithm used for our application is called *k-means++* and is based on this foundation. Instead of opting for an exact solution, Arthur and Vassilvitskii [3] propose an approximate procedure to compute the initial k centers. This way, fewer iterations of the otherwise same algorithm are needed to achieve converging behavior and thus satisfactory clusters.

The *k-means++* algorithm was applied to all of the 118 instances, with the first eigenvalue λ_1, the regularity index R, and the coefficient of quartile variation (CQV) creating a three-dimensional real Euclidean space within these instances were places. Furthermore, a value of $k = 5$ clusters proved to be the most effective, since it corresponds to the groups mentioned in the first method and results in the most visually verifiable coherence between instances. As shown in the 3D scatter plot of the clustering in Figure 5.1, this third method resulted in fairly consistent clusters and even managed to separate most of the artificial patterns from the rest, especially by using the eigenvalue. However, due to the nature of k-means clustering, there are no resulting structural properties implied for the generated clusters. We can only infer the structural properties mentioned above by looking at the ranges and visualizations for the clustered instances.

The separation between instances shown in the scatter plot is fairly profound, with only the red and orange clusters looking a bit loosely connected. Nevertheless, the third method is convincing in most respects, making it an improvement over the mere value ranges in the first two methods. Therefore, the results of the clustering method are used to categorize the structure of the problem instances. To do this, the clustered groups must first be related to a structural property, as explained above. This is done by examining the value ranges of regularity index, as in the first method, the first eigenvalue, and by looking at the visualized instances to discover common patterns or to verify the implications of R or the CQV value. The resulting structural groups and their distinctive value ranges are as follows:

1. Random to nearly regular distribution: $R > 0.9$
2. Smaller, slightly clustered areas with otherwise random structure: $R \in [0.55, 0.9]$ and $\lambda_1 < 500000$
3. Artificially structured with certain patterns of medium clustered regions, with small distinct holes within the distribution: $R \in [0.55, 0.9]$ and $\lambda_1 > 500000$
4. A few highly clustered areas: $\lambda_1 > 1700000$
5. Dispersed and highly clustered areas with few or no city nodes in between: $R < 0.6$ and $CQV > 0.5$

The first, second, and fourth groups are very consistent, with only a few outliers present. The other two groups, three and five, consist of intermediate structures, that could not be placed in any other group, but are also too different from each other, to justify only one group.

With this classification in mind, 10 instances from each structural group with a smaller and a larger instance were selected for the thesis. The dimension limitation already excluded many instances, which made some choices very obvious. For example, the first group has no instance with a size between 250 and 450 nodes. So the next largest instance, `rat195`, was chosen as a replacement. Other groups had only a few possible candidates for each dimension class, so they were chosen randomly. The selected instances are the following, where the enumeration is coherent with the stated structural groups:

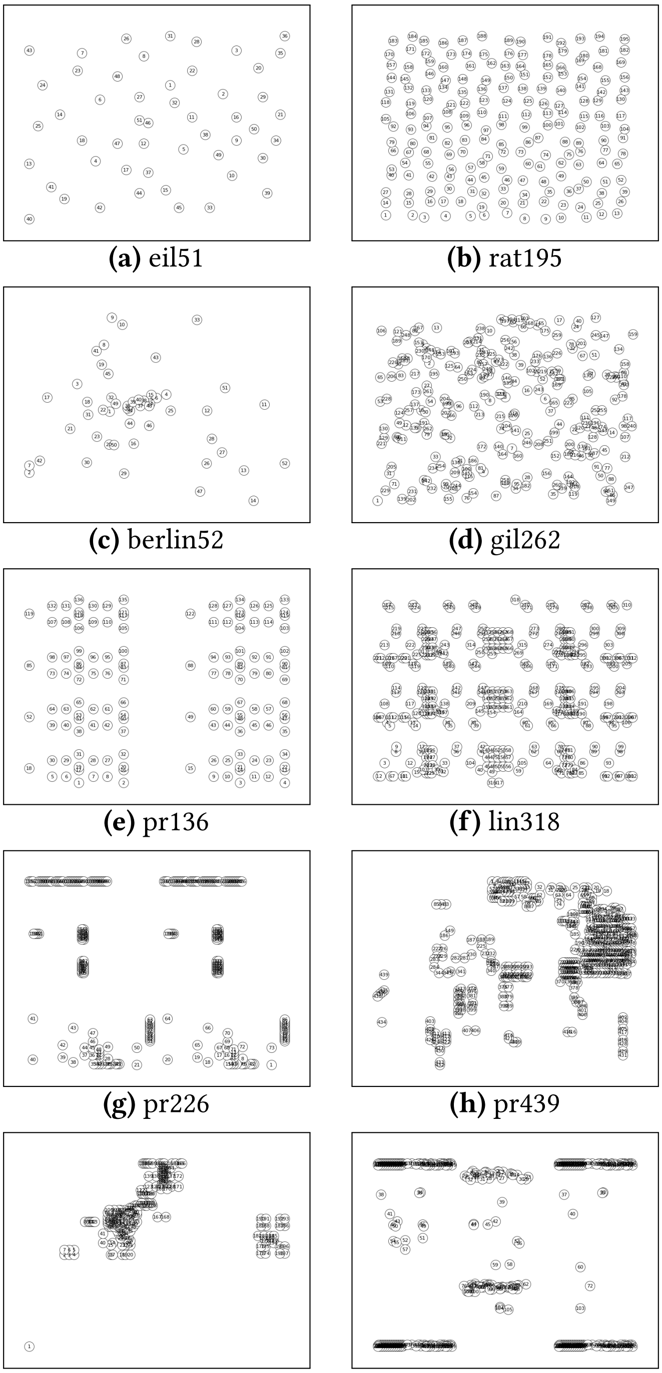
1. eil51, rat195
2. berlin52, gil262
3. pr136, lin318
4. pr226, pr439
5. d198, fl417

Figure 5.2 shows the visualizations of these 10 instances, where each row of figures depicting a cluster group from 1 (top) to 5 (bottom), with the left column depicting smaller instances, and the right column depicting larger instances. Each figure is labeled with its instance name.

5.2 Choice of Optimization Methods

The choice of HPO pipelines (i.e., acquisition function + surrogate model) to be tested was largely dictated by the ML library chosen, `scikit-optimize`. This is due to the fact that there is very little research on applying Hyperparameter Optimization to the tuning problem of metaheuristics. The only relevant work found was by Yin and Wijk [106], who have a very similar application, using HPO to tune an Ant Colony Optimization algorithm. However, they focus only on the BO using the GP as a surrogate model. Furthermore, it was noticed, that many ML-related explanations of BO ignore the fact, that the surrogate model can even be changed. Thus, in addition to the originally proposed choice of a Gaussian process, regression models from the field of ensemble learning were also used. Although both Random Forests and Gradient Boosted Regression Trees are based on decision trees, they differ greatly in the way they use their basic learners for the ensemble predictor (averaging vs. boosting). Therefore, they cover a large part of state-of-the-art ensemble methods.

As already described in Section 4.1.3, the following surrogate models are provided: Random Forests (RF), Gaussian process (GP), Random Forests (RF), Extra-Trees (ET), and Gradient Boosted Regression Trees (GBRT). All of these models were used in the experiments, with the exception of the standard RF, since ET already improves upon it. This range of models should ensure that as many optimization scenarios as possible can be tested to find the ideal method for metaheuristics, or at least the H-SPPBO algorithm. In addition, RS was also used and provides a good baseline, since it is a model-free method



(a) eil51

(b) rat195

(c) berlin52

(d) gil262

(e) pr136

(f) lin318

(g) pr226

(h) pr439

(i) d198

(j) fl417

Figure 5.2: Visualizations of the TSP instances used in the experiments. Each row of figures is a cluster group, where the left column depicting smaller instances and the right column depicting larger instances. Each figure is labeled with its instance name.

and therefore makes no assumptions about the objective function. However, the choice of acquisition function was made with respect to the surrogate model used. The following subsection describes how the BO process was initialized for each method.

Table 5.1: The HPO methods used and their initialization values.

Estimator	Acquisition Function	Sampling Method	Number of initial points
RS	-	Uniform	-
BO-GP	PI ($\xi = 0.01$)	Hammersley	10
BO-ET	EI ($\xi = 0.01$)	Uniform	10
BO-GBRT	EI ($\xi = 0.01$)	Uniform	10

5.2.1 Optimizer Initialization

The initialization process can have a large effect on the outcome of the HPO process. For all methods, a set of 10 starting points has been sampled using a specified method before the acquisition function was utilized for sampling. The RS algorithm has only this sampling method for initialization. Besides the default uniform random number generator, there are several low-discrepancy sequences, such as the Hammersely sequence, and a Latin hypercube sampling method to choose from. Since we want to use RS as a baseline in later discussion, the default uniform sampling was used.

As mentioned above, the work of Yin and Wijk [106] was a useful starting point for the GP surrogate model. They tried out different initializations for the BO, including all three acquisition functions (PI, EI, ower confidence bound (LCB)), initial sampling methods, noise functions and values for improvement ξ. After reviewing their results, the most appropriate values were used to initialize the GP model used in the thesis. The acquisition function was selected as probability of improvement (PI) with an improvement rate of $\xi = 0.01$, because it shows fast convergence behavior with good results. The kernel function was selected as a Matérn kernel with $\upsilon = 5/2$, which is the default of the `scikit-optimize` library. It also uses an additive white noise kernel to account for a noisy objective function. However, instead of the default Gaussian noise scaled by the variance during the optimization process (see Equation (3.15)), a white noise kernel with a constant variance of 0.7 was chosen, to prevent bad search areas from gaining too much relevance for sampling [106]. Furthermore, a low-discrepancy Hammersely sequence was chosen over the default uniform distribution for initial sampling, as the GS model appears to benefit from more evenly spaced out points [13].

No evidence was found to apply this reasoning to the other two models, ET and GBRT, so instead, they use the standard uniform sampling and the default EI acquisition function with $\xi = 0.01$ suggested by the library. Also, no additional noise was added to the methods. The initialization values for each estimator are summarized in Table 5.1.

As explained in Section 4.3.2, the random seed of each method was set according to the current iteration of the optimization run. This ensures new samples and results for each optimization run, while also providing reproducibility.

