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
May 12, 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 [1], to routing the traffic from "Internet of Things" (IoT) devices [2], 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 [3], 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* [4]. Metaheuristic algorithms such as Particle Swarm Optimization (PSO) and Genetic Algorithms (GAs) are used to find the ideal combination of parameters needed for a ML model to perform at its best.

According to the "No free lunch theorem" [5], there cannot be one optimization algorithm that is perfectly suited for all kinds of problems. Therefore, tackling these pressing research issues requires not one *perfect* metaheuristic, but several different ones. This is especially true when considering the emergence of new problems and challenges that require or even impose a metaheuristic to solve that optimization problem.

As a result, this increasingly growing scientific field is not only becoming more convoluted [6], but the algorithmic contexts, not necessarily the algorithms themselves, are also becoming more complex. A streamlined metaheuristic framework, such as the Simple Probabilistic Population-Based Optimization (SPPBO) proposes [7], can have multiple parameters to choose from and then, ideally, tune to the problem class and instance it is solving. While problems like the Traveling Salesperson Problem (TSP) or the Quadratic Assignment Problem (QAP) have the advantage of being an abstracted version of real-world applicable problems (and come with their own benchmark packages to boot [8, 9]), there are many of other problems with a variety of factors to consider when configuring the parameters for your metaheuristic algorithm. Moreover, the real-world is rarely static, which means that there is also a need to solve problems that are dynamically changing.

1.2 Problem and Scope

Knowing the context of modern metaheuristics research, this thesis focuses on the problem arising from feature- and parameter-rich metaheuristic frameworks, exemplified by the aforementioned SPPBO framework [7]. It combines and generalizes aspects of 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 has the potential to solve a larger problem space, it also needs to be configured correctly to perform at its best. Solving this task manually, changing the parameters at each iteration and examining the results, is not only inefficient and tedious, but also error-prone, with the risk of getting stuck in a local optimum of a multidimensional parameter space.

This is also the case for Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO) [10], an algorithm derived using the SPPBO framework and incorporating aspects of Hierarchical Particle Swarm Optimization (H-PSO) [11]. 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[1], branching out to its sibling SCEs and their less good solutions, and so on. This structure changes with each new iteration of solutions, establishing a clear hierarchy of influence among 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 is solving and react accordingly to improve the solution, as analyzed similarly in [12].

This opens up the scope of this thesis to dynamically changing problems, such as the Dynamic Traveling Salesperson Problem (DTSP) [13]. While for the "normal" symmetric 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 to, for example, destinations that change during the routing of vehicles, or new cities that need to be visited while the process is already underway.

[1] Specifically, the best solution is found at the root of the tree if no new, better solutions have been found in as many iterations as there are levels in the tree.

1.3 Approach

In summary, 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 has been a research topic since the 1990s [14]. Since then, Hyperparameter Optimization (HPO) has become an important part of this research community, being implemented in almost every modern ML training software and having several open-source standalone packages, written in most common programming languages, with *Python* being one of the most popular choices. It is precisely this knowledge of parameter optimization for functions that are often expensive to execute - be it a nondeterministic algorithm or a complex artificial neural network - that we want to apply to our problem.

In this context, the two main research questions that arise 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*, that contains all the necessary parts to answer the research question outlined above. Every aspect of this package is modular (allowing for easy replacement), highly configurable (allowing for adaptation to algorithms other than H-SPPBO), and well-documented (increasing the comprehensibility and reproducibility of the results 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 required 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. Moving on to the research part of the thesis, Chapter 5 describes the design of the experiments performed and explains in detail the reasoning behind the selection of problem instances and parameter

spaces. Chapter 6 presents the results and discusses them with respect 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 [15] in 1988, which mentioned "interchange methods", such as the 2-opt [16], 3-opt [17], or Lin-Kernighan [18] methods, for solving slow dynamic changes occurring in the TSP. The use of metaheuristics for dynamic problems began its early development starting in the 2000s. The theoretical concept of the DTSP was discussed by Huang *et al.* [19]. Subsequently, the work of Angus and Hendtlass [20] showed, that adapting the Ant Colony Optimization (ACO) to a dynamic change of the TSP is faster than restarting the algorithm, while Guntsch and Middendorf [21] 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 have been proposed by Eyckelhof and Snoek [22]. In contrast to changing the node topology of the TSP, Silva and Runkler [23] applied dynamic constraints to the nodes and left the evaluation to the ants. More recent work on the use of ACO on different versions of the DTSP has been done by Mavrovouniotis and Yang [24], [25].

Examining other metaheuristic categories, Li *et al.* [26] 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 [27], who use an algorithm based on CHC ("Cross-generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation" [28]) to solve a dynamic version of the TSP with changes in edge weights and insertions, deletions, and swapping of city nodes.

PSO has also been successfully used to react to dynamic changes in works by Janson and Middendorf [12], [29]. Several variants of the PSO - H-PSO and a partitioned version (PH-PSO) - were used to not only solve, but also to detect changes within the presented dynamic problem instances.

2.2 Parameter Optimization for Metaheuristics

The choice of parameters has always been an important consideration in metaheuristics research. Eiben *et al.* [30] give a thorough review and analysis of the different options in the field of parameter settings. Although they focus on GAs, they provide 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* methods are then distinguished by their type of value modification. Eiben *et al.* conclude that a *parameter control* strategy usually leads to better solutions compared to *parameter tuning*. Talbi [31] made a similar classification, including a further distinction between (offline) parameter tuning methods. Lastly, in their review of parameter optimization, Wong *et al.* [32] 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.* [33, 34]. The resulting values from several experiments with different parameter combinations performed on a TSP instance serve as a baseline for many subsequent studies. An updated version of these "good parameters", including variations of the original ACO, was published by Dorigo and Stützle [35]. These parameter combinations were then challenged by Gaertner and Clark [36], who classified the TSP instances by certain properties, and then tried a wide range of parameters for these new TSP categories. They found optimal parameter values, in some cases very different from those originally proposed. More recent work in the area of parameter tuning is done by Tuani *et al.* [37], where, for a heterogeneous ACO, each ant is initialized with different random distributions for the α and β values. The algorithm has been tested on several TSP instances and compared against different ACO variants and their heterogeneous counterparts.

A thorough review, classification, and research on online parameter control has been done by the aforementioned Stützle *et al.* [38]. Based on the taxonomy proposed by Eiben *et al.*, they modify and apply these categories to other approaches in the field of ACO parameter control. In addition, their experiments with deterministic parameter control strategies take into account the computational time and the anytime behavior of the algorithms. Further research on this type of parameter optimization has been done by Neyoy *et al.* [39], where fuzzy logic statements are used to improve the solution diversification behavior. A self-adaptive control scheme has been proposed by Hao *et al.* [40], which constructs a combinatorial problem of the parameter search and applies PSO to optimize them in each iteration. Benchmarks using the TSP promise good results. Similar studies have been done by Li and Zhu [41], with a version of the ACO whose

parameters are controlled by a bacterial foraging algorithm, and compared to parameter control by GA and PSO. Other interesting developments include the work of Randall [42], who constructed 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 [43]. They analyzed the similarities to problems faced by supervised learning and proposed guidelines for sampling parameter sets. Finally, they applied their results using a version of the Hoeffding race algorithm [44] (originally used to select good models, of which HPO is a subset) to, among other things, tune the parameters of a MAX–MIN Ant System to solve the TSP. Another approach in the context of applying ML concepts to parameter optimization was proposed by Dobslaw [45], who explained the possibility of training an artificial neural network (ANN) on the relationship between the characteristics of a problem instance and a parameter set that led to good solutions. The necessary training data is to be obtained using the Design of Experiment (DoE) framework. In this sense, several other authors have also proposed some form of DoE variant, in this case the Taguchi method, to optimize or select ML models and/or their hyperparameters, see [46, 47, 48].

Finally, an approach similar to this thesis, but more narrowly focused, was investigated by Yin and Wijk [49]. They used two hyperparameter optimization methods, Random Search (RS) and Bayesian Optimization (BO), to tune the parameters of a classical ACO algorithm on multiple instances of the asymmetric TSP. Their results promise great potential for tuning parameters in this way, without requiring a priori knowledge of the problem or the algorithm.

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 done by Robinson in 1949 as part of an United States research company, a think tank called "RAND Corporation", which offered its services to the U.S. armed forces [51]. Besides Robinson's proposed solution, the scientific community at that time was very interested in the TSP, applying all kinds of mathematical graph operations and often using branch-cutting algorithms to solve it [52]. The beauty of this problem lay in its simple, short description, which made it easy to understand, but also in its non-trivial and engaging solutions.

With advances in computer science also came more computationally applicable algorithms, such as the Kernighan–Lin heuristic [18]. At the same time, the TSP was found to be NP-complete and therefore, NP-hard [53] - a problem category that still remains a very interesting research topic in computer science. Through the work of Dorigo *et al.* [33] and the "ant system", metaheuristics began to be a valid choice for solving the TSP in the early 1990s. In parallel, with the emergence of the *TSPLIB* benchmarking suite [8], a wide adoption of said test instances began to compare and rank new algorithms and (meta-)heuristics. Applications for the solutions are numerous, ranging from apparent routing of travel routes to frequency assignment [54].

3.1.2 Theory

The problem description of the symmetric TSP can best be modeled by an undirected weighted graph $G = (V, E)$, with a set of n vertices $V = \{v_1, ...v_n\}$ and a set of edges $E = \{(v_i, v_j)|v_i, v_j \in V, i \neq j\}$. Staying with the traveling salesperson analogy, the vertices are often being referred to as "cities", while the edges between vertices represent the

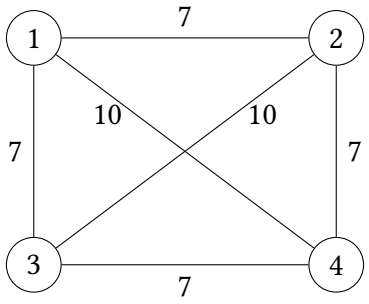


Figure 3.1: Example of a symmetric TSP instance with $n = 4$ cities

distance. This can be expressed by a distance function $d : V^2 \rightarrow \mathbb{R}_0^+$, which, in the case of the Euclidean TSP, is just the Euclidean distance between two points in a two-dimensional space. Each city can be visited only once, and since it is symmetric, it does not matter which direction the abstract salesperson travels. The problem is to find the shortest (i.e., smallest total distance) tour that visits each city exactly once, starting and ending in the same city. The solution is similar to 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 of the following definition:

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

with the city node s_{i+1} coming after s_i and $s_{n+1} = s_1$ to complete the full tour. A more precise description of the problem is as follows:

> 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, with every vertex connected to every other vertex. Especially for computational tasks, it is 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 is shown 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 several modifications of this standard problem. The asymmetric TSP (ATSP) has the added complication that the weights between the vertices are different depending on the direction the edge is being traversed, such that $d(v_i, v_j) \neq d(v_j, v_i)$, which increases the complexity and optimization potential for these instances [55]. The Dynamic Traveling Salesperson Problem (DTSP) incorporates certain dynamic aspects, such as changing edge weights or inserting, deleting, or swapping city nodes, into the problem domain. It is different in the sense that it has neither a specific problem description nor universally valid solutions, since this all depends strongly on the implementation. The basis is often 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) also becomes time-dependent $f(s, t)$.

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 it is often mathematically proven that an optimal solution can be reached in reasonable time, this time can be considerably long. A metaheuristic generally does not deal with exact solutions, but rather tries to find the *best* solution in a given, often small enough, time frame. Furthermore, a heuristic algorithm requires specific knowledge about the problem it is solving, whereas metaheuristics are generally adaptable to a larger space of optimization problems and do not expect the problem's formulation to be of such strict a format [56]. Algorithms of this type often include 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 gain experience. This principle is one of the main distinctions between different metaheuristics and their configuration [57].

From the Greek prefix *meta*, roughly translated as *high-level*, a metaheuristic can be understood as a "high-level, problem-independent algorithmic framework" [56] 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 [58]. 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 has also given rise to so-called "hybrid metaheuristics", which mix and match ideas from several frameworks into one algorithm [6].

Giving a definitive taxonomy of metaheuristics is impractical because there are many characteristics that can distinguish them. 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 [56]
- ...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 alone, 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 provide a brief overview of the field because it provides an intuitive separation.

Single-Solution Based Metaheuristics

Single-Solution Based Metaheuristics (S-metaheuristics) improve on a single, initial solution and describe a trajectory as it moved through the search space. Hence, they are often referred to as *trajectory methods*. The dynamics applied to each new iteration of solutions depend heavily on the specific strategy. However, they can generally 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 [31]. 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 its 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 cost of more computation.

Algorithm 3.1 Basic Local Search

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

Algorithm 3.1 shows a high-level description for one of the first and simplest S-metaheuristics, the Basic Local Search. The functions (generate, select) and the neighborhood N vary depending on the implementation. For example, the generation of new solutions can 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 [31]. The greatest issue with Basic Local Search is convergence to local optima, and most other implementations of a S-metaheuristic extend this algorithm to counteract this flaw in some way [59].

One of these is Simulated Annealing (SA), which is based on the physical process of annealing. When 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 metal's 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 SA eventually converges to a simple local search algorithm [59].

Tabu Search (TS), on the other hand, uses a list (*memory*) to explicitly utilize the search history to its benefit. In its simplest form, TS uses a *short-term memory* to remember the most recent solutions, limiting the neighborhood to solutions not present in the list. Thus, 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) [31].

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 [59]. 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 the current population or even replaced by it (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 [31].

Algorithms with analogies to swarm intelligence, often found in animals in nature, are an example of P-metaheuristics. These types of algorithms typically work with agents that are individually not complex and use 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), which refer to biologically inspired operations such as 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.* [34]) 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 [60]. 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, thus reinforcing this path with its own pheromone. This autocatalytic (positive feedback) behavior is counteracted by the volatility of the pheromone, which dissipates over time, weakening path reinforcement [34]. This results in the shorter paths accumulating higher amounts of pheromone, as shown in Figure 3.2. This example also shows the response to a dynamic change in the path, which is an obstacle placed directly on the shortest route. Although the left path is initially as likely as the right path, the reinforcement on the shorter right path is greater, causing the ants

[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 [31].

to adapt to the obstacle [31]. However, 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 [35].

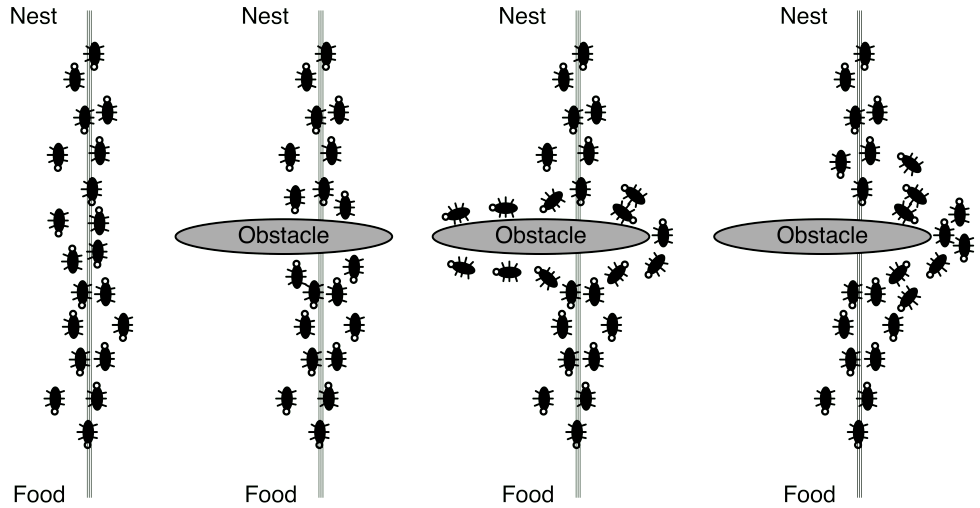


Figure 3.2: The process of ants following a pheromone trail between a food source and their nest affected by an obstacle. Modified from Talbi [31].