5.3 Choice of Parameters and Value Ranges

Since the focus of this thesis is on finding ideal parameter combinations using ML-based Hyperparameter Optimization algorithms, we do not need to specify exactly which parameter values we want to test, as many other metaheuristics work does prior to experimentation. However, we still need to specify which of the available parameters of the H-SPPBO algorithm we want to optimize automatically, if so, what range these parameters are sampled from during the process, and if not, what static parameter value we should assign and why.

In general, the parameter configuration space Λ should be as broad as computationally feasible and logically useful to avoid unwanted effects like the "Floor and Ceiling effect" [99, p. 47]. Otherwise, it would impose certain expectations on the optimization process and its parameter choices, and it would also limit the potential for interesting new global optima. For example, although we can expect good results from $\beta = 5$, due to many other parameter influences, we cannot know for sure if a value of $\beta = 10$ might also be a good choice in some parameter combinations.

As already explained in Section 4.1.1, the following are all the available parameters for the H-SPPBO algorithm that qualify for optimization, their corresponding HPO data type, and their default range:

- $w_{\text{persprev}}, w_{\text{persbest}}, w_{\text{parentbest}} \geq 0$ (real)
- $\alpha, \beta \geq 0$ (integer)
- $\theta \in [0, 1]$ (real)
- $H = \left\{H_{\text{full}}, H_{\text{partial}}, \emptyset\right\}$ (categorical)
- $0 < L < 100$ (integer)

Unfortunately, these standard data ranges were too broad to calculate in time for this thesis. Therefore, parameter influences, effects, and existing justifications for limitations were investigated. The parameters H and θ are closely related to the hierarchical part of the H-SPPBO algorithm, so there is almost no reference to existing implementations or papers, except for Janson and Middendorf [53] with their hierarchical version of a Particle Swarm Optimization (PSO) algorithm. Regarding the parameter H, Kupfer et al. [58]

found that the H_{partial} response often outperforms the H_{full} response, and the changing heuristic influence controlled by β during the optimization runs suggests an interesting behavior of this parameter. Thus, both response types were used. In order to use this categorical value for all of the aforementioned surrogate models, a one-hot encoding was applied prior to the optimization process. It maps each categorical value to a bit vector containing only a single 1 and 0 otherwise. For example, a possible one-hot encoding for H could be 01 for H_{full} and 10 for H_{partial}.

The detection rate θ seems to benefit from values higher than 0.1, and gets mixed results from values between 0.25 and 0.5, depending strongly on the problem instance and its dynamic intensity C [58]. Since values higher than 0.5 would render the need for a change handling procedure obsolete because the changes would most certainly be undetected, a range of $\theta \in [0.1, 0.5]$ was tested.

The L parameter is also related to this particular implementation of the algorithm's dynamic handling. Since its only purpose is to disable detection right after each change interval by the DTSP instance, it only needs to be high enough to account for the rearrangement of the SCE tree. And since it introduces the risk of unfair prior knowledge of the change interval, it should be as small as possible, since a value equal to the dynamic period T_d would make it impossible for the algorithm to falsely detect a change. Considering the theoretical "worst case" behavior of a complete reorganization on all three levels of the ternary tree with 13 SCEs, a value of $L = 5$, also used in [58], is very reasonable.

The values for α and β are used in almost every ACO variant and many metaheuristics in general. Since the work of [25], most papers on ACO variants use α and β values between 0 and 5, often following the recommendation by Dorigo ($\alpha = 1, \beta = 5$). However, this only really applies to these ACO versions used on symmetric TSP instances, while the H-SPPBO algorithm combined with DTSP instances behaves very differently. In addition, works such as [92, 96, 102] imply that good parameter combinations may differ greatly from the original recommendation depending on the problem type and algorithm, and have success using values of 10 or higher. Since α and β are exponents, and the expression in which they are used (see Equation (3.10)) is normalized to a probability anyway, the values should be considered relative to each other rather than absolute. Therefore, they should be at least 10% apart to cover any reasonable combination of the two. Larger value ranges would carry the risk of reducing the other parameter to a value where it loses its significance and is effectively deactivated, while this should be preferably achieved by choosing a value of 0. It could also be argued that, based on this logic, a real value chosen from $[0, 1]$ would also result in similar expressiveness. However, natural numbers are most often used for these parameters, and make for a much easier comparison. This resulted in a range of $\alpha, \beta \in [0, 10]$, with $\alpha, \beta \in \mathbb{N}$.

The three weights w_{persprev}, w_{persbest}, $w_{\text{parentbest}}$ present an interesting significance. Although they are specific to this algorithm, they are based on the work by Lin et al. [65], which in turn is based on the standard pheromone evaporation coefficient ρ used in the standard Ant Colony Optimization and its variants. This value, which acts as a weight, is often chosen as $\rho \in (0, 1]$. In [65], a global population is introduced into the algorithm, and the total weight is divided by the number of iterations, k, in which the solution is retained, or by the number of solutions generated per iteration, always using a specific formula that does not allow the full range of real numbers to be chosen. They also tested several values for the total weight (up to $w_{\text{total}} = 192$) and the elite solution weight (up to $w_{\text{elite}} = 10$), and often found that higher values were beneficial to solution quality. However, these results were obtained with α set to 1 and β set to 5. This may not be optimal, since the three weights are summed and then influenced by the control parameter α, which then has to be compared with its factor, the heuristic part, and its control parameter β. This means that the summed base, which is the three weights plus a fixed random weight (w_{rand}), is directly compared to the base of the heuristic term, which is the inverse of an element of the distance matrix $1/d_{ij}$ ($d_{ij} \in D$). This value should be less than 1 and greater than 0, at least for a non-normalized, Euclidean distance matrix over common TSP instances. Therefore, the sum of the weights should also be close to this range of values. This allows the parameters α and β to control only the influence of their respective bases, and not also to serve as a normalization exponent to bring the factors to a comparable level.

Furthermore, being able to take each weight from the entire real space of $[0, 1]$ ultimately has the same effect as having predefined formulas for each weight category that scale with a total weight as in [65]. If the term is scaled by an exponent anyway, a weight difference between 0.01 and 1 has the same influence as between 1 and 100. Finally, since the random weight is fixed to $w_{\text{rand}} = 1/(n-1)$, where n is the dimension of the TSP instance, the other weights must be comparable to it. A theoretical minimum of $n = 2$ cities results in a random weight of 1, while a maximum weight cannot be formulated, but is always greater than 0. This all led to the three weights being drawn from the real interval $(0, 1)$. However, since the python package `scikit-optimize` can only use closed real intervals, and the largest data set has a size of 450, which results in $w_{\text{rand}} = 0.0022$, the interval $[0.001, 0.99]$ was used.

To conclude, the final parameter ranges are as follows:

- $w_{\text{persprev}}, w_{\text{persbest}}, w_{\text{parentbest}} \in [0.001, 0.99]$
- $\alpha, \beta \in \{x \in \mathbb{N} | 0 \leq x \leq 10\}$
- $\theta \in [0.1, 0.5]$
- $H = \left\{H_{\text{full}}, H_{\text{partial}}\right\}$

- $L = 5$ (not optimized)

5.4 Testing Procedure

The basis of each test was the H-SPPBO algorithm implemented as explained in Chapter 4, running $i_{\text{max}} = 2600$ iterations on a DTSP instance, hereafter referred to as a H-SPPBO execution. The dynamic part (swapping a percentage of cities) happened every $T_d = 100$ iterations, starting at iteration 2000. Therefore, a dynamic change was triggered every $2000 + k \cdot 100 < 2600, k \in \mathbb{N}$ iterations. In this context, an important change was made to the solution quality function, which returns the tour length at the end of each H-SPPBO execution to the optimization process. Instead of reporting only the last solution, i.e. the global best solution at iteration i_{max}, which would effectively only evaluate the response to the last dynamic change between iterations 2500 and 2599, all solutions just before each next dynamic change are stored. This means, that between iterations 1999 and 2600, a total of six solutions are stored, which are then computed to solution qualities (lengths) and then averaged to $\overline{L}$. Thus, we are not only able to evaluate the dynamic response for all five trigger points, but can also include the solution quality of the static TSP up to iteration 1999, albeit with a small weighting of $1/6$ in the average. The run and experimentation modes are not affected by this change.

Each of the 10 problem instances mentioned above was used, with a varying dynamic intensity set to $C \in \{0.1, 0.25, 0.5\}$. The experiments performed in this thesis can be divided into three main parts. First, optimization data was collected for all four HPO methods. Second, multiple optimization runs of only the best performing HPO algorithm were executed. And third, the best parameter sets from the previous test were used to repeat multiple experiment runs. An overview of these tests and their execution parameters is summarized in Table 5.2. The details and rationale for each part are explained in the following.

Table 5.2: The three experimentation parts and their execution parameters

Mode of Operation	HPO Method $\mathcal{A}$	n_calls	runs ($r_{\text{opt}}/r_{\text{exp}}$)	Dynamic Intensity C	Number of Instances	H-SPPBO Executions
optimizer	RS, GP, ET, GBRT	30	3	0.25	5 (small)	1800
optimizer	GBRT	60	6	all	5 (small)	5400
experimentation	-	-	20	all	10	2 x 600

The first part deals with the selection of the most appropriate HPO method for the H-SPPBO algorithm and its dynamic problem instances. For this purpose, we performed three optimization runs for each of the four optimization algorithms, i.e., RS, GP, ET, GBRT, and selected the algorithm based on the highest average solution quality and convergence rate. Each of these runs made 30 calls to the H-SPPBO algorithm (`n_calls` = 30) on only the five smaller problem instances and only the medium dynamic intensity ($C = 0.25$). Thus, four optimization methods, each performing three optimizer runs with 30 objective calls per run, on five instances with one dynamic intensity. This resulted in a total of 1800 H-SPPBO algorithm executions. Some shortcuts had to be taken in this step to save some execution time. Since a full evaluation of all three dynamic intensities would take too long, we limited ourselves to the smaller TSP instances and the medium dynamic intensity of 0.25. Furthermore, we limited our BO process to only 30 objective function calls instead of 50 or more, which means that a convergent behavior should have started after about 20 calls [45]. This de facto requirement for fast convergence can also be seen as a demand on the hyperoptimization method we want to choose. Finally, the stability or robustness of the parameters chosen by the optimizers is of only secondary importance in this part, so that three runs are sufficient for an average solution quality.