The artificial ants in ACO are inspired by this behavior and iteratively build 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 [61].

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 straightforward. 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 uniformly, so that each path is initially equally likely to be chosen. With a total of n cities to visit, the resulting matrix

is of dimension $n \times n$, where each entry τ_{ij} represents the amount of pheromone being present at the edge (i, j). For every complete cycle, each of m ants probabilistically creates a solution with this pheromone information τ_{ij} and heuristic information η_{ij}. Starting from a randomly selected city i, the probability to visit the next possible city j from a set S of not yet visited cities is given by

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

where η_{ij} holds the problem specific heuristic value, which is the inverse distance $\eta_{ij} = \frac{1}{d_{ij}}$ between cities i and j in case of the TSP. The constants $\alpha, \beta \geq 0$ control the influence of either the stochastic pheromone or the heuristic value. 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 [31].

The pheromone information is then updated based on the generated solutions. First, each pheromone entry is subject to evaporation controlled by a constant value $\rho \in (0, 1]$ and defined by 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 chosen 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 after each ant has finished constructing its solution. More common, however, are offline strategies that are called after each ant has finished. One of the simpler implementations in this category is the "elitist pheromone update", where the updates of the pheromone matrix are strongly influenced by the global best solution. Another approach is the "quality-based pheromone update" or "ant-cycle" [34], where each ant $k = 1, 2, ..., m$ updates the pheromone matrix relative to the length of the solution L_k they found in that iteration, which is further controllable by 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 number of iterations. With the pheromone update implemented as an "ant-cycle" the time complexity of this

algorithm is $O(\mathrm{NC} \cdot n^2 \cdot m)$ [34]. Based on this algorithm, a large number of variants have been created. One of them is a population-based approach, which is discussed in the following.

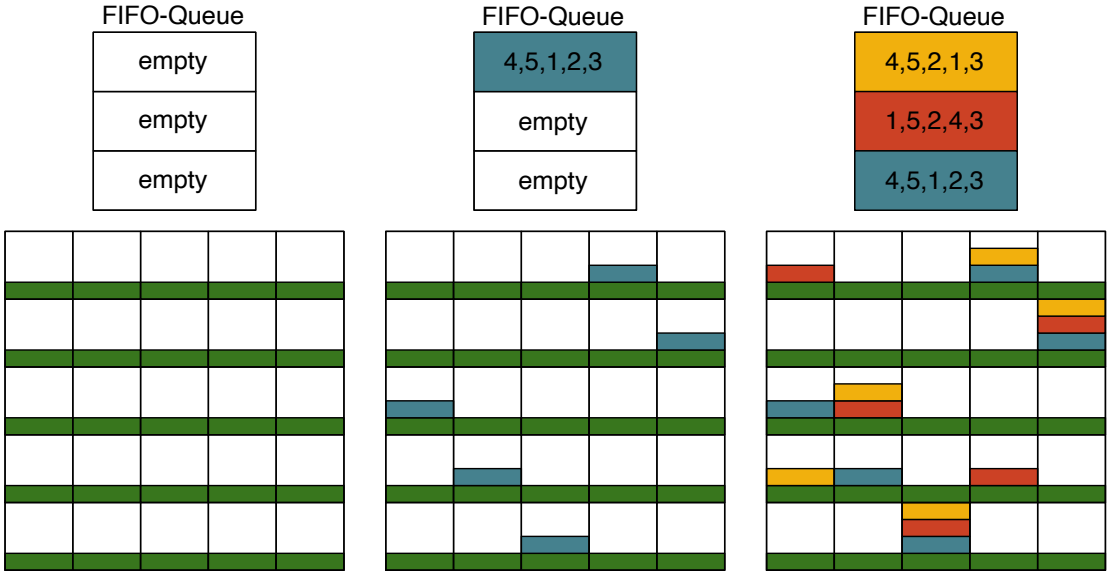
Population-Based Ant Colony Optimization

The Population-Based Ant Colony Optimization (PACO) was first proposed by Guntsch and Middendorf [62] as a simplification of the ACO, effectively reducing the operations required to update the pheromone information by quantifying and limiting the values added to the matrix. Their motivation was to speed up the process and reduce the influence of older solutions in order to apply the PACO to the DTSP. Instead of storing all updates to the pheromone matrix, the approach keeps track of the solutions that have updated the solution in a queue, called the *population*. Therefore, every solution in this queue is actually present in the matrix and vice versa. To limit the influence of old solutions, the population size is set to a value k. When the queue is full after k iterations, several 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 analogous to the ACO presented earlier. The matrix is initialized with a 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 [62].

Figure 3.3 shows an example of a pheromone matrix with a solution population of $k = 3$ being updated over the course of three iterations for a problem of size n. In Figure 3.3a, the population is empty and the matrix is initialized with τ_{init}, visualized by the green bars. The best solution of the first iteration has the city referenced at position 4 as its start, continuing with position 5, and so on. This update is visualized by blue colored bars. Eventually, after two more iterations (Figure 3.3c), the population queue is full. In a next iteration, the solution visualized in blue would leave the matrix and a new one would be placed at the top of the queue.

3.2.3 Particle Swarm Optimization

The Particle Swarm Optimization (PSO) method was introduced by Kennedy and Eberhart [63] 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) 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).

a given problem space of dimension d. Therefore, each of the N particles in a swarm represents 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 $f(\vec{p_i})$

To let the particles influence each other, a neighborhood rule must 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 for letting the particles exchange information is the local best method (**lbest**). In this method, particles interact based on a given topology, such as a ring, where only direct neighbors exchange information. Thus, regardless of the strategy chosen, each neighborhood k has a leader with the best solution $\vec{g_k}$ [31]. Putting both aspects together, the particles update their velocity based on personal success (*cognitive aspect*) and neighborhood success (*social aspect*) [11].

Algorithm 3.3 Particle Swarm Optimization

```
Initialize swarm
repeat
    for all particles i ∈ [1, N] do
        UpdateVelocities
        UpdatePosition
        if f(x_i) < f(p_i) then p_i = x_i
            if f(x_i) < f(g_k) then g_k = x_i
until termination criterion met
```

Algorithm 3.3 shows a high-level description of the PSO procedures. Typically, the swarm is randomly initialized, with each particle assigned a velocity and a 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:

$$\vec{v}_i(t+1) = w \cdot \vec{v}_i(t) + c_1 \cdot r_1 \cdot (\vec{p}_i - \vec{x}_i(t)) + c_2 \cdot r_2 \cdot (\vec{g}_k - \vec{x}_i(t)) \tag{3.7}$$

with an inertia weight $w > 0$ that controls the influence of particle velocity. The parameters $c_1, c_2 > 0$ define the influence of the personal best $\vec{p}_i$ and the 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 Figure 3.4, which shows a particle i being updated according to all three possible influences.

Research has been performed to understand how the algorithm behaves and how to vary the algorithm. In particular, the choice of neighborhood topology has a significant impact on performance. The **gbest** method seems to perform better on unimodal problems, where there is one clear function optimum, while the **lbest** method gives better performance on multimodal problems with many optimal function values [11].

One such variant is the Hierarchical Particle Swarm Optimization (H-PSO) proposed by Janson and Middendorf [11], 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

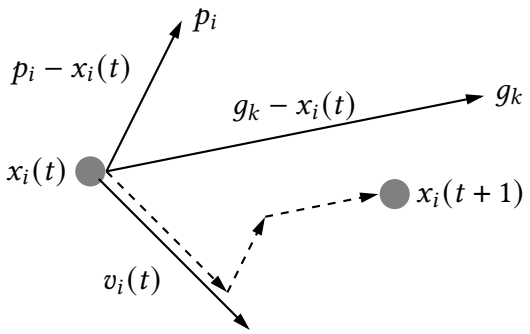


Figure 3.4: The vector summation of a PSO particle i during velocity and position update.

particles are represented by the nodes of the tree, and, therefore, the neighborhood of each particle i is defined only by its direct parent j in this hierarchy, so $\vec{g}_i = \vec{p}_j$. The updating of velocity and position remain the same as in 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 found 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, capable of solving combinatorial problems such as the *MMRAP* (multi-level, multi-state redundancy allocation problem), as proposed by Yeh [64]. The solution of the particle is no longer represented by the position and the velocity in the multidimensional 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 tunable intervals [65]. 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 $\mathbf{s}_j^{t+1}$ for particle j at iteration $t+1$ 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 [7].

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 that combines generalized aspects of population based approaches, such as PSO and SSO, and strategies that effectively generate probability distributions, such as ACO and PACO. As discussed by Lin *et al.* [7], 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, they all share the same schematic foundation. First, a distinction must be made between populations and solution creating entities (SCEs). Reminiscent of an ant (ACO) or a particle (PACO), a set of SCEs $\mathcal{A}$ create the solutions by applying probabilistic rules Prob_V and, optionally, some form of heuristic information η. These solutions may then enter a 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 is shown in Algorithm 3.4.

Algorithm 3.4 SPPBO

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

After the populations are initialized with random solutions, each SCE creates one solution, resulting in $k_{\text{new}} = |\mathcal{A}|$ new solutions per iteration. The underlying probability distribution is affected 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 on the population can be further controlled by the weights w_p and w_r. The populations are then updated based on a set of rules specific to the implementation. For example, in a version of SPPBO with only one global best population, the update procedure may insert the iteration best solution and save a total of k iterations [7].

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 such as PACO, the goal of this algorithm, as stated by Kupfer *et al.* [10], was to not only to solve the DTSP, but also to detect these dynamic changes and react accordingly.

Similar to the descriptions of SPPBO, there is a set $\mathcal{A}$ of SCEs and a set of populations $\mathcal{P}$. The important difference here is that these populations $P \in \mathcal{P}$ each belong to a 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. To take advantage of these parent-children relationships in the population, the hierarchical aspect is implemented similarly to H-PSO (see Section 3.2.3). The SCEs are organized in an m-ary tree[2], where each "child SCE" is influenced by its parent(A), and a "root SCE" $A*$, defined as its own parent (parent($A*$) = $A*$). This hierarchy allows for a clear definition of the following populations $P \in \mathcal{P}$ for each SCE A:

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

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. Note also that due to the tree structure $P^A_{\text{parentbest}} = P^{\text{parent}(A)}_{\text{persbest}}$. Each of these populations has a corresponding weight $w_{\text{persprev}}, w_{\text{persbest}}, w_{\text{parentbest}} \geq 0$ to control their respective influence. Now, to create

[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.

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 best solution of the parent and a distinct random influence p_r through a random weight w_{rand}.

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

Algorithm 3.5 H-SPPBO

```
Initialize the SCE tree
Initialize SCEs with random populations P^A_persprev, P^A_persbest
repeat
    for all SCE A ∈ 𝒜 do
        CreateSolution(A)                                   // using (3.10) and (3.11)
        UpdatePopulations(P^A_persprev, P^A_persbest)
    swapNum = 0
    for all SCE A ∈ 𝒜_tree do
        if f(s^A_persbest) < f(s^parent(A)_persbest) then
            Swap(A, parent(A))
            swapNum ← swapNum + 1
    if swapNum > [θ · |𝒜|] and no change in L_pause previous iterations then
        ChangeHandlingProcedure(H)
until termination criterion met
```

Algorithm 3.5 shows the process for solving a dynamic problem. It is similar to the template given for SPPBO (see Algorithm 3.4), with a solution creation and a population update phase. However, since the populations are directly related to the SCEs, the update phase is performed 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, at each iteration, every SCE creates one solution using the following procedure: Start with a set of all unvisited nodes $U \subseteq V$ and set a random starting 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 to check whether the current, possible edge (i,k) has been previously visited by the SCE ($P^A_{\text{persprev}}, P^A_{\text{persbest}}$) or its parent ($P^A_{\text{parentbest}}$), i.e., formally, if the ordered set (i,k) is a subset of the solution vector $\mathbf{s}_P$ of the 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 in the ACO metaheuristic, $\alpha, \beta \geq 0$ are parameters to control the stochastic and heuristic influence, respectively. The probability of 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 to a probability distribution over all unvisited nodes U. Finally, a node j is randomly drawn from this distribution, added to the solution vector $\mathbf{s}$ and removed from the unvisited set $U \leftarrow j \setminus U$. With this new node being the next current node, the process is repeated until the set of unvisited nodes is exhausted. And eventually, the populations are updated.

With new solutions calculated, the hierarchy is now subject to change. The SCE tree is iterated in a top-down, breadth-first manner ($\mathcal{A}_{\text{tree}}$), and each SCE compares its personal best solution with its parent by using an evaluation function $f : V^n \rightarrow \mathbb{R}^+_0$, 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 this swap operation is shown in Figure 3.5. By using this top-down approach, comparatively worse solutions can descend all the way to the lowest 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 when a certain threshold of rearrangements in the SCE tree is exceeded. Specifically, the number of swaps from the previous part is being compared to 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 $\theta = 1$ requiring a complete rearrangement of the tree to render the condition true. However, this

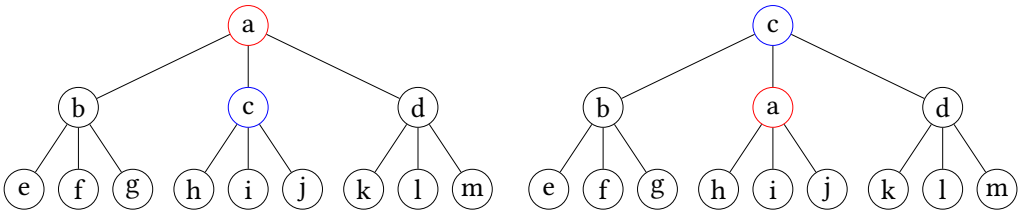


Figure 3.5: Example of a ternary SCE tree ($m = 3$) with height $h = 3$ showing the SWAP(A,C) operation before (left) and after (right) the execution. Modified from Kupfer *et al.* [10].

condition may be met without the problem instance actually changing, leading to false detections. Regardless, when 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.

While H_{full} can be understood as a complete reset of the algorithm, H_{partial} tries to apply some of the best "knowledge" to solve this changed problem (exploitation), while the lower performing SCEs may instead concentrate on finding new solutions (exploration). Additionally, due to the potentially major rearrangements in the tree as a result to these change handling procedures being executed, the parameter $L_{\text{pause}} > 0$ acts as a guardrail to prevent the procedure from virtually triggering itself, providing the hierarchy a few iterations to settle [10].

3.3 Parameter Optimization for Metaheuristics

3.3.1 Overview

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

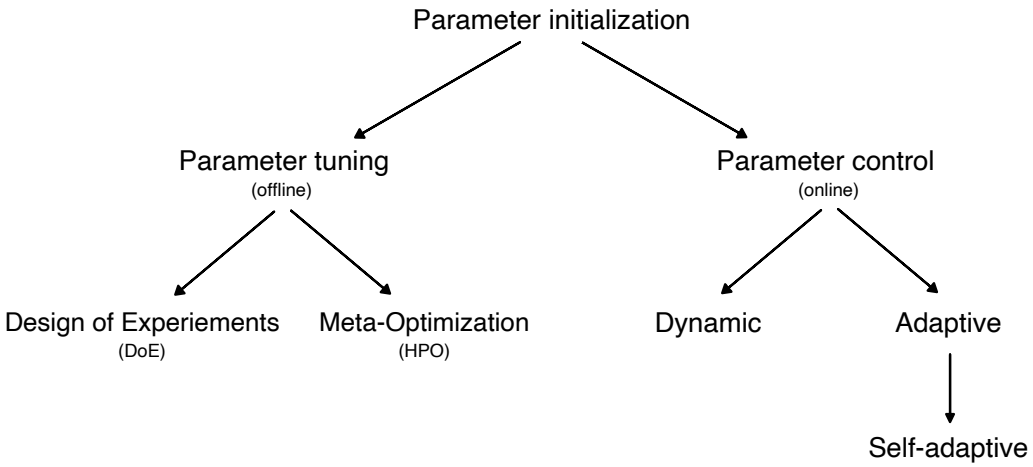


Figure 3.6: Taxonomy of parameter optimization. Modified from Eiben *et al.* [30].

This thesis applies methods from machine learning research to address this 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 from Eiben *et al.* [30], with additions from Talbi [31] and Stützle *et al.* [38]. There are two main differences: "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 while the algorithm is running, allowing for potentially better adaptation to the problem.

Parameter Control

Parameter control methods can be advantageous because they can 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 of implementing such online parameter initialization methods have been proposed. *Dynamic* strategies alter the values according to a specific deterministic rule, while also allowing for minor random influences on 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 this. Lastly, *self-adaptive* parameter control implements the parameter change into the metaheuristic algorithm itself, making it part of its search space and thus the solution.

Parameter Tuning

The simplest 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 the possibility of further statistical significance tests. The other method uses meta-optimization, which is an algorithmic layer 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. Similar to parameters in metaheuristics, tuned hyperparameters can greatly improve the performance of a ML model over the default settings. In particular, state-of-the-art deep learning techniques, which have a large number of tunable parameters, can only be properly utilized if the hyperparameters are specifically tuned for their problem.

Traditionally, simple gradient descend-based approaches are often used for optimization problems. In its simplest form, the objective function $f(x)$ is to be minimized by $\min_{x \in \mathbb{R}} f(x)$, with a region of feasible values $D = \left\{x \in X | g_i(x) \leq 0, h_j(x) = 0\right\}$, and possible range and equality constraints $g_i(x), h_j(x)$ from a set of all values X. Following the negative gradient direction, a global optimum could be obtained for convex functions [4]. However, ML models pose some challenges to these established 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.

In parallel with research on optimal parameters for metaheuristics (see 2.2), the manual tuning of hyperparameters in ML applications began to become impractical in the face 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 first applied [14].

The HPO problem statement can be formulated as follows [14]: Let $\mathcal{A}$ be a ML algorithm with N hyperparameters, where Λ_n denotes the n-th hyperparameter. The complete hyperparameter configuration space can then be defined as $\mathbf{\Lambda} = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_N$, with a vector of a possible hyperparameter configuration $\lambda \in \mathbf{\Lambda}$. Thus, a ML algorithm initialized with hyperparameters λ is denoted by $\mathcal{A}_\lambda$. Based on this, the process of HPO consists of four main components: 1) an estimator (most often a regressor, but a classifier is also possible), 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 performance of the configuration (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 $\mathcal{D}$ divided into training data D_{train} and validation data D_{valid} that minimizes the expected value $\mathbb{E}$ of 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 related to the objective function itself, e.g., the resulting tour length in solutions to 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 $f : \mathbf{\Lambda} \rightarrow \mathbb{R}$ be defined directly by its parameter space $\mathbf{\Lambda}$, which is mapped into the real solution quality space $\mathbb{R}$.

There are several ways to solve the above mentioned problem statements. A common distinction between these methods is their use of the objective function and its loss landscape. A HPO algorithm can either treat this as a black-box function, using full evaluations of the objective function to model its behavior, or it can 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 [14]. Another important distinction, especially in the area of black-box HPO, is whether or not to use a statistical model, .i.e an estimator. Model-free techniques rely solely 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 [4]. Although, this approach of using metaheuristics to tune a metaheuristic is similar to the process proposed by [31] (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 later 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, it evaluates all possible combinations of hyperparameter values. GS only requires finite sets of value ranges to be defined for each hyperparameter, similar to Λ. It then iterates over the entire search space by creating the Cartesian product of these sets. This approach is easily interpretable and repeatable, while also being trivially implemented and parallelized [66]. However, each new parameter causes the Cartesian product to grow exponentially, so GS suffers greatly from the "curse of dimensionality". Another problem is its inability to explore promising value ranges on its own. Since these must be predefined, a user would have to manually adjust these ranges before each run [4].

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 to try all possible combinations [66]. 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, thus giving each of N hyperparameters B different evaluations, as opposed to $B^{1/N}$ with GS (would it also operate on a fixed budget) [66]. This gives parameters with a higher partial dependence on 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, as a model-based HPO algorithm might have [4]. Nevertheless, it still serves as a well-performing baseline for many ML benchmarks, with hyperparameters relatively close to the optimum, given sufficient resources [14].

Algorithm 3.6 Random Search

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

3.3.4 Bayesian Optimization

BO is not just 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 the assumption of the evident data it samples $\mathcal{D}_t = \{(\lambda_\mathbf{i}, f(\lambda_\mathbf{i}))|i \in [1, t]\}$ yields a likelihood $P(\mathcal{D}_t|f)$, which can then be combined with the prior distribution $P(f)$ leading to the posterior distribution by applying Bayes' theorem:

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

In essence, this expresses the likelihood of the sampled data under the assumptions made for the objective function [67]. Applying this in practice means that more sampled data points will cause the posterior function to adjust its mean for those points, reducing its uncertainties and increasing its predictive power [68].

A common interpretation of this theory for effective implementations of BO is an iterative algorithm, given by Algorithm 3.7, consisting of two main parts: a surrogate model analogous to the posterior function over the objective and an acquisition function that guides the sampling process to the optimum. After initializing the model with an optional number of initial samples $\mathcal{D}_{\text{init}}$, a maximum of *n_calls* are made to the objective function

Algorithm 3.7 Bayesian Optimization

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

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 evaluate the search space to find the most beneficial next point to sample. High acquisition corresponds to potentially optimal objective function values. By selecting widely unsampled parameter areas, the function realizes exploratory behavior, while further sampling in already well-performing areas employs exploitative strategies. The specific choice of the acquisition function always defines a (customizable) trade-off between these two [67]. With the most promising parameter input λ_t of iteration t specified, the (potentially) noisy objective function $f(\lambda_t) + \epsilon_t$ gets sampled. The noise is often modeled as an independent Gaussian distribution with zero mean and variance σ_n^2:

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

following the original proposition of Williams and Rasmussen [68]. Afterwards, the probabilistic surrogate model is fitted to the observations $\mathcal{D}_t$, which implements the update procedure of the algorithm. This surrogate model ideally represents the actual objective function as closely as possible, while also being mathematically advantageous and easy to compute [14]. A standard choice for this would be a Gaussian Process (GP), although many other models can be used, such as Random Forests (RF) or Gradient Boosted Regression Trees (GBRT). An example of the process can be seen in Figure 3.7, which shows the first three iterations of BO applied to a 1D toy function, using a GP surrogate with two initial random samples.

Various 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, greatly improving on model-free methods. However,

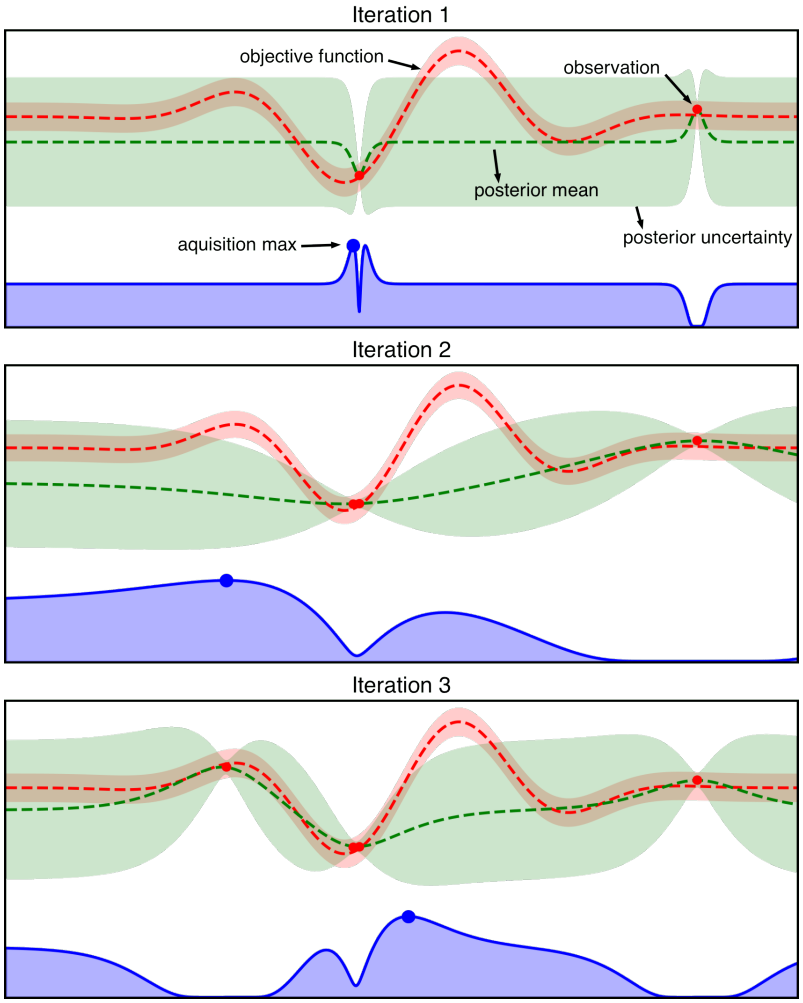


Figure 3.7: An example of BO using a GP surrogate (mean prediction as green dotted line, uncertainty as green tube) and an acquisition function (lower blue curve) on a noisy 1D toy function (red dotted line with red tube as noise). The figure shows three different iterations of the BO process using two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

the sequential dependence on previously sampled data makes parallelization difficult [4]. The following subsections briefly describe some popular acquisition functions and then discuss three surrogate models in more detail. These models were chosen based on their variance with respect to the BO process. A more detailed justification is given in Section 5.2.

Acquisition Functions

As mentioned before, an acquisition function guides the sampling process, balancing exploration and exploitation. Given the distribution of the surrogate model, including its predictive mean $\mu(\lambda)$, its variance function $\sigma(\lambda)$, and the previously sampled data $\mathcal{D}$, where $f_{min} = \text{argmin}_{\lambda \in \mathcal{D}} f(\lambda)$ is the best observed value so far, the acquisition function is defined as $u : \mathbf{\Lambda} \rightarrow \mathbb{R}^+$ [69]. Two common choices for this function are the probability of improvement (PI) and the expected improvement (EI).

Probability of Improvement PI as proposed by Kushner [70] attempts to maximize the probability of improving over the current best value f_{min}, with little regard for 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 by nature, but offers a guaranteed improvement of at least ξ [67].

Expected Improvement As proposed by Jones *et al.* [71], EI improves on 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 the sample to be chosen to give a maximum improvement over the previous best observation. EI can also be expressed in a closed form with an additional trade-off parameter $\xi \geq 0$ to balance 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, favoring high means, while the second addend handles the exploration, selecting points with large surrogate variances [67].

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 their mean $m(\lambda)$, and a covariance function $k(\lambda, \lambda')$:

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

where the mean function is most often taken to be zero, so that the process relies solely on the covariance function, whose specification assumes a distribution over the objective function [68]. The default choice for this so-called *kernel* (implying the use of the *kernel trick*) in the original proposal is a squared exponential function, which specifies 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 the 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 [67, p.8]). A disadvantage of the squared exponential kernel is that it assumes a very smooth objective function, which is unrealistic for real physical processes [72]. Instead, a Matérn kernel is often suggested as a replacement. It uses a parameter v to control smoothness, and common settings for ML applications are $v = 3/2$ and $v = 5/2$ [68].

The default GP scales cubically with the number of data points, which limits the number of function evaluations. Another issue is poor scalability to higher dimensions, which is attempted to be overcome by using other kernels [14].

Surrogate Model: Random Forest

Very different from GP are the next two approaches, which come from the field of ensemble methods often used in ML. The basic idea of these methods is to train several 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 produce a final predictor of the

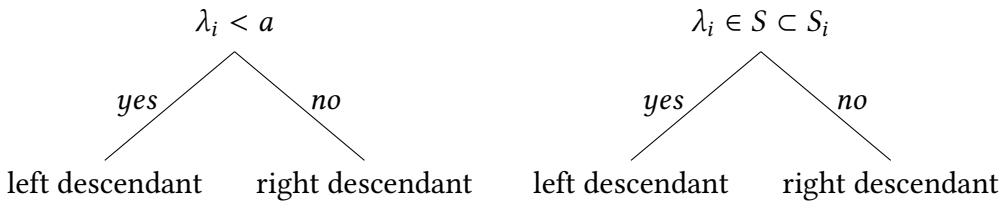


Figure 3.8: Example of two decision trees. (Left) Splitting a continuous variable λ_i at point a. (Right) Splitting a categorical variable λ_i via subset S.

objective function $\hat{f}(\lambda)(\approx f(\lambda))$. In the case of regression, which applies to HPO, this means a simple average of all base learners [73]:

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

RF is one of these methods, and was first introduced under this name by Breiman [74] as an improvement on his bagging algorithm for decision trees. Each base learner, as described above, is a binary partitioned regression tree, where each partition or "split" is based on a predictor variable $\lambda_i \in \lambda$ (i.e., 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 this task, while classification typically uses the Gini index to calculate the "purity" of each potential class [73].

The bagging algorithm now improves on the foundation of decision trees [75]. 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 selects a random subset of the same size n as the original data set, but with replacement, thus 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 to validate the estimator. The tree is then fitted using this sample as described above.

RF introduces more randomization into the decision tree building process by taking not only a subset of the available data for each base learner, but also taking a random sample (with replacement) of size m, where $m < N$ of the predictor values for each split. As a result, the base learners do not overfit to presumably highly predictive variables, which reduces the correlation among the sub-sampled data sets and increases the accuracy of the ensemble prediction [76]. The resulting implementation template is described in Algorithm 3.8. The process can also be easily parallelized since each base learner is constructed individually [73].

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)

Extremely Randomized Trees As the name suggests, the extremely randomized trees approach proposed by Geurts *et al.* [77] 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 predictor variables. To explain further, the sampled m predictor variables are each randomly split once, evaluated (e.g., using the mean squared error), and then the best performing split is chosen for the current node. In addition, each base learner is now built using the entire data set, rather than just a bootstrap sample. The motivation behind ET is to reduce variance through random splits and minimize bias by using the full data set, while also having the potential to improve the computational time needed to build the estimator.