In the second part, multiple evaluations were performed using all three dynamic intensities, with the most appropriate optimization method selected from the previous part. These results give us insight into what the optimal parameter selection might be for each, or potentially all, problem instances. To do this, we executed six optimizer runs r_{opt} using one optimization algorithm, each run making 60 objective calls (`n_calls`) to the H-SPPBO algorithm, again only on the five smaller problem instances, but on all three dynamic intensities $C \in \{0.1, 0.25, 0.5\}$ for a total of 5400 H-SPPBO algorithm executions. As explained earlier, we could not use the larger instances due to time constrains. However, to get a more robust sense of "good" parameter configurations, the number of optimization runs was increased to six.

The resulting data sets $\mathcal{D}$ of these first two experimentation parts can best be expressed by a four-dimensional tensor, with the dimensions being the HPO method $\mathcal{A}$, the TSP instance p, the dynamic intensity C and the number of consecutive optimization runs r_{opt}. Each element of this tensor consists of an optimizer result, consisting of, among other things, the entire parameter history of this run $\mathcal{H}(\lambda, f(\lambda))$, the best parameter configuration found λ^*, and the trained surrogate model $\mathcal{M}$. So each data entry can be characterized by the function $\mathcal{D}(\mathcal{A}, p, C, r_{\text{opt}}) \to \{\mathcal{H}(\lambda, f(\lambda)), \lambda^*, \mathcal{M}\}$, where $\mathcal{D}^i_{\mathcal{A}_j, p_k, C_l} \in \mathcal{D}(\mathcal{A}, p, C, r_{\text{opt}})$ and $1 \leq i \leq r_{\text{opt}}$.

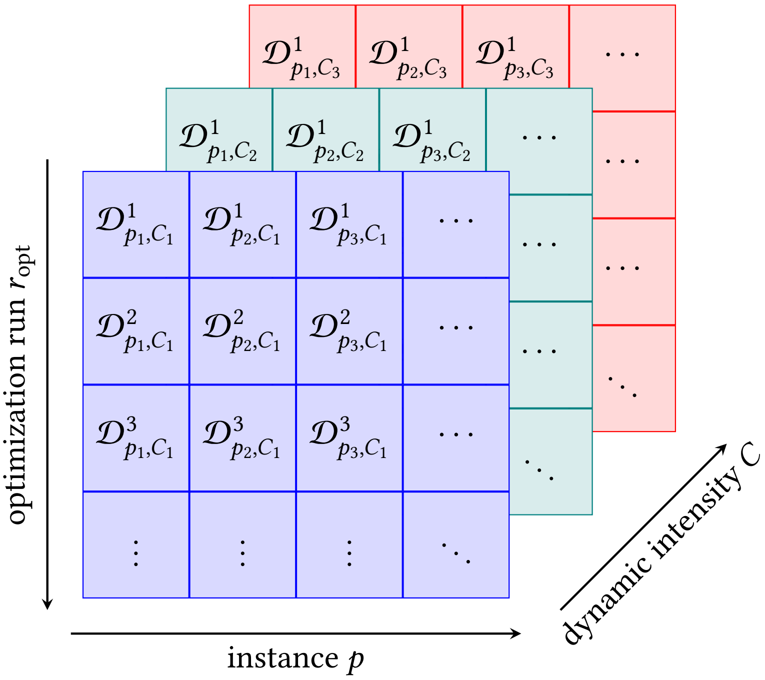

$\mathcal{D}^1_{p_1,C_3}$
$\mathcal{D}^1_{p_2,C_3}$
$\mathcal{D}^1_{p_3,C_3}$
$\mathcal{D}^1_{p_1,C_2}$
$\mathcal{D}^1_{p_2,C_2}$
$\mathcal{D}^1_{p_3,C_2}$
$\mathcal{D}^1_{p_1,C_1}$
$\mathcal{D}^1_{p_2,C_1}$
$\mathcal{D}^1_{p_3,C_1}$
$\mathcal{D}^2_{p_1,C_1}$
$\mathcal{D}^2_{p_2,C_1}$
$\mathcal{D}^2_{p_3,C_1}$
$\mathcal{D}^3_{p_1,C_1}$
$\mathcal{D}^3_{p_2,C_1}$
$\mathcal{D}^3_{p_3,C_1}$
$\cdots$
$\vdots$
$\ddots$
optimization run r_{opt}
instance p
dynamic intensity C

Figure 5.3: 3D tensor representation of the data set $\mathcal{D}_{\mathcal{A}=GBRT}(p, C, r_{\text{opt}})$ of the second experimentation part.

However, since the first part only used only a single dynamic intensity C and the second part used only a single HPO method $\mathcal{A}$, each of the resulting data sets can be reduced to a 3D-tensor. Figure 5.3 shows a representation of this explanation for the second part and its data sets.

After this second part, we had 15 sets of optimal parameters, one for each combination of problem category (five groups as explained in Section 5.1.2) and dynamic intensity ($C \in \{0.1, 0.25, 0.5\}$). Although the urge to aggregate these parameter configurations in some way (e.g., an average over all three dynamic intensities for each problem category) is tempting, because it would not only simplify the next part, but also provide a multipurpose parameter recommendation, it was omitted for several reasons. First, it introduces a whole new perspective to this work, namely how parameters can be generalized across different problem descriptions. However, the goal of this work, as explained in the Approach, is to directly apply Hyperparameter Optimization to metaheuristics and validate it as a viable option. Any generalization would compromise this goal and open up the research question to several new variables. Second, by looking at the results of the second part of the experiment, it was clear early on that there were huge differences between the optimization runs and each of their optimal parameter sets. Except perhaps for α and β, any aggregation of parameters would have been a mostly arbitrary choice, with a few exceptions. This circumstance is discussed further in Section 6.2.

Table 5.3: The values of the general purpose parameter set used as a reference in the third part of the experimentation.

α	β	w_{persprev}	w_{persbest}	$w_{\text{parentbest}}$	θ	H
1	5	0.075	0.075	0.01	0.25	partial

The third and final part verifies the previously selected “optimal” parameter selection. These 15 parameter sets were tested on their respective problem instances and dynamic intensities, with 20 experimental runs r_{exp} each using the experimentation mode (see Section 4.3.3). This time, all 10 instances were used. Since the problem classification effort was also made to obtain coherent instance groups, these parameter sets acquired using the smaller instances were also applied to the larger instances of each group. This results in 600 runs of the H-SPPBO algorithm. As a reference for how well these HPO parameter configurations perform, a general purpose parameter set (see Table 5.3) was also applied to all 10 instances with all three dynamic intensities, again repeating each experimental run 20 times, yielding another 600 H-SPPBO algorithm executions. The values for α, β, w_{persprev}, w_{persbest}, and $w_{\text{parentbest}}$ were taken directly from the original work by Kupfer et al. [58], while the choice of θ and H was influenced by how well a particular value performed in their results.

All of these tests were conducted using the implementation explained in Chapter 4, which is available on GitHub (`https://github.com/Bettvorleger/XF-OPT-META`). The scripts were executed using *Python* version 3.11.0rc1, and the workload was split between two Linux servers, both running Ubuntu 22.04.1 LTS. The first server had 16GiB of system memory and one Intel(R) Xeon(R) Gold 6130 CPU running at a frequency of 2.10GHz on eight available cores, with two threads per core. The second system had 36GiB of memory and two Intel(R) Xeon(R) Gold 6130 CPUs @ 2.10GHz on nine available cores, with two threads per core. When possible, the computation was parallelized natively by Linux or through library support in *Python* (see Section 4.1.1).

5.5 Analysis Procedure

Each of the three data sets was analyzed differently, depending on the objective. These objectives and the methods used to achieve them are explained in the following. The functionality is encompassed by a separate `analyzer` module and accompanying *Python* scripts, both of which are not explained in detail in this thesis, but are also part of *XF-OPT/META* and available in the GitHub repository.

Table 5.4: An excerpt of the optimizer run parameter history $\mathcal{H}(\lambda, f(\lambda))$ for $D\left(A = \text{GP}, p = \text{eil51}, C = 0.25, r_{\text{opt}} = 1\right)$, with an optimal solution of $L^*_{\text{eil51}} = 426$.

n_call	α	β	w_{persbest}	w_{persprev}	$w_{\text{parentbest}}$	θ	H	$f(\lambda)$	RPD
1	0	0	0.001	0.001	0.001	0.1	full	1347.840	2.163
2	10	10	0.763	0.930	0.989	0.416	partial	448.816	0.054
3	8	8	0.763	0.334	0.989	0.416	partial	473.827	0.112
...									
30	0	10	0.001	0.639	0.001	0.5	partial	466.423	0.095

In general, since we have used TSP instances from the *TSPLIB*, we know the optimal solution for each of these instances L^*. Therefore, when referring to the solution quality $f(\lambda) = L$ of a parameter set λ, instead of giving the actual length of the TSP tour L, a relative difference to the optimal solution *RPD* was used, which is defined as follows:

$$RPD = (L - L^*)/L^* \tag{5.3}$$

This makes it easier to evaluate solutions and to compare different problem instances with varying optimal solution.

Furthermore, continuing the explanation of the first two parts of the experiment and their data sets $\mathcal{D}(\mathcal{A}, p, C, r_{\text{opt}}) \rightarrow \{\mathcal{H}(\lambda, f(\lambda)), \lambda^*, \mathcal{M}\}$, the *RPD* value has been applied to all of the solution qualities of the parameter histories $\mathcal{H}(\lambda, f(\lambda))$. To help illustrate further explanations, an excerpt of such a parameter run history is shown in Table 5.4.