Surrogate Model: Gradient Boosted Trees

GBRT is also an ensemble method that uses decision trees as base learners to produce an ensemble prediction, and was originally proposed by Friedman [78]. However, unlike RF, which averages many full-depth decision trees, GBRT sequentially builds many small, high-bias decision trees (depth $d \approx 4$), improving on each other using the residuals from the last iteration. The schematic architecture of this approach is outlined in Figure 3.9. For simplicity, let $f_j = f_j(\lambda)$. The ensemble regressor $\hat{f}_j$ is the j-th of all J estimators in an additive sequence

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

where $v \in (0, 1]$ is a "shrinkage" parameter that controls the learning rate, leading to better generalization [80]. Let $\mathcal{L}(f, \hat{f})$ be the loss function for the estimator. Now, at each iteration j, a new base learner h_j is added to the ensemble by minimizing over its

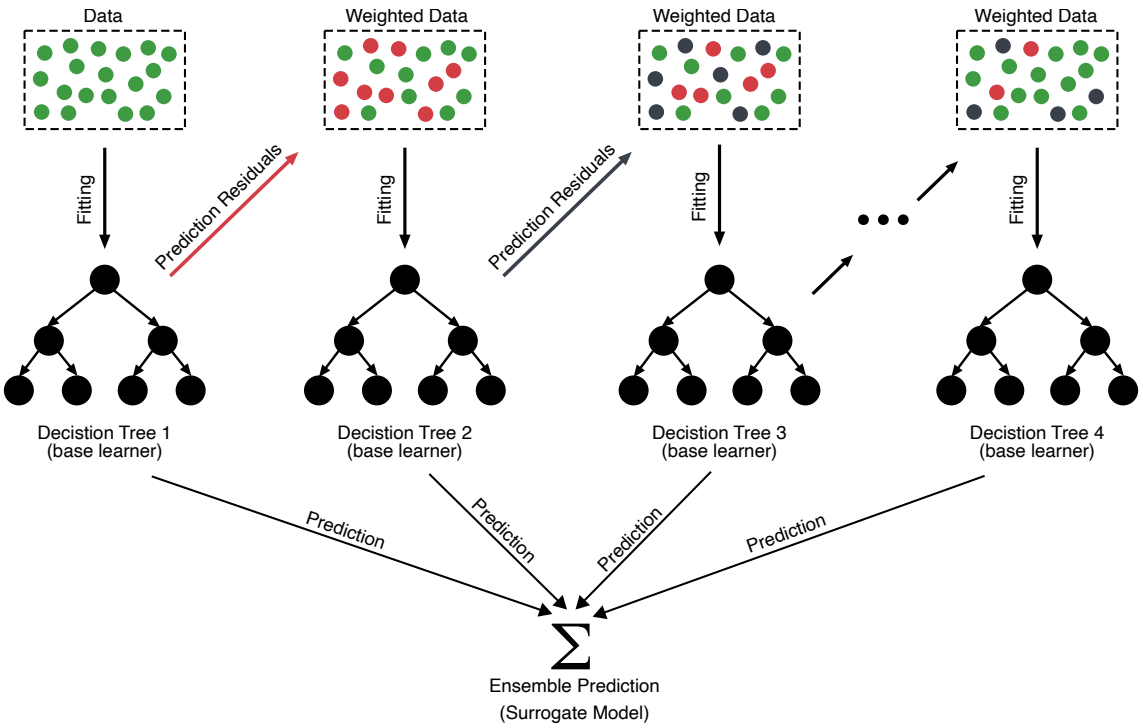


Figure 3.9: Schematic view of GBRT. Modified from Deng *et al.* [79].

sum of losses for the entire data set $\mathcal{D}$ ($|\mathcal{D}| = n$), with (λ_i, y_i) being the i-th element of $\mathcal{D}$[3] and $y_i = f(\lambda_i)$ [78]:

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

To make this a computationally closed-form, a first-order Taylor approximation to the loss-function is used to obtain 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 the gradient g_{ij} of 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 is fitted using all predictor variables to minimize the negative gradient of the all n data points:

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

[3] Note that this is an exception to the previously established notation that λ_i is the i-th parameter in the configuration.

Given a typical squared error loss function $\mathcal{L}(f, \hat{f}) = \frac{1}{2}(f - \hat{f})^2$, the negative gradient can be simplified to an ordinary residual $r_{ij} = -g_{ij} = y_i - \hat{f}_{j-1}(\lambda_i)$ [81, 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) [82]. Although this algorithm uses a constant initialization for the residuals, other methods could be used as well.

Algorithm 3.9 Gradient Boosted Regression Trees (Squared Loss)

Initialization: $\forall i \in [1, n] : r_i = y_i$
for $j = 1$ to J **do**
  $h_j \leftarrow$ CreateTree($\{(\lambda_1, r_1), ..., (\lambda_n, r_n)\}, d$)
  for $i = 1$ to n **do**
    $r_i \leftarrow r_i - v \cdot h_j(\lambda_i)$
$\hat{f} = v \cdot \sum_{j=1}^{J} h_j$
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, the following is a theoretical example using the mathematical symbols and formula from above. This example focuses on BO using a RF surrogate model, and the underlying problem to be solved is 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 $\mathbf{s} = (s_1, ..., s_n) \in V^n$, where V is 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 solution $f : V^n \rightarrow \mathbb{R}_0^+$ 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 $f(\lambda) = L = y$, where y is 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 $\mathcal{D}_t = \{(\lambda_\mathrm{i}, y_i) | i \in [1, t]\}$ be the data set consisting of the parameter pairs and their corresponding solution qualities, sampled so far during the HPO process.

Starting with the BO process, we first obtain an initial data set $\mathcal{D}_\mathrm{init}$ 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 $\{((..., 8, 2, ...), 100), ((..., 7, 3, ...), 90)\} \subset \mathcal{D}_\mathrm{init}$. 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 $\mathcal{D}_\mathrm{init}$ 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 function 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. 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 of 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 highlighted 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:

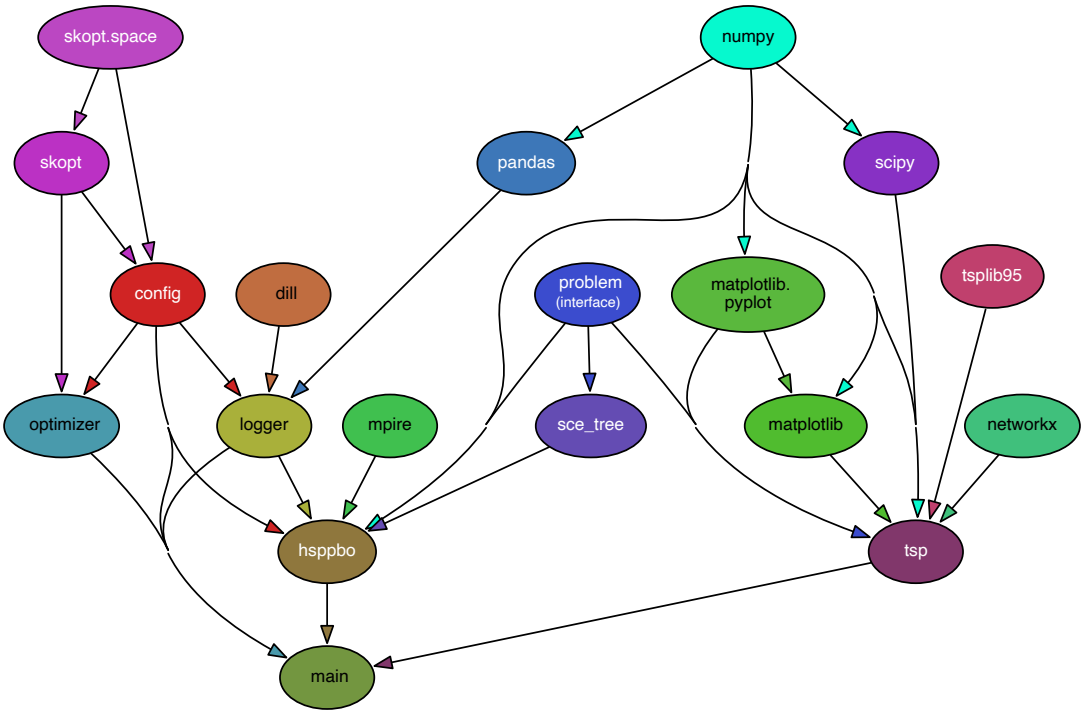


Figure 4.1: Dependency graph of the *XF-OPT/META* python software package

- 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 $H = \{H_{\text{full}}, H_{\text{partial}}, \emptyset\}$
- A number of iterations to pause detections for $L_{\text{pause}} > 0$
- 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 [10]. 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 $\frac{1}{n-1}$ where n is 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 may 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* [83] 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.

```
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 [84] 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` [85]. 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 provides 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 [8]. 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` [83]. Since all solutions generated by the `tsp` module are of the same type, they share a solution quality function $f : V^n \rightarrow \mathbb{R}_0^+$. 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.* [10] 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

This means that starting at iteration $i_{\min}$, every T_d iterations ($i_{\min} + k \cdot T_d < i_{\max}, k \in \mathbb{N}$) a number of $\frac{n \cdot C}{2}$ distinct pairs of cities are randomly selected and their entries in the distance matrix D are swapped. 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 [86, 87, 88]. 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 [89] (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 [4] were reviewed and checked for potential use with metaheuristics. Ultimately, the `scikit-optimize` library [90] 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` [91], 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 HPO 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 saves the complete results, including the trained regression model, in a so-called "pickle" file using the dill package [92]. 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 method (`skopt.BayesSearchCV`) 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 λ, problem parameters (t, p, w_{di})
$\mathbb{L} \leftarrow$ InitLogger( )
$\mathcal{P} \leftarrow$ InitProblem(t, p, w_{di})
$\mathcal{M} \leftarrow$ InitHSPPBO$(\mathcal{P}, \mathbb{L}, \lambda)$
$\mathbf{s} \leftarrow$ Execute$(\mathcal{M}, \mathcal{P})$
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 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 can be obtained again. 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_{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_{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

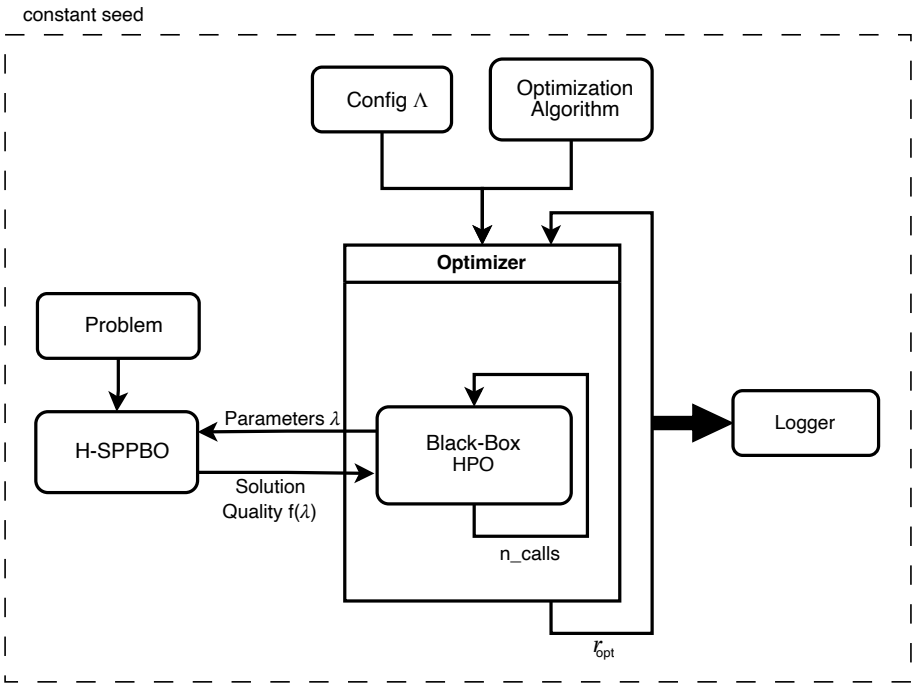


Figure 4.2: Visualization of the optimizer mode workflow.

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. For this reason, the random seed is not explicitly set to a fixed value in this mode. See Algorithm 4.3 for an implementation template.

Algorithm 4.3 XF-OPT/HSPPBO: Experimentation Mode

Require: Number of runs r_{exp}, parameter configuration λ, 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_{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 Kupfer *et al.* [10]. 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 [8], 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 also being repeated multiple times (r_{opt}) for each dynamic configuration and problem instance $\mathcal{P}$, the maximum dimension used is 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 [83]:

- 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 [93]. 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 [86]. 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 [88]:

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

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 [88]:

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

The area A covered by the nodes, i.e. the convex hull of the graph, was obtained using the *SciPy* library [94], which provides algorithms for scientific computing in *Python*. Although not originally applied to the TSP, the works of Crişan *et al.* [87] and Dry *et al.* [88] 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 [95], the "gap" between the first two eigenvalues could be used as a measure of connectivity [96].

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.* [87] and Dry *et al.* [88], additional TSP instances from the University of Bonn's *Tnm* test data set [97] 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 [88]. 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 [96], 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 matrices used here are not. However, applying the TSP-related approach of Cvetković *et al.* [95] 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 [98]. This NP-hard problem was originally solved by Lloyd [99], resulting in “Lloyd’s algorithm”,

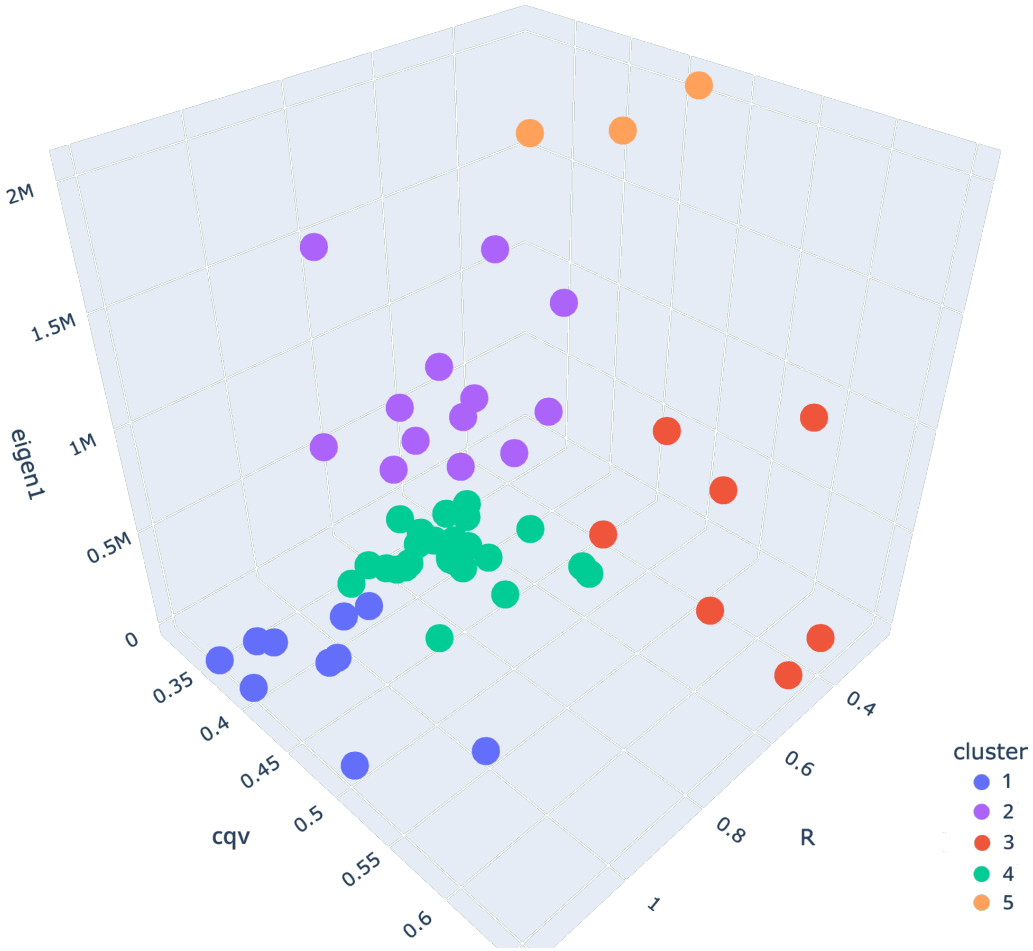


Figure 5.1: Visualization of the k-means cluster analysis applied to 118 TSP instances using $k = 5$ clusters. Each data point is an instance placed into three-dimensional space using its regularity index R, its first eigenvalue (eigen1), and its coefficient of quartile variation (CQV).

or "Voronoi iteration". 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 [98] 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 attached to 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 of 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:

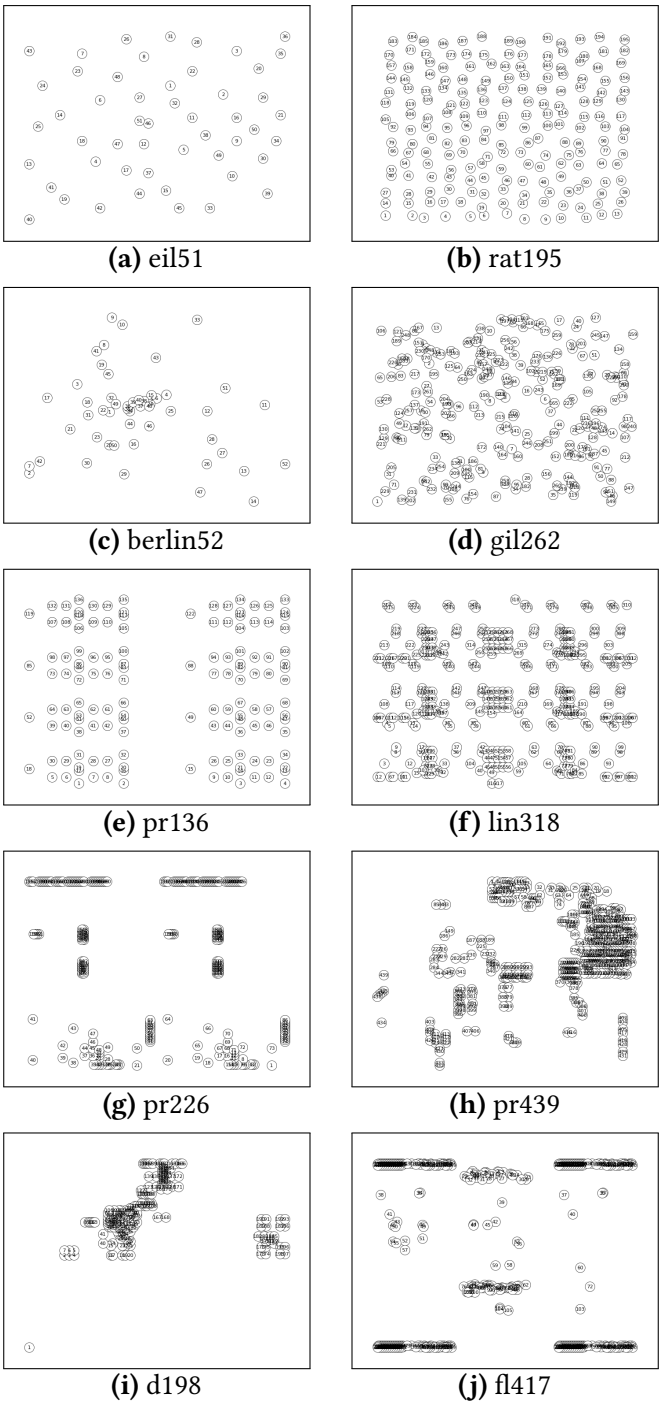
1. eil51, rat195
2. berlin52, gil262
3. pr136, lin318
4. pr226, pr439
5. d198, fl417

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 [49], who have a very similar application, using HPO to tune an Ant Colony Optimization algorithm. However, they focus only on BO using 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 on 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 $\mathcal{D}_{\text{init}}$
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 ($\mathcal{D}_{\text{init}}$) has been sampled using a specified method before the acquisition function was utilized for sampling, which is the default value of the library. The RS algorithm has only this sampling method as a possible initialization metric. 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 [49] 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, lower 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 [49]. 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 [100].

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" [101, 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)
- $H = \{H_{\text{full}}, H_{\text{partial}}, \emptyset\}$ (categorical)
- $0 < L_{\text{pause}} < 100$ (integer)

Unfortunately, these standard data ranges were too broad to use in the experiments. 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 [29] with their hierarchical version of a Particle Swarm Optimization (PSO) algorithm. Regarding the parameter H, Kupfer *et*

al. [10] 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. 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 [10]. 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_{pause} 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 dynamic 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 dynamic event. 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 [10], 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 [34], 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 [32, 37, 38] 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 $w_{\text{persprev}}, w_{\text{persbest}}, w_{\text{parentbest}}$ present an interesting significance. Although they are specific to this algorithm, they are based on the work by Lin *et al.* [7], 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 [7], a global population is introduced into the algorithm, and the total weight is divided by the number of iterations k for 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 [7]. 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:

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

- $L_{\text{pause}} = 5$ (not optimized)

5.4 Testing Procedure

The basis of each test was the H-SPPBO algorithm implemented as explained in Chapter 4, running $i_{\max} = 2600$ 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 event between iterations 2500 and 2599, all solutions just before each next dynamic change are stored. This means that between iterations 1999 and 2599, a total of seven solutions are stored, which are computed to solution qualities (lengths) and then averaged, resulting in $\overline{L}$. Thus, we are not only able to evaluate the dynamic response for all six dynamic events, but can also include the solution quality of the static TSP up to iteration 1999, albeit with a small weighting of 1/7 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, three optimization runs were performed for each of the four optimization algorithms (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, the test was limited to the smaller TSP instances and the medium dynamic intensity of 0.25. Furthermore, the BO process was limited to only 30 objective function calls instead of 50 or more, which means that a convergent behavior should have started after about 20 calls [102]. This de facto requirement for fast convergence can also be seen as a demand on the HPO method chosen. 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, six optimizer runs r_{opt} were executed 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, the larger instances were not used 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 $\mathcal{D}(\mathcal{A}, p, C, r_{\text{opt}}) \rightarrow \{\mathcal{H}(\lambda, f(\lambda)), \lambda^*, \mathcal{M}\}$, where $\mathcal{D}^i_{\mathcal{A}_j, p_k, C_l} \in \mathcal{D}(\mathcal{A}, p, C, r_{\text{opt}})$ and $1 \leq i \leq r_{\text{opt}}$.