5.5.1 Part I - Choosing the Optimization Algorithm

The first part focuses on the ideal HPO method to use. Therefore, the convergence behavior, the resulting solution quality, and the robustness in finding a good solution are subject of this part. For this purpose, a convergence plot was created for each of the five problem instances containing all four optimization methods. This line graph shows the best/minimum current solution quality (y-axis) obtained until each further iteration of objective call (x-axis), one line for each of the three optimization runs, differentiating the four optimization algorithms by color. An additional bold line shows the mean progression averaged along the objective call axis for each algorithm. Equations (5.4) and (5.5) define the formulation for the pairs $\{(x, y) \mid 10 < x \leq \texttt{n\_calls}\}$, where $f(\lambda)_i$ is the solution quality for the parameters acquired during the i-th optimizer iteration of all `n_calls`.

$$y = \min_{i \leq x} f(\lambda)_i \,, \quad f(\lambda)_i \in \mathcal{D}^i_{\mathcal{A},p,C} \tag{5.4}$$

$$\overline{y} = \frac{1}{r_{\text{opt}}} \sum_{i=1}^{r_{\text{opt}}} y_i \tag{5.5}$$

The area under the curve (AUC) and some non-parametric statistical hypothesis tests were also computed for each algorithm and problem. The most common analysis of variance tests have been considered in the selection of these statistical tests and reviewed for application to our experiments. Since the multidimensional HPO process using Bayesian Optimization and especially the underlying decision tree models are non-parametric in nature, standard ANOVA was ruled out from the start. Furthermore, each run of the algorithm is subject to a new random initialization, which affects the chosen path, the dynamics of the problem instances, and the sampling of the hyperparameters during optimization. Although the model built during the optimization procedure chooses each parameter set based on the last iterations, the completely different behavior of a newly initialized H-SPPBO and problem instance makes them not directly comparable. Therefore, each run can be considered as an individual entity and not as part of a series of optimization iterations, making them unpaired or independent of each other. This is important to note before choosing the appropriate tests, and applies to the comparison across all optimization parameters (methods, dynamics, problem instances). Since our comparison of optimization methods is non-parametric, and since we have more than two groups to compare, we can efficiently narrow down the ideal tests. Thus, the Kruskal-Wallis H test, which can be seen as a Mann-Whitney U test for more than two groups, was chosen. It already takes into account the number of groups and does not require a separate correction for Type I error or multiple comparisons. A standard significance level of 0.05 was chosen to reject the null hypothesis.

To get more information about the comparison between the optimization distributions, a post-hoc test was performed in case of H_0 rejection. Under the same assumptions as before, there are several applicable tests after a Kruskal-Wallis H-test, the most popular being Dunn's test. However, the lesser known but supposedly more powerful Conover-Iman test [17] was chosen. This test performs multiple pairwise comparisons of all group members, with each result giving a p-value consistent with the null hypothesis that the samples come from the same distribution. Since multiple tests are performed on the same data set, it is necessary to correct for the Type I error. Using a simple Bonferroni correction, we get more false positive H_0 rejections (smaller p-values), but also fewer false negatives, which is fine in our case since we have several other measures to use as well. Both tests are explained in more detail in the next two subsections.

All of the evaluations mentioned above start after iteration 10, because these first iterations are randomly sampled and do not reflect the model/algorithm behavior.

Kruskal–Wallis H Test

This statistical method is used to determine whether multiple samples come from the same underlying distribution. Given a number of N total observations divided into k samples of possibly different sizes n_i ($1 \leq i \leq k$), the null hypothesis H_0 can be formulated, that "all of the k population distribution functions are identical" [16] and the alternative H_1 under rejection of H_0 at significance level 0.05 is, that at least one distribution differs considerably from the others. In our case, we want to test, if one particular parameter search behavior of the optimization methods performs significantly different from the others. Due to the interlaced sampling procedure of HPO, the test assumption of completely random acquired samples can only be partially ensured here. The test statistic T is defined as follows:

$$T = \frac{1}{S^2}\left(\sum_{i=1}^{k}\frac{R_i^2}{n_i} - \frac{N(N+1)^2}{4}\right) \tag{5.6}$$

$$S^2 = \frac{1}{N-1}\left(\sum R(X_{ij})^2 - \frac{N(N+1)^2}{4}\right)$$

where X_{ij} denotes the j-th entry from the i-th sample of all k samples, and $R(\cdot)$ a rank function, mapping an integer value between $[1, N]$ to all ordered samples ignoring the sample groups from smallest ($R = 1$) to largest ($R = N$) sample value. Lastly, let R_i be the sum of all ranks for the i-th sample. The significance (p-value) can then be acquired through a table or through a software package. The thesis is using the *SciPy* implementation of the Kruskal–Wallis Test, which approximates its p-value through a χ^2 distribution.

Conover–Iman Test

Conover and Iman [17] proposed a squared ranks test for comparing the variances of multiple samples. Except for allowing the means of the distributions to differ, the Conover-Iman test uses a H_0 hypothesis similar to the Kruskal–Wallis test, that all k populations are identically distributed. If H_0 could be rejected for a particular paring of samples X and Y, the statement of the alternative hypothesis H_1 depends on the focus of the test: A two-tailed test asserts, that the variances of the two samples are not equal ($Var(X) \neq Var(Y)$), where a lower- or upper-tailed test claims that one of the sample's variance is smaller/larger than the other one ($Var(X) \lessgtr Var(Y)$). Before the test statistic can be calculated, the samples X_{ij} are normalized by subtracting the population mean from each observation ($Z_{ij} = |X_{ij} - \overline{X}_i|$) and then the rank R_{ij} is calculated as explained with the Kruskal–Wallis test. The statistical value T can now be defined as follows [16, Chapter 5.3]:

$$T = \frac{1}{D^2}\left(\sum_{i=1}^{k} \frac{S_i^2}{n_i} - N \cdot \overline{S}^2\right) \tag{5.7}$$

where

$$S_i = \frac{1}{N}\sum_{j=1}^{n_i} R_{ij}^2\,, \quad \overline{S} = \frac{1}{N}\sum_{i=1}^{k} S_i$$

and

$$D^2 = \frac{1}{N-1}\left(\sum R_i^4 - N \cdot \overline{S}^2\right)$$

This test was performed using the *scikit-posthoc* Python package [94]. In addition, it was slightly modified to not use the absolute value of the normalized sample values. This allows an easy realization of a lower-/upper-tailed Conover-Iman test, since the sign is explicitly needed for this and was not provided in the aforementioned implementation. Thus, we can determine, whether the distribution of a particular HPO method is lower than the others, which in turn, has some expressiveness when it comes to faster convergence to better solutions.

5.5.2 Part II - Choosing the Parameter Sets

The second part focuses on the differences (and similarities) between the optimal parameters for each problem category. Box plots and scatterplot matrices were generated for all subcategories of parameters to analyze how they behave on their respective problem instances and dynamics, but also how they influence each other. For this last aspect in particular, the surrogate model was used to create a so-called "partial dependence"plot. These plots show how each hyperparameter, or for a two-dimensional contour plot, a combination of two hyperparameters, affects the prediction of the surrogate model. This is done by effectively calculating the following expected value for a set of parameters of interest $\mathbf{\Lambda}_S$ and their complement $\mathbf{\Lambda}_C$ over the entire space of parameters $\mathbf{\Lambda}$ [44, Chapter 10.13.2]:

$$\hat{f}_S(\mathbf{\Lambda}_S) = \mathbb{E}_{\mathbf{\Lambda}_C}\left[f(\mathbf{\Lambda}_S, \mathbf{\Lambda}_C)\right] = \int \hat{f}(\mathbf{\Lambda}_S, \mathbf{\Lambda}_C) P(\mathbf{\Lambda}_C) d\mathbf{\Lambda}_S \tag{5.8}$$

where $\hat{f}(\mathbf{\Lambda}_S, \mathbf{\Lambda}_C)$ is the response function of the model for the given hyperparameters. This is done computationally by fixing the parameters $\mathbf{\Lambda}_S$ at regular intervals and then averaging the objective value over a number of random samples. However, since the surrogate model is already an approximation of the actual objective function (i.e. the H-SPPBO algorithm), and the sampling process consists of 250 random points, the resulting partial dependence model is only a rough estimate. Nevertheless, it provides insight into the hyperparameter correlations and influences [33].

Another aspect evaluated during this part of the analysis is the feature importance. Given that we chose one of the two regression tree-based surrogate models (ET or GBRT), we can use the underlying `scikit-learn` library to obtain the feature importance. For both methods, this was realized by calculating the Mean Decrease in Impurity (MDI), which defines the importance of each parameter by counting how often it was used to split a node in the decision tree, weighted by the resulting samples left on the branches. However, this procedure overestimates high cardinality parameters, i.e. parameters with many unique values, which is considered in the later discussion [91].

5.5.3 Part III - Evaluating the Parameter Sets

The third part focuses on the validation of the parameters. While its methods are those of a metaheuristic analysis inspired by the run plots and metrics of [58], it also tries to answer the general question of the thesis, whether HPO methods are suitable for metaheuristics (or at least H-SPPBO), and whether generalization from smaller to larger problem instances within the same group was successful. In addition, to better analyze the dynamic capabilities of the H-SPPBO algorithm, the precision and recall metrics, along with receiver operating characteristic (ROC) curves, were generated to evaluate the accuracy of the change detection mechanism.

6 Results and Evaluation

All experiments and analyses were performed as previously explained in Sections 5.4 and 5.5.

6.1 Part I - Choosing the Optimization Algorithm

The results of the first experimentation part are very insightful and allow for a sound analysis of the most appropriate optimization algorithm to tune the parameters of the H-SPPBO algorithm. Note that an evaluation of the quality of the solution itself and of the performance of the H-SPPBO algorithm is not discussed in this first part. The solution quality is only used to compare the optimization methods.

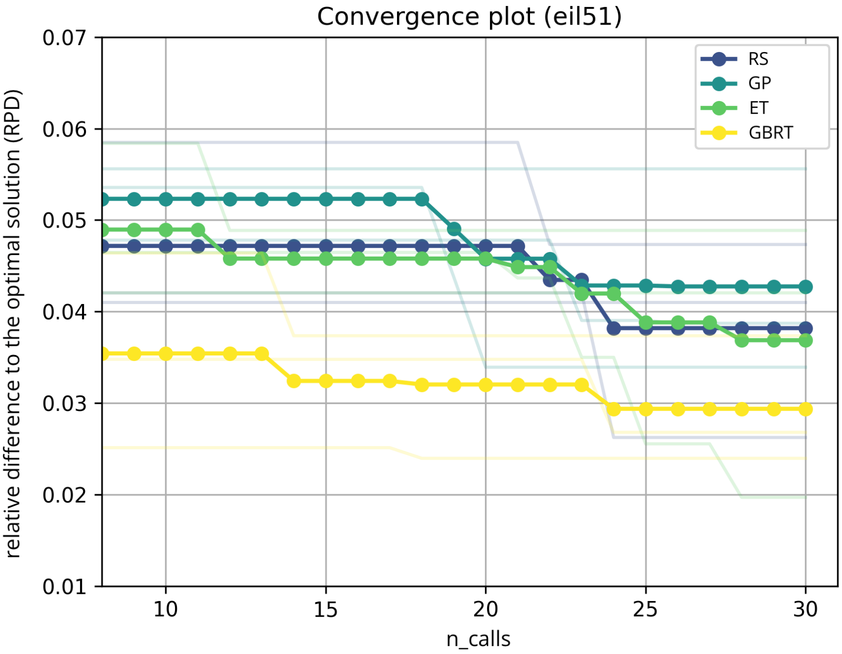


Figure 6.1: Convergence plot of TSP instance `eil51`, comparing four different optimization algorithms.