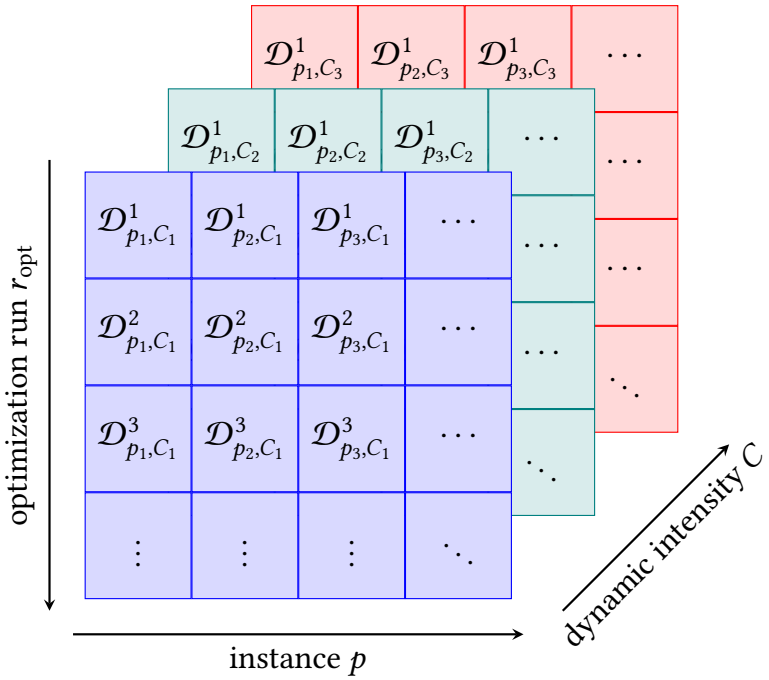


Figure 5.3: 3D tensor representation of the data set $\mathcal{D}_{\mathcal{A}=GBRT}(p, C, r_{\text{opt}})$ of the second experimentation part.

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 inclination to aggregate these parameter configurations in some way (e.g., an average of all three dynamic intensities for each problem category) is tempting, not only because it would simplify the next part, but also because it would 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 subsection, was 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

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

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.

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. Therefore, each of these 15 parameter sets is used two times, for a total of 30 different combinations of problem instance and dynamic. 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.* [10], 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).

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

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.

In general, since TSP instances from the *TSPLIB* were used, the optimal solution for each of these instances L^* is known. 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 of pairs $\{(x, y) \,|10 < x \leq \texttt{n\_calls}\}$, 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 [103] 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" [104] and the alternative H_1, rejecting H_0 at the 0.05 significance level, 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, R_i is the sum of all ranks for the i-th sample. The significance (p-value) can then be acquired through a table or by using a software package. The thesis is using the *SciPy* implementation of the Kruskal–Wallis Test, which approximates its p-value using a χ^2 distribution.

Conover–Iman Test

Conover and Iman [103] 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 ($Var(X) \lesseqgtr Var(Y)$). 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 [104, 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 [105]. 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 $\mathbf{\Lambda}$ [81, 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. In brief, this can be

understood as averaging out the influence of the other parameters. 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 [78].

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 [106].

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 [10], 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 their harmonic mean, the so-called F_1 score, were computed over certain combinations of data groups using the following equation:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{5.9}$$

A correct classification of a true positives (TP), i.e. a change handling procedure reacting to the dynamic event, is counted as such, if H was triggered within the first $L_{\text{pause}} = 5$ iterations of the dynamic event. For example, after the first dynamic event at iteration 2000, the execution of H was counted as a TP if it was triggered at iterations 2000, 2001, 2002, 2003, and 2004. After this time, the detection pause L_{pause} looses effect, and any further trigger of the change handling procedure was counted as a false positives (FP). This experimentation data was evaluated for all dynamic iterations, resulting in 30 iterations where the trigger of H counts as positive and 570 iterations counting as false trigger-points. Figure 5.4 shows an illustration of what is classified as a successful detection, or TP in all of the 600 dynamic iterations after the iteration 2000, indicated by the full rectangle. Each dot, whether filled or not, represents one iteration. Realistically,

the left rectangle would contain only six dots because there are six dynamic events, while the right rectangle would contain 570 dots, accounting for the subtraction of five iterations of pause L_{pause} after each dynamic event. The circle depicts all iterations where H-SPPBO detected a dynamic change and therefore triggered the change handling procedure. However, only the green semicircle on the right contains all iterations, where the problem actually changed, the TP. Now, the precision is the fraction of iterations that expresses how many detected iterations are actually true positives - the green semicircle of the full circle. And recall is the fraction of iterations that express how many true positives were detected out of all possible dynamic events - the green semicircle of the left rectangle. These two metrics are also displayed in Precision-Recall (PR) plots, to visually evaluate the accuracy of the change detection mechanism.

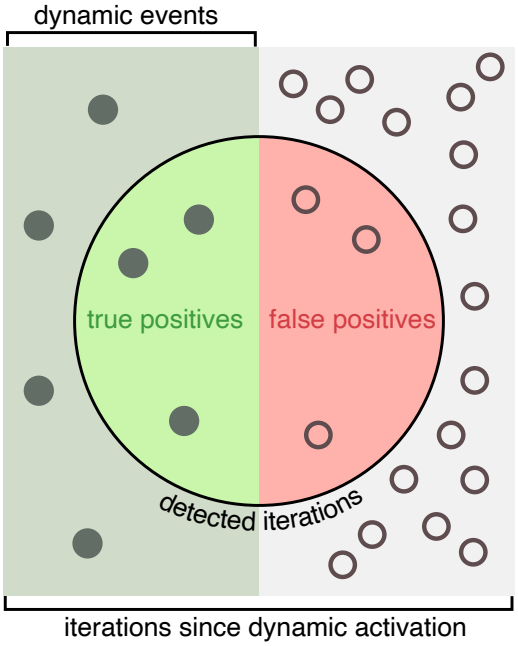


Figure 5.4: Illustration of iteration classification for dynamic problems. Each dot represents one iteration between 2000 and 2600. The circle depicts all iterations where H-SPPBO detected a dynamic change and therefore triggered the change handling procedure.

6 Results and Evaluation

All experiments and analyses were performed as previously described 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.

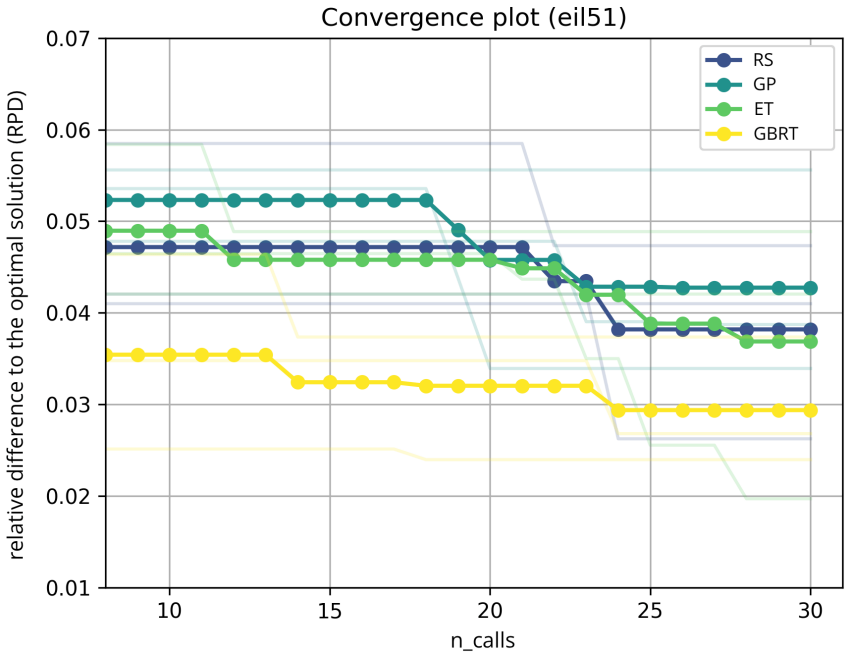


Figure 6.1: Convergence plot of TSP instance `eil51`, comparing four different optimization algorithms.

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 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`.

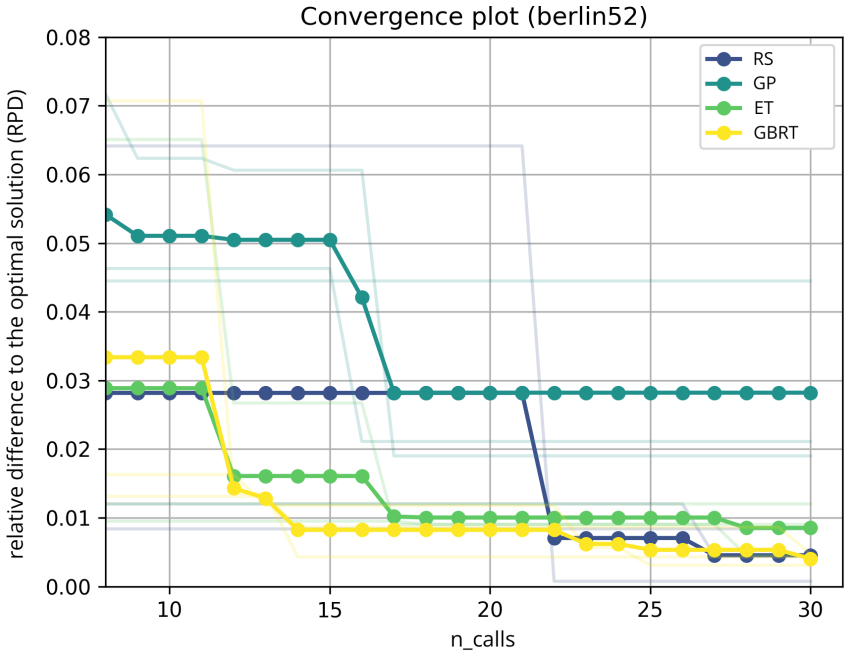


Figure 6.2: Convergence plot of TSP instance `berlin52`, comparing four different optimization algorithms.

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 a final 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.

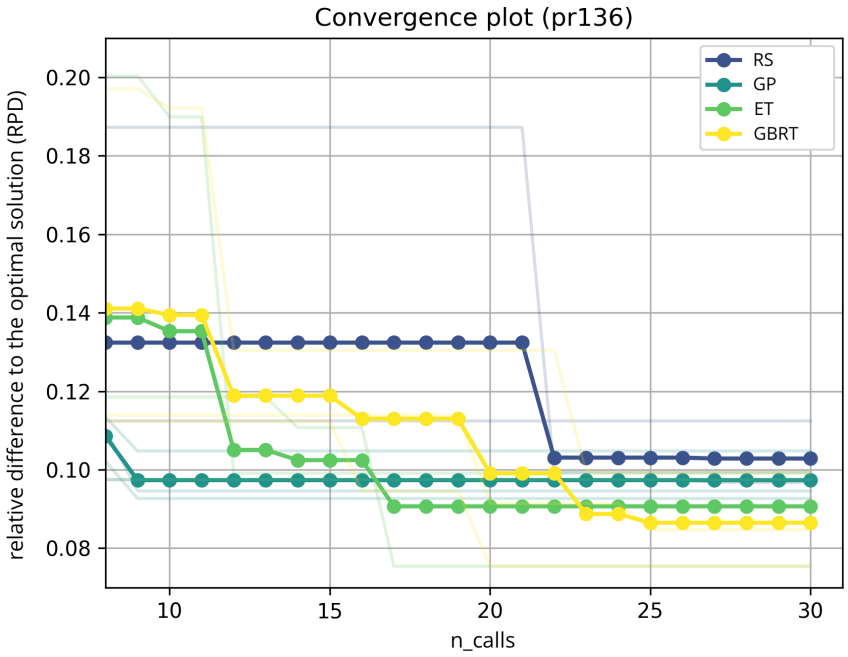


Figure 6.3: Convergence plot of TSP instance `pr136`, comparing four different optimization algorithms.

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`.

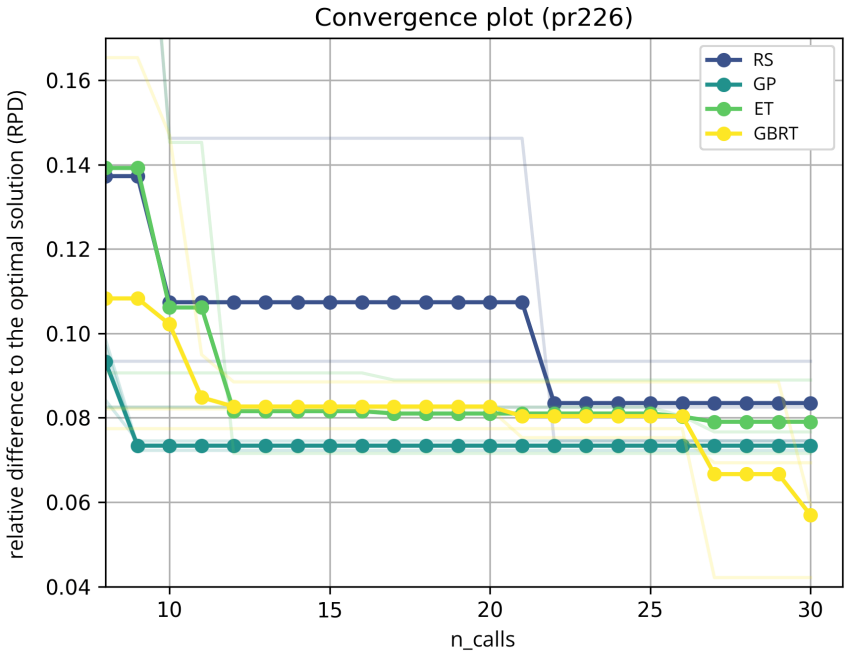


Figure 6.4: Convergence plot of TSP instance `pr226`, comparing four different optimization algorithms.

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 $RPD = 8\%$. Apart from the last five objective calls GBRT showed a similar convergence behavior, improving by more than 2% near the end, 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 potentially 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 of 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

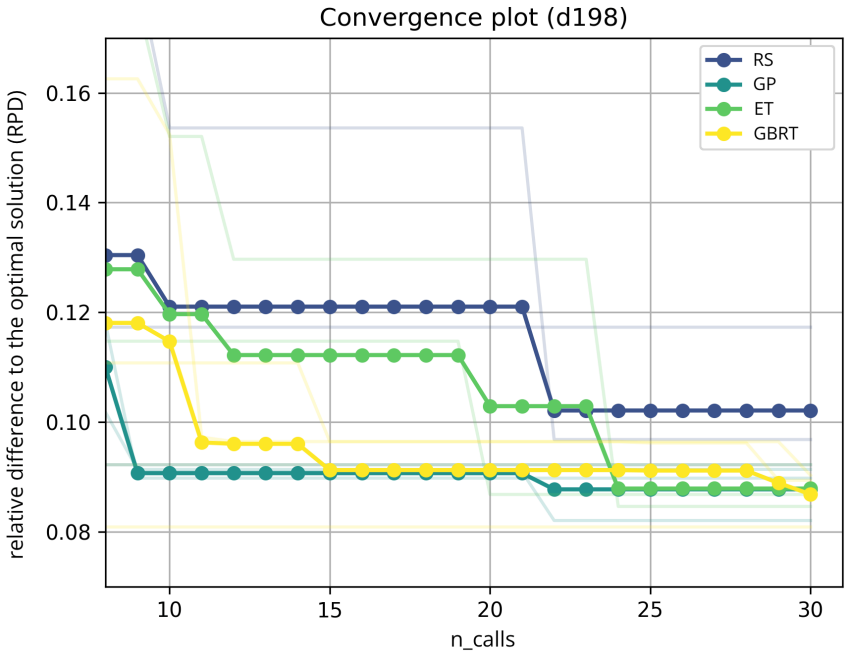


Figure 6.5: Convergence plot of TSP instance `d198`, comparing four different optimization algorithms.

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 methods 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 from previous descriptions, 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, averaged over all available optimizer runs, for all five instances, and their mean. In this context, a low AUC value indicates that the optimization algorithm found a well-performing parameter set, and therefore a low *RPD*,

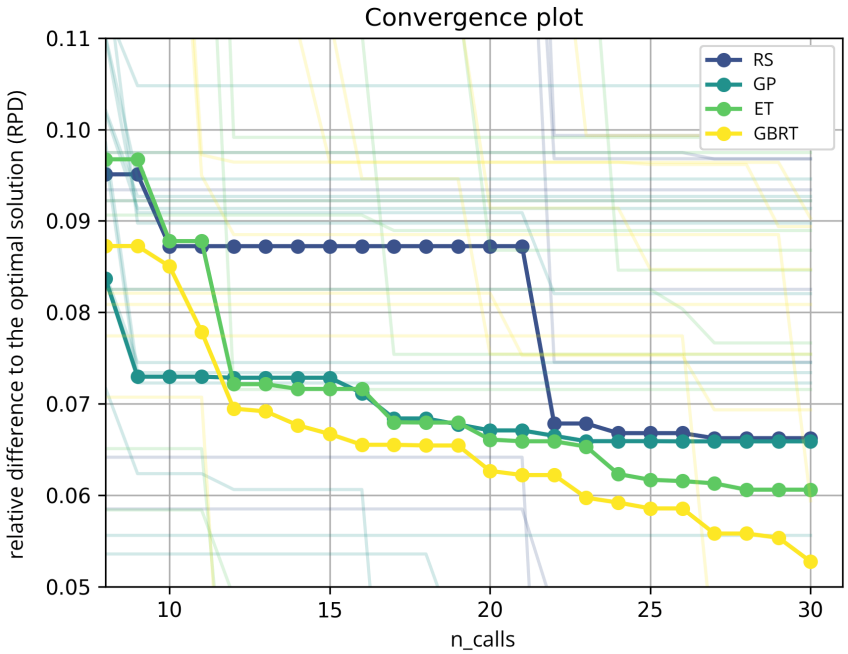


Figure 6.6: Convergence plot of the average over five TSP instances `eil51`, `berlin52`, `pr136`, `pr226`, `d198`, comparing four different optimization algorithms.

in a short time, i.e., few objective calls. However, the comparatively lowest AUC does 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 with 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.

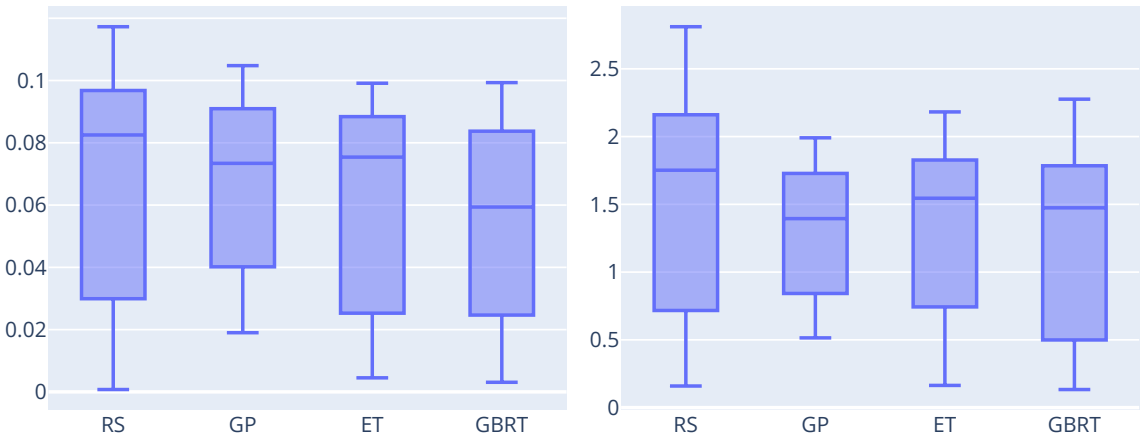
	eil51		berlin52		pr136	
	AUC	RPD_{min}	AUC	RPD_{min}	AUC	RPD_{min}
RS	0.830	0.026	0.347	0.001	2.266	0.097
GP	0.900	0.034	0.650	0.019	1.850	0.093
ET	0.819	0.020	0.227	0.005	1.809	0.075
GBRT	0.601	0.024	0.159	0.003	1.948	0.075

	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 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 data from all problem instances combined. 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.

Howeer, 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 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

The second part of the experiments was performed using only the Gradient Boosted Regression Trees (GBRT) optimization method, but this time with twice the objective calls and for all dynamic intensities applied to the smaller instances. Unlike the previous part, there was no need to manually select a "winner" based on the results, since the HPO procedure provided six parameter sets and a corresponding solution quality (*RPD*) for each of the 15 combinations of problem instance and dynamic intensity C (see Table 5.2).

From these, the best performing set with the lowest tour length L was selected and is shown in Table 6.5. The parameters were explicitly chosen by this quantifiable procedure to free the further analysis from any assumptions about the the parameter aggregation process other than solution quality. While it would have been possible to average each parameter set over its six optimizer runs, or even over different dynamic intensities or problem instances, this would have introduced many unknown influences and would have allowed speculation as to which averaging method would be the most beneficial.

Table 6.5: Best parameters from all six optimizer runs for all 15 combinations of TSP instance and dynamic intensity C from the second part of the experiment.

TSP	C	α	β	w_{persbest}	w_{persprev}	$w_{\text{parentbest}}$	θ	H
eil51	0.1	1	5	0.970	0.497	0.942	0.391	partial
eil51	0.25	2	9	0.143	0.239	0.541	0.364	full
eil51	0.5	2	10	0.059	0.960	0.487	0.460	full
berlin52	0.1	3	9	0.045	0.478	0.196	0.276	partial
berlin52	0.25	2	8	0.831	0.222	0.313	0.436	full
berlin52	0.5	1	9	0.249	0.774	0.972	0.388	partial
pr136	0.1	2	8	0.094	0.008	0.469	0.386	partial
pr136	0.25	2	10	0.038	0.687	0.939	0.390	partial
pr136	0.5	2	9	0.073	0.325	0.667	0.139	partial
pr226	0.1	2	10	0.112	0.051	0.436	0.183	partial
pr226	0.25	2	9	0.488	0.471	0.939	0.240	partial
pr226	0.5	2	9	0.428	0.153	0.984	0.291	partial
d198	0.1	2	9	0.002	0.974	0.652	0.212	partial
d198	0.25	2	10	0.670	0.090	0.892	0.386	full
d198	0.5	2	10	0.694	0.217	0.666	0.216	partial

These best parameter results alone would have been sufficient to continue with the validation in the third part, as HPO is intended to enhance and accelerate this parameter tuning process. However, there is much information to be gained from this data set in terms of robustness, correlation between parameters and with problem descriptions, and relative parameter importance. All of this provides insight into what most influences the HPO process, and thus the performance of the H-SPPBO metaheuristic.

6.2.1 Robustness of Parameter Values

First, we look at how often a particular value for a given parameter was chosen by the HPO and how reliable that choice is - in short, the parameter robustness and dispersion. To do this, statistical box plots were generated over all 90 available optimizer runs, each containing an optimal parameter set. Each plot shows all H-SPPBO parameters except the dynamic reaction type, which has only two possible categorical values. Instead, a separate analysis for this parameter is provided below. For each parameter's box plot, the interquartile range (IQR) is immediately visible as the large rectangle with a bold outline around it. A small IQR implies a low dispersion and therefore a robust parameter selection by the HPO. The median (Q_2) is represented by the dividing line in the middle of the rectangle. The lower (Q_1) quartile line is the bottom line of the rectangle and marks the value below which 25% of the data falls, and the opposite upper (Q_3) quartile line shows the value above which 25% of the data falls. Each of these two resulting regions (upper from Q_2 to Q_3 and lower from Q_2 to Q_1) contains the same number of data points. The whiskers at the top and bottom represent the upper (Q_4) and lower (Q_0) fence parameter values, respectively. Outliers are marked as dots, if their value exceeds $1.5 \times IQR$ from the upper or lower fence or if they are $3 \times IQR$ above Q_3 or below Q_1. All of the underlying statistical values can also be found in the tables in Appendix B.

Starting with the box plot, which is not grouped by any metric, Figure 6.8 shows, in addition to the information already described, all 90 values for each parameter as scattered dots next to the box itself. The results for α suggest that it is the most robust parameter of the six, with the median falling together with the upper quartile at 2, and the minimum and lower quartile sharing their value at 1. A value of $IQR = 1$ is remarkably low and can only be improved later on in the analysis, when looking at aggregated box plots. The GBRT surrogate model seemed to have no problem finding appropriate values for α, and since a value of zero was never chosen, a floor effect can be ruled out. The β shows a much larger spread around its median of 9 and has a very low fence of 5. However, its IQR of 2 suggests that the HPO process was mostly consistent in finding values for this parameter. The abrupt end of the upper quartile and the maximum at 10 suggests a ceiling effect (see Section 5.3), which was supposed to be avoided by a sufficiently large value range. Apparently, an upper limit of 10 was not enough for β. However, since at least 50% of the values are at or below 9, this effect should not have had a significant influence on the choice of this parameter during Hyperparameter Optimization.

The three weights, w_{persbest}, w_{persprev}, and $w_{\text{parentbest}}$, on the other hand, show considerably high dispersion. In particular, the box plot for w_{persbest} has an $IQR = 0.72$, which is almost as large as the possible value range from 0 to 1, with the whiskers reaching these boundaries. Therefore, the median of 0.37 has almost no significance in choosing an optimal parameter based on means alone. This situation improves slightly when looking

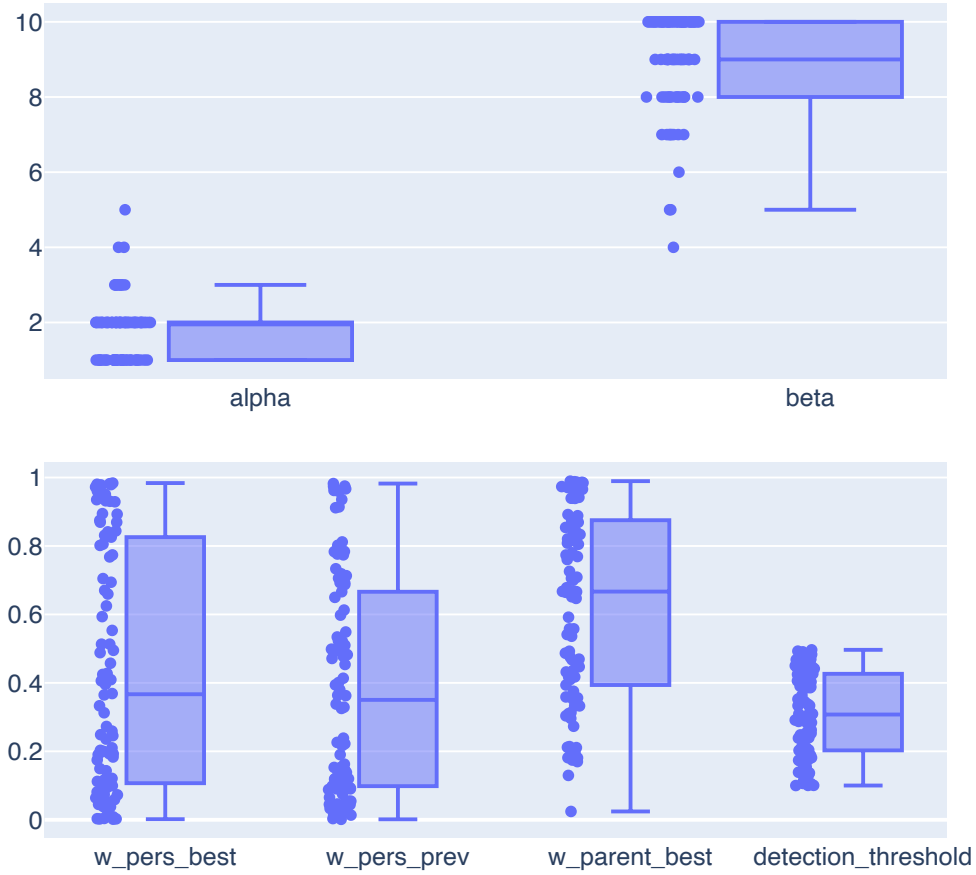


Figure 6.8: Statistical box plot of all H-SPPBO parameters (except H) over all 90 optimizer runs.

at the remaining two weights. Although their box plots also span the entire value range, their IQR is at about half of the range ($IQR_{\text{persprev}} = 0.57, IQR_{\text{parentbest}} = 0.48$). Therefore, the medians of w_{persprev} at 0.35 and of $w_{\text{parentbest}}$ at 0.66 can both be considered somewhat robust. However, the fact that all three of these weights are used together in a sum makes any value suggestion an almost random one. Only the absolute parameter configuration range between 0 and 1 and a vague specification of this range for the latter two weights can be confirmed to some extent.

The box plot for the dynamic detection threshold θ shares its y-axes with the three weights, but only has a possible value range between 0.1 and 0.5. Therefore, its IQR is quite large at 0.22 or more than half of its range. The same dispersed behavior can be seen with its whiskers, which reach their minimum and maximum values almost exactly to the third decimal point. Without any kind of grouping applied to the data, no recommendation can be given for the detection threshold and regarding the HPO process, this parameter could not be chosen reliably.

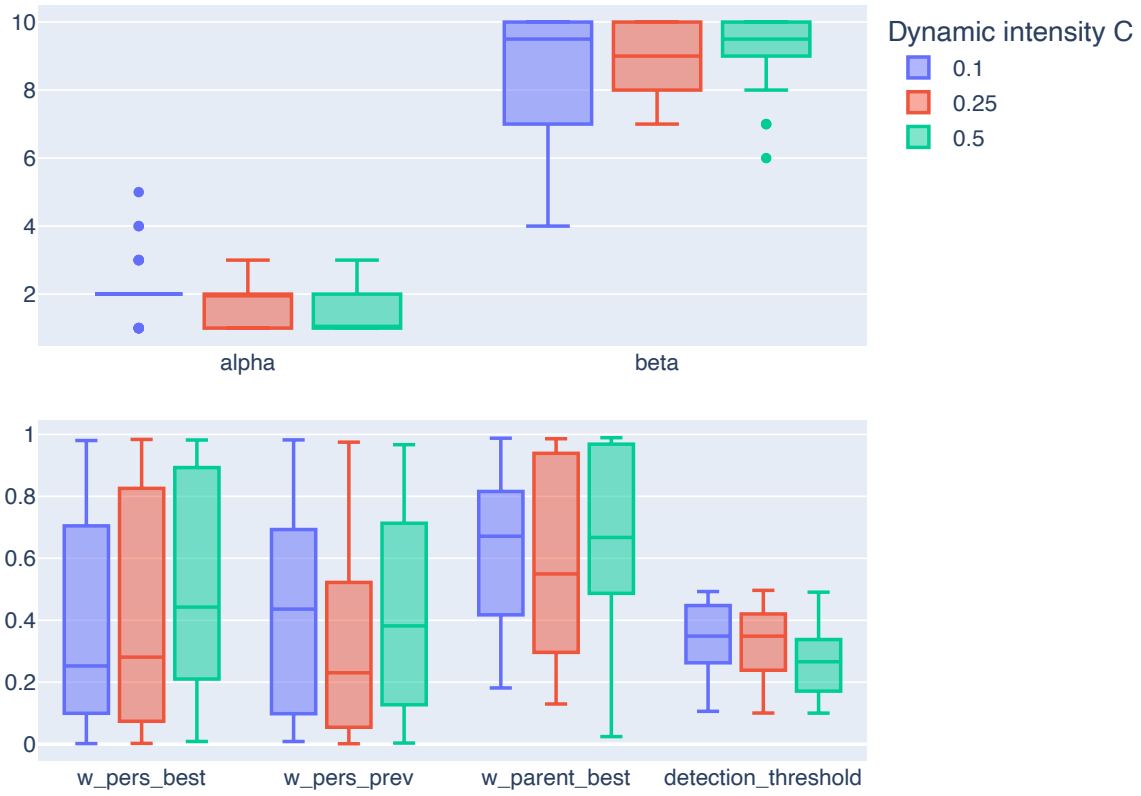


Figure 6.9: Statistical box plot of all H-SPPBO parameters (except H) over all 90 optimizer runs and compared by dynamic intensity C.

Continuing with the box plots shown in Figure 6.9, where the data points are grouped by the three dynamic intensities C tested, an interesting trend emerges for α and β. For a low dynamic intensity, i.e. very few cities swap places every $T_d = 100$ iterations, an α value of exactly 2 seems to be the ideal choice for the HPO process, with an IQR of 0 and only four outliers in total. Parameter β, on the other hand, shows a similarly low dispersion for the highest dynamic of $C = 0.5$. With an $IQR = 1$ and a median of 9.5, a strong heuristic influence seems to be beneficial in obtaining good solutions in highly dynamic problems. In the case of α, the other two higher dynamics introduce more spread in the choice of values, but always remain in a median range between 1 and 2, with a small dispersion of 1. For β, the robustness decreases significantly with lower dynamic intensity, reaching a high $IQR = 3$ for $C = 0.1$.

As with the ungrouped box plots, the three weights show no sign of robust value behavior. Although small improvements can be observed for certain weights under certain dynamic intensities, e.g., $w_{\text{parentbest}}$ has a comparatively low dispersion of $IQR = 0.4$ and a reduced minimum for $C = 0.1$, the spread is still far too large to provide any value suggestions under certain dynamic environments. Aside from the robustness, the two personal weights, w_{persbest}, w_{persprev}, have a particularly different median for the lowest dynamic

of $C = 0.1$, although the medians for the other dynamic intensities are quite similar. This may be due to the fact that, especially in a low dynamic environment the populations for both personal weights become equal after a certain number of H-SPPBO iterations. Therefore, a reset of the personal best population (whether partial or full) still gives the SCE a copy of that solution through the personal previous solution, which then influences that weight to be more important during HPO.

The same statements about robustness can be made for the dynamic threshold θ, where an improvement in the IQR of about 0.05, which is 12.5% of its value range, still does not allow for a sophisticated parameter choice. The only hinted trend might be, contrary to intuition, that lower values of θ, i.e. less SCE tree swaps necessary to trigger the change handling procedure, benefit solving higher dynamic problem instances ($C = 0.5$). However, with a median of 0.27, the upper quartile region still completely overlaps with the other two groups.

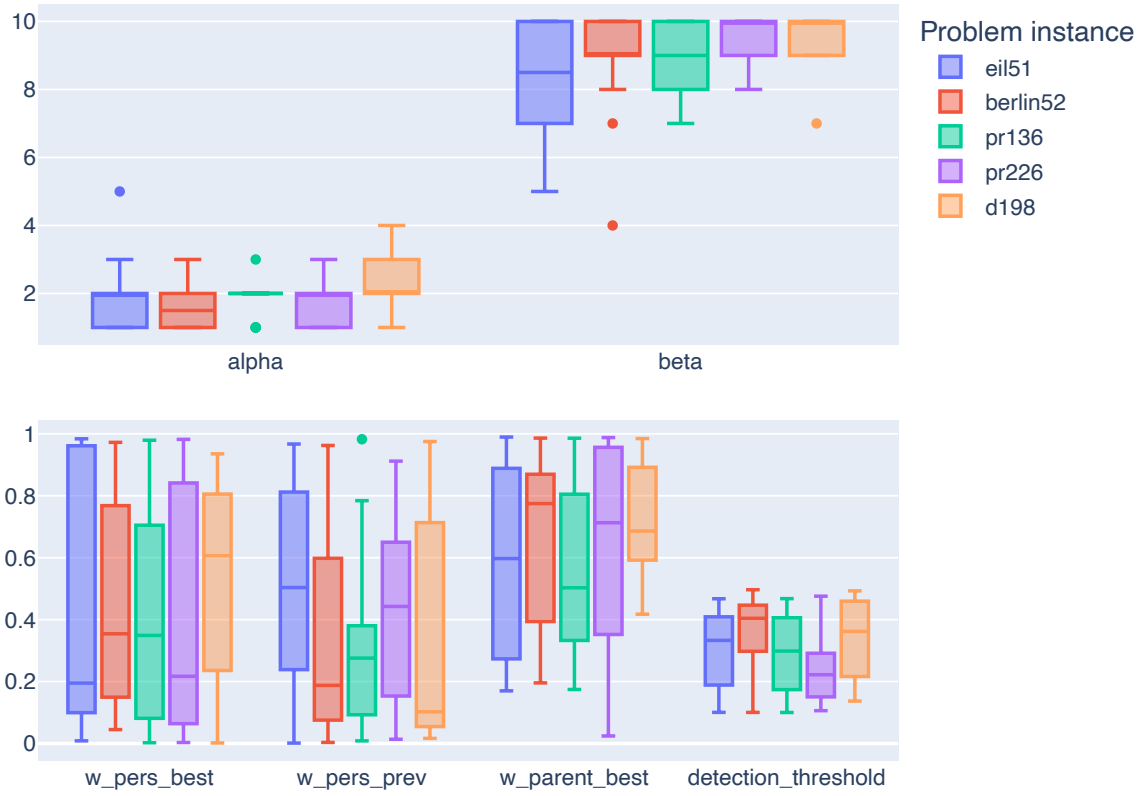


Figure 6.10: Statistical box plot of all H-SPPBO parameters (except H) over all 90 optimizer runs and compared by problem instance.

Figure 6.10 shows the parameter box plots grouped by the TSP problem instance on which they were acquired on. The parameter α shows a low dispersion for `pr136`, an instance with a moderately clustered structure, and an unusually high dispersion for `d198`, an instance with highly clustered areas. The other problem instances show similar

results as before, and the median for all five instances is still between 1.5 and 2. The same conclusions can be drawn for β, where the instances `berlin52`, `pr226`, and `d198` show low IQRs of 2, while the remaining two instances, especially `eil51`, show very unreliable values with medians as low as 8.5.

The three weights show very dispersed distributions, as expected from previous observations, with `eil51` probably being the most spread out of all values. Only some instances have an improving effect on the robustness, e.g. w_{persprev} for instance `pr136` with a low $IQR = 0.29$ and a median of 0.28, or $w_{\text{parentbest}}$ for instance `d198` with $IQR = 0.30$ and a median of 0.69. Also, compared to the values from the previous figures, $w_{\text{parentbest}}$ seems to be much more robust across all instances (except `pr226`), suggesting that this value could be very dependent on the particular problem it is used for. The detection threshold θ shows in some instances, i.e. `berlin52` and `pr226`, a more robust value selection with IQRs around 0.15, which is even better, than the aggregation by dynamic intensity shows. However, these two robust values are very different, with medians of 0.40 and 0.22, respectively, suggesting that dynamic detection may be strongly influenced by the problem instance on which it is used on. This is also seen in the data for the reaction type H, which is discussed in the following.