First, the convergence plots of all five individual instances and their average are discussed. As explained in Section 5.5.1, these graphs show the minimum of the relative difference to the optimal solution *RPD* up to each objective function call (`n_call`). Figure 6.1 shows this plot for the `eil51` TSP instance. The bold, yellow line, which shows the average of the three GBRT runs, indicates, that this algorithm found the best solution over the course of the 30 objective calls. Except for a single run, where ET performed admirably, achieving a 2% difference from the optimal solution, none of the other algorithms were able to come close to the GBRT. However, the average convergence behavior of all four algorithms seems to be very similar, with moderate improvements between objective calls 17 and 25. This behavior suggests, that the first 10 sampling calls were sufficient to obtain a decent parameter configuration for the `eil51` instance. The specific runs (light colored lines) show, that RS and ET fluctuated the most in terms of initial solution quality and improvement over time. Although GP produced the worst solutions out of all four algorithms, it still managed to achieve an average solution quality of almost 4% after the full 30 `n_calls`.

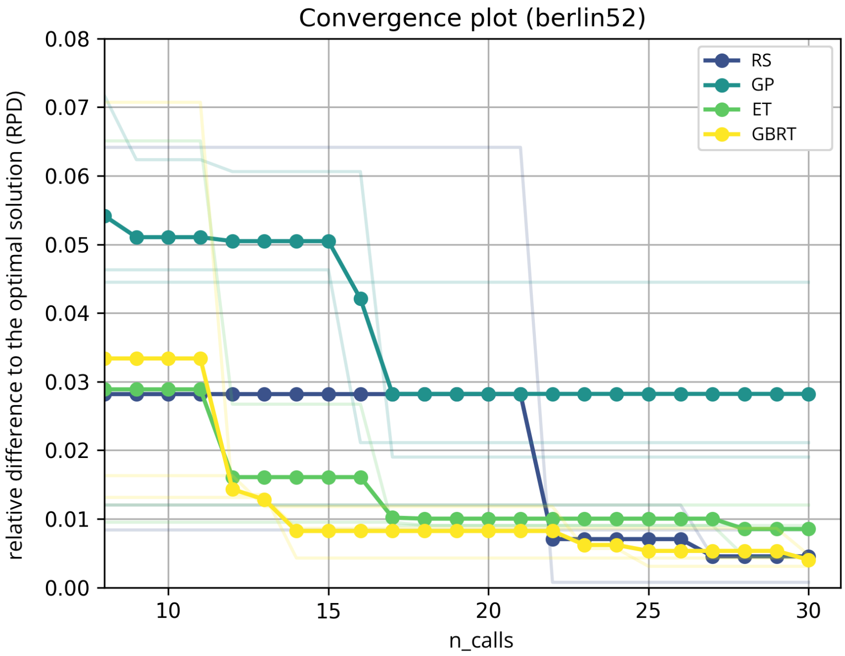


Figure 6.2: Convergence plot of TSP instance `berlin52`, comparing four different optimization algorithms.

The next instance is `berlin52`, whose convergence plot is shown in Figure 6.2. Here, GP continues to be the least well performing HPO method, with an average *RPD* of only about 3%, with one particular run reaching as low as 4.5%. All other methods achieved at least 2% lower *RPD* values, with RS and GBRT having an almost identical final *RPD* of 0.5%. However, GP was able to improve its solution by a significant 2% over the course of seven additional objective calls, suggesting that the algorithm was quickly making use of the now utilized, underlying model. Due to the very random nature of RS, some other individual runs started with a very high $RPD = 6.5\%$ and then, improved very abruptly after about 21 objective calls. The convergence behavior of RF and GBRT is quite similar and shows, that both started with a good solution already at about 3% for the tenth objective call, and then gradually improved up to objective call 17, analogous to GP, but with less relative improvement. After that, only small gains in solution quality were achieved.

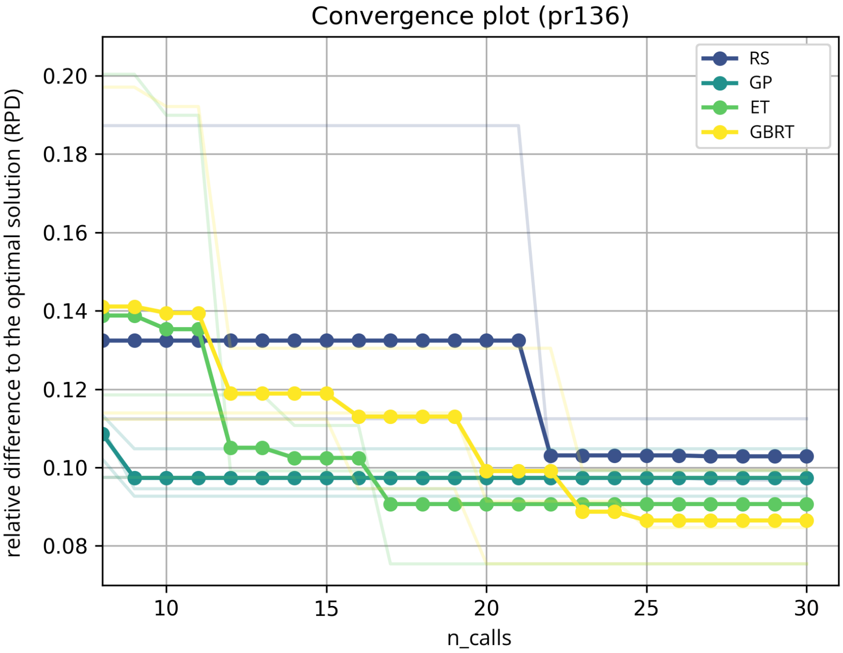


Figure 6.3: Convergence plot of TSP instance pr136, comparing four different optimization algorithms.

The pr136 TSP instance, presented in Figure 6.3, continues the trend of GBRT performing the best of the four optimization algorithms, albeit with a small lead of less than 0.5% over the next best method, ET. Interestingly, GP started with a considerably better initial sample at $RPD = 11\%$ after 10 objective calls, which gave it a head start, but also caused the algorithm to stagnate over the course of the remaining 20 calls. This might be due to the fact that Hammersley sampling was chosen only for this particular algorithm, or it could just be another random influence. ET and GBRT gradually improved the most out of the four methods, with steep drops of around 2% or more by the 23rd objective call. RS rarely saw any meaningful improvement after the initial sampling, and the huge drop in average RPD was due only to a single run that remained at a relatively poor value of $RPD = 18.8\%$ up until the 21st `n_call`.

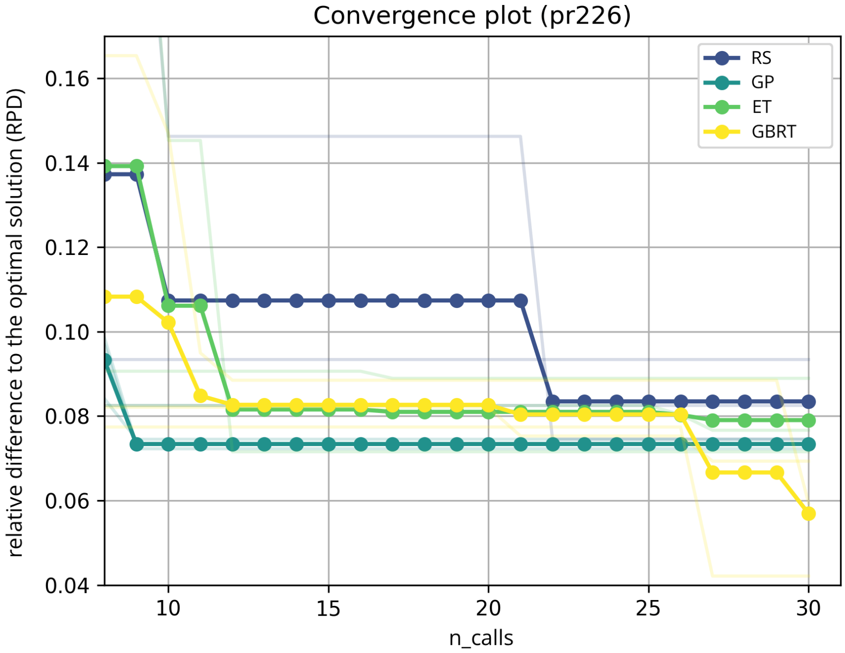


Figure 6.4: Convergence plot of TSP instance `pr226`, comparing four different optimization algorithms.

The convergence plot in Figure 6.4 for instance `pr226` again has RS as the weakest method, with only small gains of about 2% on objective call 22, bringing it to the average *RPD* of the other algorithms. Some other single runs performed even worse. Contrary to the good performance of the other instances, this time ET starts its model training at 10 objective calls with a *RPD* value almost the same as RS, and only manages to significantly improve its parameters two `n_calls` later. It then shows stagnant behavior at *RPD* = 8%. Only in the last five objective calls GBRT showed a similar convergence behavior, improving by more than 2%, giving it the lead over GP. This suggests that with larger instance dimensions and thus potentially more complex solutions, GBRT benefits from more objective calls that improve the underlying model. As with `pr136`, the initial sample of the Hammersley method gave GP a lead in solution quality, but then stagnated after the ninth call and was unable to improve its *RPD* value of about 7.5%.

The last TSP instance, `d198`, confirms the observations of the previous instances, but this time all methods except RS are within very close proximity to each other after the 24th objective call. Although GBRT takes the lead in final relative solution quality with just under 9%, it is not by much. Together with GP, both methods managed to achieve considerable improvements up until the 15th objective call, after which they more or less stagnated. ET started with a similarly sub-optimal *RPD* as RS at objective call 10, but

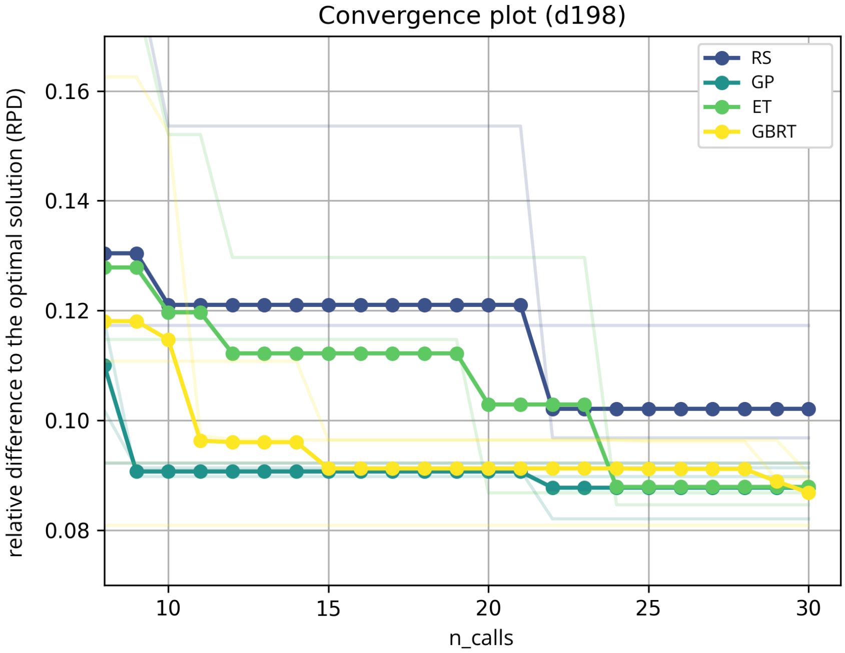


Figure 6.5: Convergence plot of TSP instance `d198`, comparing four different optimization algorithms.

was able to improve steadily over the next 15 `n_calls`. As expected at this point, RS only managed to meaningfully improve in one run at objective call 21, still placing its final *RPD* more than 1% above the rest of the methods.

Lastly, the average convergence plot aggregates the runs of all five instances. Since GBRT is the best performing method out of all the individual comparisons, its average convergence behavior emphasizes this by giving it a lead in final *RPD* of about 1.5% over the next best method, ET. Both of the show similar convergence behavior over the entire runtime, with large improvements in the first few objective calls, and small, but steady (almost linear) improvements thereafter. The behavior of GP is not very different, and is mainly distinguished by its better initial *RPD* value, which reinforces the proposed relationship with the Hammersley sampling method. After this initial lead, the algorithm almost stagnates, and after the 22nd objective call, it converges to the trajectory of RS.

Regarding the visually observable convergence behavior of the four optimization methods, we can also look at the area under the curve (AUC) of the plots and the minimum/best relative solution quality obtained, i.e. the *RPD* value at `n_call` = 30, hereafter called RPD_{min}. This data is presented in Table 6.1 for all five instances, their mean, and is made up of all available optimizer runs. In this context, a low AUC value would indicate that the optimization algorithm found a well-performing parameter set, and therefore a low *RPD*, in a short time, i.e., few objective calls. However, the comparatively lowest AUC does

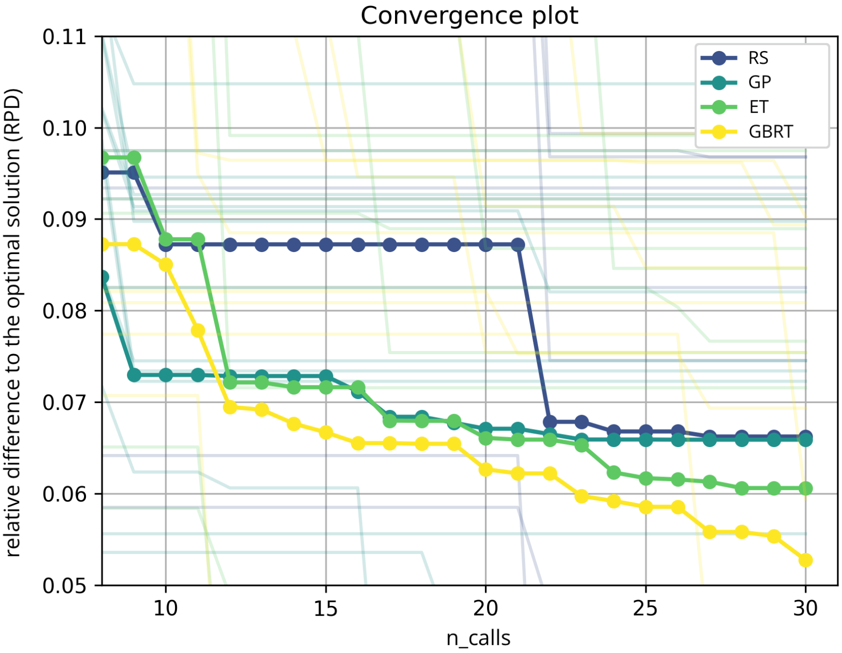


Figure 6.6: Convergence plot of the average over five TSP instances `eil51, berlin52, pr136, pr226, d198`, comparing four different optimization algorithms.

not necessarily mean that it also found the lowest RPD_{min} value among all algorithms. Thus, both values are discussed for each instance to determine how fast each algorithm converged to its best solution. For the `eil51` TSP instance, GBRT has by far the lowest AUC of 0.601. Paired with the second best $RPD_{min} = 2.4\%$, this method continues its favorable performance from the convergence plots. With a 0.4% improvement in RPD_{min} over GBRT, ET was the second fastest to convergence to this final solution. GP delivered the worst performance in this instance.

The `berlin52` instance implies a similar behavior, with GBRT leading in convergence speed via the lowest AUC. However, as discussed in the corresponding convergence plot, RS managed to find the best solution out of the four methods, with $RPD_{min} = 0.1\%$, outperforming GBRT by 0.2%. Again, GP shows the worst performance, this time with an AUC of 0.650, almost twice as high as its direct competitor RS with 0.347. ET is again the second fastest converging algorithm after GBRT, with an almost equally good $RPD_{min} = 0.5\%$.

The results for `pr136` show an equal best $RPD_{min} = 7.5\%$ for GBRT and ET, and an almost equal AUC for GP and ET. Therefore, in this instance, ET outperforms GBRT as the best converging algorithm. Interestingly, even GP achieves a better AUC of 1.850 than GBRT, albeit with a worse minimum relative solution quality, which is more in line with RS.

Table 6.1: The AUC and minimum relative solution quality at the last objective call (RPD_{min}) of all optimization runs for each method and instance, and for the mean over all instances.

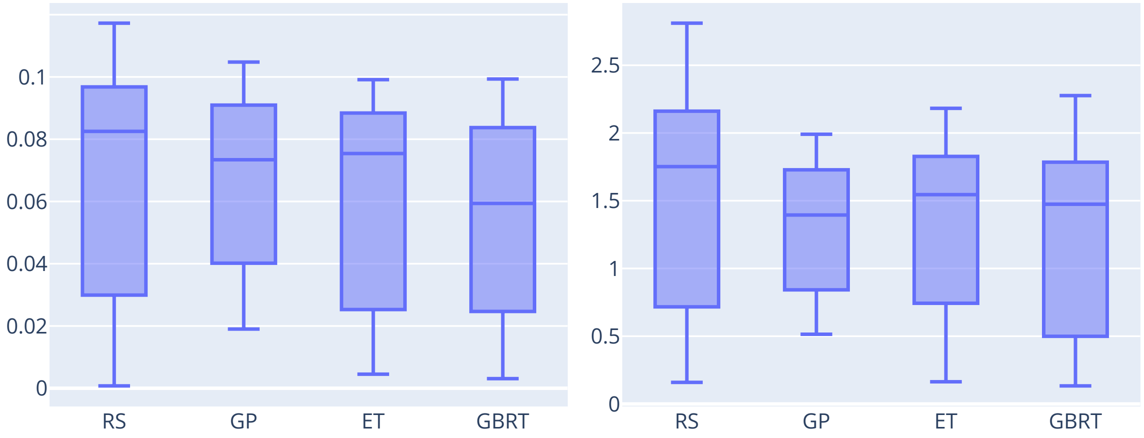
	eil51		berlin52		pr136	
	AUC	RPD_{min}	AUC	RPD_{min}	AUC	RPD_{min}
RS	0.830	0.026	0.347	0.001	2.266	0.097
GP	0.900	0.034	0.650	0.019	1.850	0.093
ET	0.819	0.020	0.227	0.005	1.809	0.075
GBRT	0.601	0.024	0.159	0.003	1.948	0.075

	pr226		d198		mean	
	AUC	RPD_{min}	AUC	RPD_{min}	AUC	RPD_{min}
RS	1.837	0.075	2.139	0.092	1.484	0.058
GP	1.394	0.072	1.698	0.082	1.298	0.060
ET	1.547	0.072	1.940	0.085	1.268	0.051
GBRT	1.497	0.042	1.745	0.081	1.190	0.045

pr226, the largest TSP instance out of the five tested in this part of the experimentation discussion, continues the improvement of GP in terms of AUC, resulting in the best value out of all four algorithms, with a solid 7% difference compared to the next best algorithm, GBRT, which itself, was able to achieve the best RPD_{min} at 4.2%. RS continues to perform worse as the instance dimension increases, which could be an opposite trend for the convergence speed of GP.

The last TSP instance, d198, confirms this implication, resulting in GP again being the best algorithm in terms of AUC, and the second best in terms of RPD_{min} at 8.2%. This suggests, that this algorithm could benefit from larger, more complex instances, perhaps even from stronger clustering, as the latter two instances were categorized. GBRT achieved the best RPD_{min} value at 8.1% and the second best AUC, which is only 2.7% higher than the value of GP. In this instance, the performance of ET was comparatively underwhelming, being closer to the AUC value of RS than to the other two methods.

Finally, the mean results confirms, that GBRT was the algorithm that converged the fastest ($AUC = 1.19$) to the best performing parameter set ($RPD_{min} = 4.5\%$). Even in later, higher dimensional instances, it performed at least the second best or better, implying good search consistency across the parameter space. ET obtained better solutions at an earlier time for the three smaller instances, resulting in lower averages compared to GP, which showed the opposite trend, improving with higher problem dimension.



(a) Relative difference to optimal solution quality at last the objective call (RPD_{min}) for all runs and instances.

(b) Area under the curve (AUC) of solution quality convergence graph.

Figure 6.7: Statistical box plots illustrating the data of all five individual convergence plots and their individual runs.

Figure 6.7 shows statistical box plots for both of the metrics from the table over all runs and not grouped by instance. This visualization is also used to evaluate the consistency across all four optimization methods, which was previously only implied by their average or best-case performance. However, this is not normalized by instance, so the absolute values and differences, as with the mean from the previous table, might not be as expressive. Nevertheless, since all four algorithms are affected by this bias, qualitative statements are still possible.

The dispersion of RPD_{min} in Figure 6.7a implies that GP reached its solution most consistently at around its median of 7.5%, but also never reached the same minimum as the other three algorithms. RS, on the other hand, was the most unreliable method, with its maximum and minimum being far of from the upper and lower quartiles, respectively. It also has the highest median at about 8%. GBRT and ET share similarities in their interquartile range (IQR) and minimum/maximum values, with GBRT having a slightly smaller dispersion. However, they differ tremendously in their median, with ET having a value almost 2% higher than GBRT, which means that although both their solution consistency is not ideal, GBRT still obtained the best final solution out of all algorithms.

The box plots for the AUC (Figure 6.7b) show similar consistency behavior. GP is again the most reliable algorithm when it comes to converging to its final solution, but again it cannot provide the same best-case performance (minimums) as the other three algorithms.

However, its median is the lowest, with GBRT having the second lowest value, but also a significantly higher IQR than GP. Lastly, RS shows the most inconsistent behavior with the highest median.

6.1.1 Statistical Tests

To validate all of the above results, two statistical tests were also performed on the optimization data. The first test was the Kruskal–Wallis H test, explained in section Section 5.5.1, with each of the four optimization algorithms as a sample group, and separated by the five instances used. The results for each instance are shown in Table 6.2. A common significance level of 0.05 was chosen to reject the null hypothesis. The data from the table shows, that for each of the five instances, we can definitely reject the null hypothesis by looking at the p-values, thus suggesting that the four optimization algorithms show a significantly different convergence behavior from each other.

Table 6.2: Results of the Kruskal–Wallis H test for all optimization run data of the first part, with each of the four optimization algorithms as a sample group, separated by TSP instance.

	eil51	berlin52	pr136	pr226	d198
statistic	47.893	46.960	29.898	57.998	46.514
p-value	2.244×10^{-10}	3.544×10^{-10}	1.450×10^{-6}	1.574×10^{-12}	4.410×10^{-10}

A post-hoc Conover–Iman test was then performed to gain more insight into which specific pairs of optimization algorithms differ and how. As explained in Section 5.5.1, a one-sided test was used, resulting in two tables - one for the statistic value (Table 6.4) and one for the p-value (Table 6.3). Note that this table format was preferred over a symmetric matrix for each problem instance, with the columns and rows containing all five methods each. As presented here, redundant information can be excluded by directly pairing the algorithms, resulting in $\binom{4}{2} = 6$ pairs (columns) for each TSP instance (rows).

Starting with the p-value to reject the null hypothesis, we can then look at the statistic value, more specifically at the sign, to formulate the alternative hypothesis, i.e. to infer how the distribution of the convergence behavior for the algorithms differ. A positive value implies that in this particular pairwise comparison (X vs. Y), the first stated algorithm (X) has a distribution that is significantly above the others, suggesting that either the convergence speed, the solutions obtained, or both, are also worse than the second one (Y). Furthermore, in both tables the cells are marked green, whenever the p-value is lower than 0.05 (the significance level) and the null hypothesis can be rejected for that combination of TSP instance and optimization method pairing.

Table 6.3: The p-value of the Conover–Iman test for all optimization run data of the first part, with each of the four optimization algorithms as a sample group, separated by TSP instance. The cells are marked green, whenever the p-value is lower than the significance level of 0.05.

TSP	RS vs. GP	RS vs. ET	RS vs. GBRT	GP vs. ET	GP vs. GBRT	ET vs. GBRT
eil51	0.626	1.000	1.361×10^{-11}	0.045	9.879×10^{-15}	1.710×10^{-9}
berlin52	6.318×10^{-10}	1.000	0.056	4.844×10^{-8}	5.094×10^{-15}	0.003
pr136	4.407×10^{-6}	1.473×10^{-7}	1.491×10^{-4}	1.000	1.000	0.525
pr226	1.077×10^{-22}	1.851×10^{-10}	1.839×10^{-11}	2.229×10^{-8}	2.069×10^{-7}	1.000
d198	4.038×10^{-15}	0.001	2.656×10^{-7}	1.201×10^{-7}	0.001	0.210

Table 6.4: The statistical value of the Conover–Iman test for all optimization run data of the first part, with each of the four optimization algorithms as a sample group, separated by TSP instance. The cells are marked green, whenever the p-value is lower than the significance level of 0.05.

TSP	RS vs. GP	RS vs. ET	RS vs. GBRT	GP vs. ET	GP vs. GBRT	ET vs. GBRT
eil51	−1.644	1.099	8.358	2.743	10.002	7.259
berlin52	−7.486	−1.002	2.668	6.485	10.154	3.669
pr136	5.400	6.223	4.491	0.823	−0.909	−1.731
pr226	14.432	7.766	8.290	−6.666	−6.142	0.524
d198	10.207	3.936	6.083	−6.271	−4.125	2.146

The data from both tables show, that the distribution of the GBRT method is below all other algorithms for the `eil51` instance, although the null hypothesis in the pairing with ET was only slightly below the significance level. Furthermore, the distribution of ET was also below that of GP, confirming the visual inference from the convergence plot. In the data for the `berlin52` instance, GBRT is below GP and ET, but interestingly, fails to reject the null hypothesis when compared to RS, confirming the unrepresentatively good performance of this algorithm for this instance. Also, the convergence distribution of GP is above all other three algorithms by comparison, which is also shown in Figure 6.2. For the next three instances, `pr136`, `pr226`, and `d198`, the distribution for RS lies above all other three algorithms, verifying the underwhelming performance described in previous discussion. For the two latter instances, we can also determine that the convergence behavior of GP is above the distribution of both ET and GBRT.

In summary, RS could only establish itself as having a favorable distribution only once, ET managed to have a lower convergence behavior five times, GP could undercut in its direct pairing seven times, and GBRT was the optimization algorithm that fell below its comparison distribution the most with 10 times. All of this confirms the favorable position of GBRT, but also strengthens the case for GP over ET.

6.1.2 Conclusion

As expected, Random Search (RS) proved to be the worst performing algorithm under these test conditions. Although each of the other three methods was able to reliably outperform its solutions, usually with fewer objective calls, it still produced satisfactory solutions that differed from the rest by only a few percent. Therefore, its aforementioned position as a “baseline” HPO method is more than justified.

The Gradient Boosted Regression Trees (GBRT) algorithm shows the fastest convergence with the best solution quality, while providing the second most robust results. Therefore, it is used to perform the second part of the testing procedure. Another advantage, besides its good performance, is its ability to provide a trained machine learning model from which the parameter importance and other qualitative statements about its prediction can be derived.

However, the theoretical second choice is not so clear, since both Gaussian process (GP) and Extra-Trees (ET), performed very well in certain instances. As suggested in some earlier discussions, ET seems to perform its best on TSP instances with less than 100 city nodes, while GP improves its convergence and solutions significantly on TSP instances with dimensions above 150. Another factor may be the city placement characteristic established in Section 5.1. However, confirming this possible relationship would require a different experimental setup, whereas the data from this first part would not be sufficient. Therefore, both of these algorithms should be considered as potential candidates for future tests, especially with varying problem dimensions and complexity.

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] S. AI. *MPIRE (MultiProcessing Is Really Easy)*. `https://github.com/Slimmer-AI/mpire`. 2023 (cit. on p. 47).

[2] 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).

[3] D. Arthur, S. Vassilvitskii. *k-means++: The advantages of careful seeding*. Tech. rep. Stanford, 2006 (cit. on pp. 61, 62).

[4] J. Bergstra, Y. Bengio. “Random search for hyper-parameter optimization.” In: *Journal of machine learning research* 13.2 (2012) (cit. on p. 33).

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

[6] 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 p. 17).

[7] D. G. Bonett. “Confidence interval for a coefficient of quartile variation”. In: *Computational statistics & data analysis* 50.11 (2006), pp. 2953–2957 (cit. on p. 59).

[8] I. Boussaid, J. Lepagnot, P. Siarry. “A survey on optimization metaheuristics”. In: *Information sciences* 237 (2013), pp. 82–117 (cit. on p. 15).

[9] L. Breiman. “Bagging predictors”. In: *Machine learning* 24 (1996), pp. 123–140 (cit. on p. 39).

[10] L. Breiman. “Random forests”. In: *Machine learning* 45 (2001), pp. 5–32 (cit. on p. 38).

[11] 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. 34, 37, 38).

[12] 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).

[13] C. Cervellera, D. Macciò. “Learning with kernel smoothing models and low-discrepancy sampling”. In: *IEEE transactions on neural networks and learning systems* 24.3 (2013), pp. 504–509 (cit. on p. 66).

[14] X. Chen. *treelib*. `https://github.com/caesar0301/treelib`. 2018 (cit. on p. 48).

[15] P. J. Clark, F. C. Evans. “Distance to nearest neighbor as a measure of spatial relationships in populations”. In: *Ecology* 35.4 (1954), pp. 445–453 (cit. on pp. 49, 59).

[16] W. J. Conover. “Practical nonparametric statistics”. In: Wiley, 1999. Chap. 5.3 (cit. on pp. 76, 77).

[17] W. J. Conover, R. L. Iman. “On multiple-comparisons procedures”. In: *Los Alamos Sci. Lab. Tech. Rep. LA-7677-MS* 1 (1979), p. 14 (cit. on pp. 75, 76).

[18] G. C. Crişan, E. Nechita, D. Simian. “On Randomness and Structure in Euclidean TSP Instances: A Study With Heuristic Methods”. In: *IEEE Access* 9 (2021), pp. 5312–5331 (cit. on pp. 49, 60).

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

[20] A. Cutler, D. R. Cutler, J. R. Stevens. “Random forests”. In: *Ensemble machine learning: Methods and applications* (2012), pp. 157–175 (cit. on pp. 38, 39).

[21] D. Cvetković, Z. Dražić, V. Kovačević-Vujčić, M. Čangalović. “THE TRAVELING SALESMAN PROBLEM”. In: *Bulletin (Académie serbe des sciences et des arts. Classe des sciences mathématiques et naturelles. Sciences mathématiques)* 43 (2018), pp. 17–26 (cit. on pp. 60, 61).

[22] 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. 41).

[23] 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).

[24] 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).

[25] 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, 18, 20, 68).

[26] M. Dorigo, T. Stützle. *Ant Colony Optimization.* The MIT Press, June 2004. DOI: 10.7551/mitpress/1290.001.0001 (cit. on pp. 10, 18).

[27] M. Dorigo, T. Stützle. *Ant colony optimization: overview and recent advances.* Springer, 2019 (cit. on p. 19).

[28] M. Dry, K. Preiss, J. Wagemans. "Clustering, randomness and regularity: Spatial distributions and human performance on the traveling salesperson problem and minimum spanning tree problem". In: *The Journal of Problem Solving* 4.1 (2012), pp. 1–17 (cit. on pp. 49, 59, 60).

[29] 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, 29, 30).

[30] 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).

[31] 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).

[32] M. Feurer, F. Hutter. "Hyperparameter optimization". In: *Automated machine learning: Methods, systems, challenges* (2019), pp. 3–33 (cit. on pp. 7, 31–33, 35, 38).

[33] J. H. Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232 (cit. on pp. 40, 78).

[34] J. H. Friedman. "Stochastic gradient boosting". In: *Computational statistics & data analysis* 38.4 (2002), pp. 367–378 (cit. on p. 40).

[35] 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).

[36] M. Gendreau, J.-Y. Potvin, et al. *Handbook of metaheuristics.* Vol. 2. Springer, 2010 (cit. on p. 15).

[37] P. Geurts, D. Ernst, L. Wehenkel. "Extremely randomized trees". In: *Machine learning* 63 (2006), pp. 3–42 (cit. on p. 39).

[38] 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. 18).

[39] 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 p. 21).

[40] 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).

[41] A. Hagberg, P. Swart, D. S Chult. *Exploring network structure, dynamics, and function using NetworkX.* Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008 (cit. on p. 49).

[42] 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).

[43] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2` (cit. on pp. 47, 49, 58).

[44] 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 pp. 41, 77).

[45] T. Head. *Comparing surrogate models.* 2016. URL: `https://scikit-optimize.github.io/stable/auto_examples/strategy-comparison.html#comparing-surrogate-models` (visited on 03/23/2023) (cit. on p. 71).

[46] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, I. Shcherbatyi. “scikit-optimize/scikit-optimize: v0. 8.1”. In: *Zenodo* (2020) (cit. on p. 50).

[47] 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. 39).

[48] S. Hougardy, X. Zhong. “Hard to solve instances of the euclidean traveling salesman problem”. In: *Mathematical Programming Computation* 13 (2021), pp. 51–74 (cit. on p. 60).

[49] 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).

[50] 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).

[51] 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, 22, 23).

[52] 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).

[53] 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 pp. 9, 67).

[54] 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).

[55] 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. 37).

[56] 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).

[57] 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. 22).

[58] 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, 26, 29, 46, 49, 57, 67, 68, 73, 78).

[59] 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. 37).

[60] 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).

[61] 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).

[62] 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. 10).

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

[64] 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).

[65] 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, 24, 25, 69).

[66] S. Lloyd. "Least squares quantization in PCM". In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137 (cit. on p. 61).

[67] L. Lovász. "1 Background from linear algebra". In: (2007) (cit. on pp. 60, 61).

[68] 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).

[69] 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).

[70] 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).

[71] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, M. A. Aivazis. "Building a framework for predictive science". In: *arXiv preprint arXiv:1202.1056* (2012) (cit. on p. 51).

[72] 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. 41).

[73] 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).

[74] 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).

[75] 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).

[76] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 50).

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

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

[79] 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).

[80] 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).

[81] G. Reinelt. “TSPLIB—A traveling salesman problem library”. In: *ORSA journal on computing* 3.4 (1991), pp. 376–384 (cit. on pp. 5, 13, 49, 57).

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

[83] 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).

[84] 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).

[85] 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).

[86] 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).

[87] 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).

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

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

[90] M. L. Stein. *Interpolation of spatial data: some theory for kriging.* Springer Science & Business Media, 1999 (cit. on p. 38).

[91] C. Strobl, A.-L. Boulesteix, A. Zeileis, T. Hothorn. “Bias in random forest variable importance measures: Illustrations, sources and a solution”. In: *BMC bioinformatics* 8.1 (2007), pp. 1–21 (cit. on p. 78).

[92] 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, 29, 68).

[93] E.-G. Talbi. *Metaheuristics: from design to implementation.* John Wiley & Sons, 2009 (cit. on pp. 10, 16–20, 22, 29, 32).

[94] M. Terpilowski. “scikit-posthocs: Pairwise multiple comparison tests in Python”. In: *The Journal of Open Source Software* 4.36 (2019), p. 1169. DOI: `10.21105/joss.01169` (cit. on p. 77).

[95] 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).

[96] 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 pp. 10, 68).

[97] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2 (cit. on p. 60).

[98] 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).

[99] W. P. Vogt, B. Johnson. *Dictionary of statistics methodology: A nontechnical guide for the social sciences.* Sage, 2011 (cit. on p. 67).

[100] C. K. Williams, C. E. Rasmussen. *Gaussian processes for machine learning.* Vol. 2. 3. MIT press Cambridge, MA, 2006 (cit. on pp. 34, 35, 38).

[101] 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).

[102] 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 pp. 10, 68).

[103] 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, 31–33, 35, 50).

[104] 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. 24).

[105] 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. 24).

[106] E. Yin, K. Wijk. *Bayesian Parameter Tuning of the Ant Colony Optimization Algorithm: Applied to the Asymmetric Traveling Salesman Problem.* 2021 (cit. on pp. 11, 64, 66).

Acronyms

ACO Ant Colony Optimization. 9, 18

ANN artificial neural network. 11

AUC area under the curve. 75

BO Bayesian Optimization. 11, 32, 45

CDF cumulative distribution function. 37

CQV coefficient of quartile variation. 49

DoE Design of Experiment. 11, 30

DTSP Dynamic Traveling Salesperson Problem. 6, 15

EA Evolutionary Algorithm. 18

EI expected improvement. 37

ET Extra-Trees. 39

GA Genetic Algorithm. 5

GBRT Gradient Boosted Regression Trees. 35

GP Gaussian process. 35

GS Grid Search. 33

HPO Hyperparameter Optimization. 7, 31, 45

H-PSO Hierarchical Particle Swarm Optimization. 6, 23

H-SPPBO Hierarchical Simple Probabilistic Population-Based Optimization. 6, 26, 45

IQR interquartile range. 86

LCB ower confidence bound. 66

MDI Mean Decrease in Impurity. 78

ML machine learning. 5, 31

PDF probability density function. 37

PI probability of improvement. 37

P-metaheuristic Population-Based Metaheuristic. 17

PACO Population-Based Ant Colony Optimization. 6, 21

PSO Particle Swarm Optimization. 5, 22

QAP Quadratic Assignment Problem. 5, 26

RF Random Forests. 35

ROC receiver operating characteristic. 78

RS Random Search. 33

SA Simulated Annealing. 17

S-metaheuristic Single-Solution Based Metaheuristic. 16

SCE Solution Creating Entity. 6, 25

SPPBO Simple Probabilistic Population-Based Optimization. 5, 24

SSO Simplified Swarm Optimization. 6, 24

TS Tabu Search. 17

TSP Traveling Salesperson Problem. 5, 13

XF-OPT/META EXperimentation Framework and (Hyper-)Parameter Optimization for Metaheuristics. 52

List of Figures

List of Tables

List of Listings

List of Algorithms

A Additional Figures

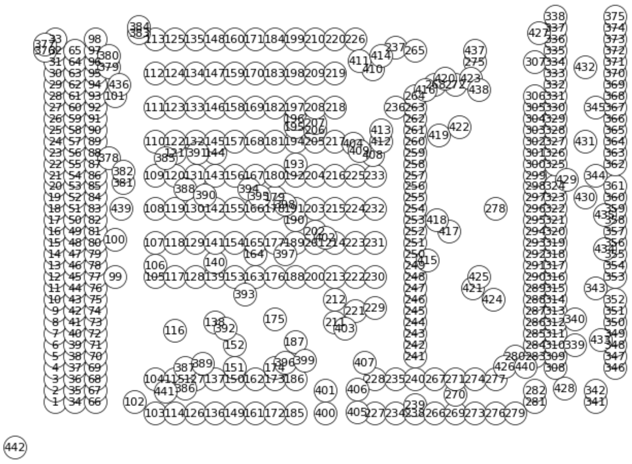


Figure A.1: Visualization of the TSP instance *pcb442*.

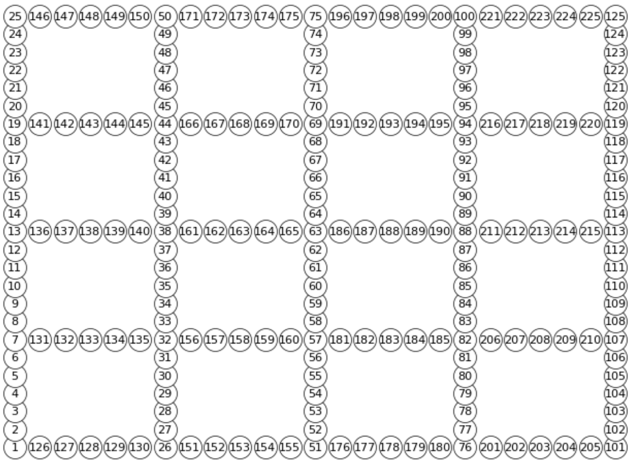


Figure A.2: Visualization of the TSP instance *ts225*.