The dynamic reaction mechanism H was chosen to be discussed separately, because a two-valued categorical value is not effectively represented by a box plot. Instead, Table 6.6 shows the distribution across the same comparison groups as used in the box plots. In addition to the two grouped versions, distributions for only those parameter sets that resulted in 50% or better relative solution qualities (RPD) were added to represent what values the best parameter sets use for H.

It is immediately evident that the partial reaction H_{partial} is most often preferred by the Hyperparameter Optimization, except for some deviations. In the overall (ungrouped) comparison, twice as many parameter sets used this method of dynamic reaction. For the lowest ($C = 0.1$) and highest ($C = 0.5$) dynamic intensities, this preference can also be confirmed, while for medium intensity, H_{full} is also a viable choice 43.3% of the time. Except for a slight percentage decrease for H_{partial} with $C = 0.1$, this behavior does not change for the best 50% of solutions. When grouped by TSP instance, the only problem that does not favor a partial reset appears to be `d198`, where 61.1% of the parameter sets use the H_{full} method. Contrary to this observation, the HPO process chose the partial reset 83.3% of the time for `pr226`, which is the highest bias toward any H method. Looking at the best 50% of parameter sets, `eil51` clearly deviates from any preference and has both methods almost at 50%. The other instances also show a decrease in their bias towards H_{partial}, except for `d198`, which now seems to choose the full reset less often for its better solutions.

Table 6.6: The distribution of the H-SPPBO reaction type H. Either for all parameter sets or for parameter sets grouped by dynamic intensity C or problem instance, the percentages of these sets using one or the other reaction type are given.

Comparison	Group	$H_{\text{partial}}[\%]$	$H_{\text{full}}[\%]$
-	-	66.7	33.3
Dynamic Intensity	0.1	76.7	23.3
	0.25	56.7	43.3
	0.5	66.7	33.3
Dynamic Intensity Best 50% RPD	0.1	66.7	33.3
	0.25	53.3	46.7
	0.5	66.7	33.3
Problem Instance	eil51	61.1	38.9
	berlin52	72.2	27.8
	pr136	77.8	22.2
	pr226	83.3	16.7
	d198	38.9	61.1
Problem Instance Best 50% RPD	eil51	55.6	44.4
	berlin52	66.7	33.3
	pr136	66.7	33.3
	pr226	77.8	22.2
	d198	44.4	55.6

Parameter Value Recommendations

Based on the previous discussion of parameter robustness, their median values, and possible preferences and trends with respect to certain combinations of dynamic intensity and problem instance, some general recommendations for potentially well-performing parameter sets are given. These suggested value ranges could be used to refine the HPO parameter configuration or as guidelines for manual parameter tuning.

Starting with the two most robust recommendations, α and β, which had the lowest IQRs out of all parameter box plots. The median for α was almost always at 2, with boundaries rarely exceeding the interval $[1, 2]$. This was also the parameter that was least affected by a change in the problem description, making the choice of $\alpha = 2$ the most robust suggestion. The parameter β usually had a median between $[9, 10]$ and is slightly influenced by dynamic intensity and problem instances. Since in some combinations β was chosen as low as 4, with a lower quartile value of around 7, and considering the ceiling effect, a value range between $[7, 12]$ might lead to satisfactory solutions.

The dynamic reaction type H was also most often chosen as H_{partial}. The influence of different dynamic intensities also seems to be marginal, since only the medium dynamic ($C = 0.25$) leads to a change in the distribution, with the highest and lowest tested values still favoring the partial reset. This also holds true for the problem instances, where the results only suggest that certain city placement characteristics - very regular instances like `eil51` (group 1) or highly clustered instances like `d198` (group 5) - might have an influence on this choice. Nevertheless, the data indicates that the H_{partial} method is the preferred option.

The three weights, w_{persbest}, w_{persprev}, and $w_{\text{parentbest}}$, are not only agnostic to varying dynamic intensities and problem instances, but their choice also seems to be influenced by some other, possibly random, relationship. Regardless, their median values may still be useful as a symptom of the underlying well-performing parameter sets. The parameter w_{persbest} has a median between 0.28 and 0.44, and is only influenced by higher dynamics ($C = 0.5$) and certain, highly clustered, problem instances (`d198`). On the other hand, w_{persprev} seems to be strongly influenced by dynamic intensity and problem instance, resulting in a median between 0.10 and 0.50. And $w_{\text{parentbest}}$, which has the lowest IQR of the three, has its median range between 0.50 and 0.78, showing a slight dependence on dynamic intensity and a stronger influence from problem instances. Also, since this range is above the other two weights, it could be implied that the parent-best population might be slightly more significant to the solution construction procedure. These three median ranges can be used as value range suggestions and at least somewhat confirm the HPO configuration range between 0 and 1, with no strong floor or ceiling effect present in the data.

Finally, the detection threshold θ was slightly influenced by the dynamic intensity, with an interesting preference towards lower values of 0.27 for highly dynamic intensities of $C = 0.5$, and was strongly influenced by problem instances. Again, the configured HPO value range between 0.1 and 0.5 was confirmed in terms of the lower and upper quartiles never getting too close to these boundaries. However, the high median interval for the problem aggregated statistic of $[0.22, 0.40]$ still spans half of the available range, with the advantage that this suggestion still narrows the range.

In summary, the following parameter range configuration can be seen as a possible refinement to the one given in Section 5.3 to reduce the number of objective calls needed to find good parameter sets using HPO with GBRT for the H-SPPBO metaheuristic, regardless of dynamic intensity or TSP problem instance:

- $\beta \in \{x \in \mathbb{N} | 7 \leq x \leq 12\}$
- $w_{\text{persprev}} \in [0.10, 0.50]$
- $w_{\text{persbest}} \in [0.28, 0.44]$

- $w_{\text{parentbest}} \in [0.50, 0.78]$
- $\theta \in [0.22, 0.40]$
- $H = H_{\text{partial}}$ (not optimized)
- $\alpha = 2$ (not optimized)
- $L_{\text{pause}} = 5$ (not optimized)

In addition, this configuration should further speed up the optimization process and may even increase the quality of the solutions found by reducing the dimensions of the configuration space.

6.2.2 Parameter Interaction and Influences

Besides the analysis of the parameter robustness, another point of interest is the parameter interaction, not only among themselves, but also with respect to certain dynamic intensities. For this purpose, scatter matrix plots were generated showing every possible combination of two H-SPPBO parameters on the x- and y-axis, respectively, with an optional aggregation over dynamics. Since the resulting matrix would have been anti-symmetric, the upper right triangle was omitted to improve readability. The resulting assignment of a parameter to a particular axis, rather than showing the opposite paring as well, does not reflect any real dependence of one parameter on another. As with the previous box plots, the reaction type H was excluded not only for lack of information gain, but also to reduce the size of the matrix and make the figure legible on paper. When analyzing the scatter plots, the ungrouped and dynamic-aggregated figures are used together for each parameter pairing, rather than being discussed one after the other.

Starting with the ungrouped scatter matrix, Figure 6.11, and looking at the first parameter pairing, with values for α on the x-axis and values for β on the y-axis, we can see an interesting distribution towards the upper left corner of the plot. This suggest a correlation, with higher β values, between 8 and 10, allowing for higher α values, between 2 and 4. This confirms the relationship between these two parameters for many other metaheuristic parameter tuning procedures as well. However, some parameter sets with high β values between 8 and 10, also found small α values of 1 to produce favorable solutions. Furthermore, the region of high values for α and low values for β seems to be of little interest for good parameter sets.

Figure 6.12 shows yet another probable correlation due to the influence of different dynamics. In the case of overlapping points, the color of the dynamic intensity was displayed, which is the most common for this parameter combination. For α and β,

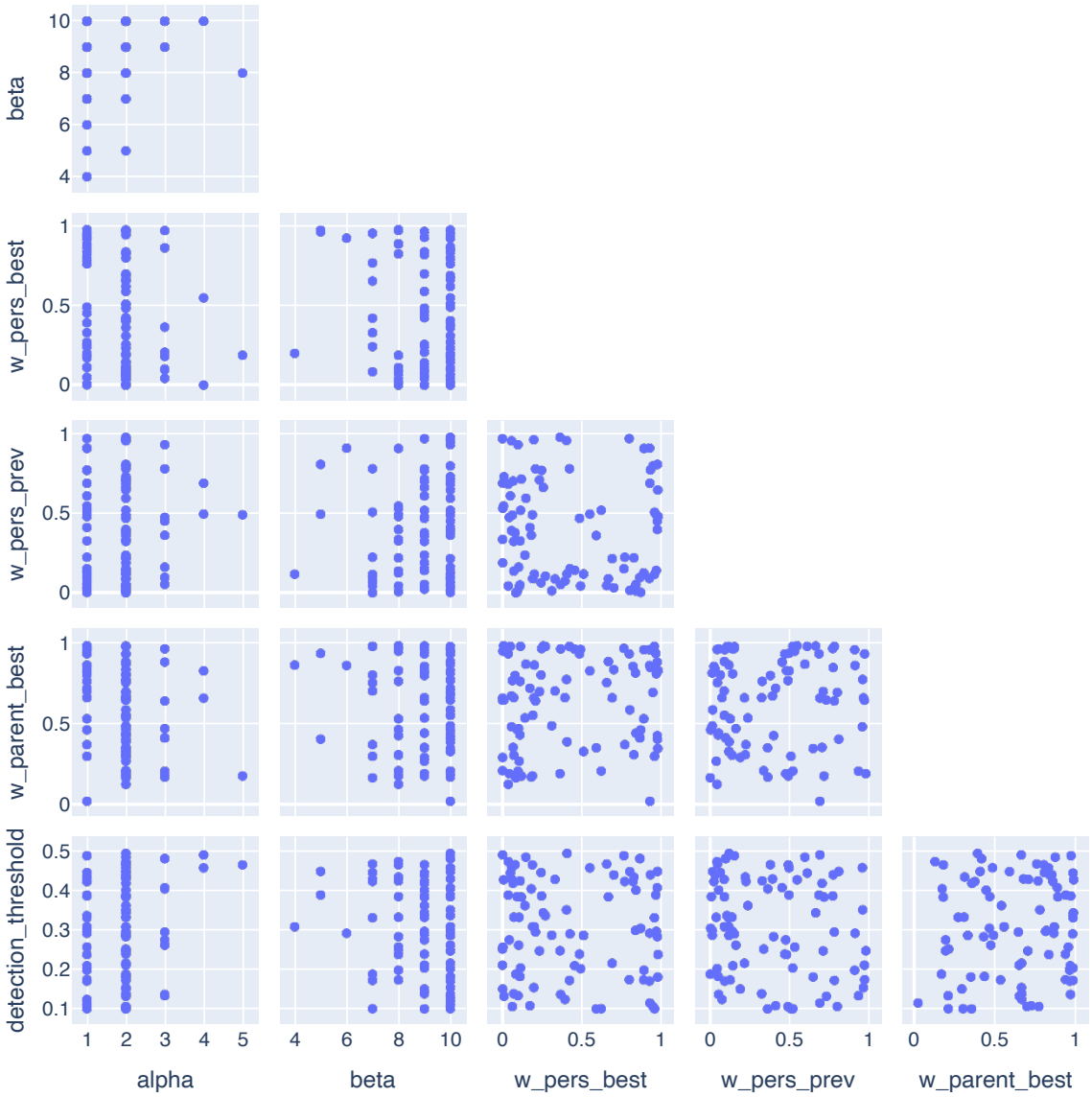


Figure 6.11: Scatter matrix plot of all H-SPPBO parameters (except H) over all 90 optimizer runs.

interesting enough, the previously preferred combination of high values of β allowing higher values of α most often applies to low dynamic intensities of $C = 0.1$, while the opposite correlation, $\beta \in [4, 5]$ and $\alpha = 1$, is also used for this dynamic. For the highest dynamic intensity of $C = 0.5$ the values for α were not chosen as high with increasing β, implying that the heuristic influence is relatively more important than the stochastic solution construction for highly dynamic problems.

Continuing down one row, we have w_{persbest} being paired with α and β. First, for $\alpha = 1$ there is an unusual, but significant gap for the value of w_{persbest} between 0.5 and 0.75, which increases even further for $\alpha = 3$, but inexplicably disappears for $\alpha = 2$. Similar gaps

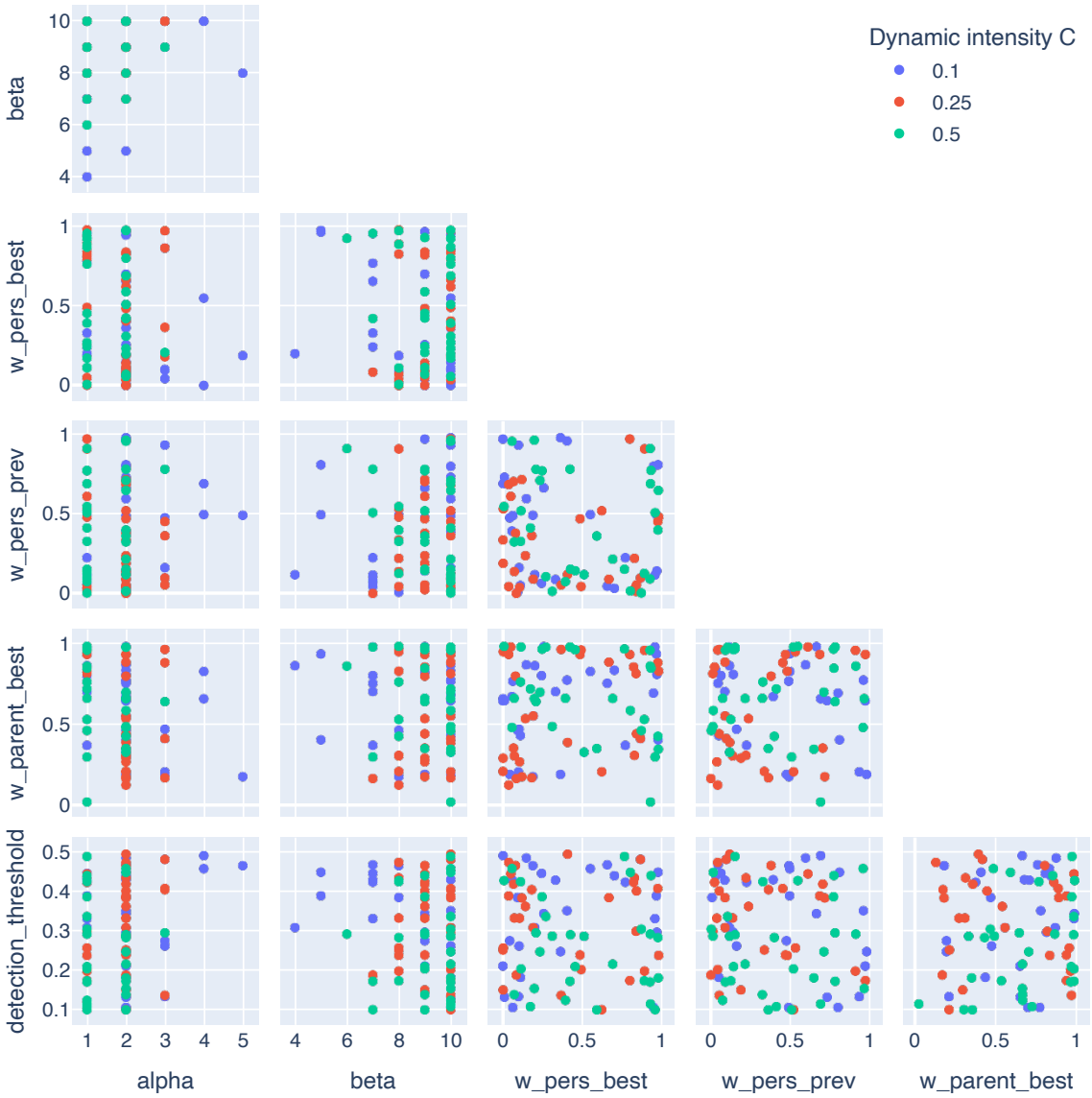


Figure 6.12: Scatter matrix plot of all H-SPPBO parameters (except H) over all 90 optimizer runs and compared by dynamic intensity C.

in the choice of values can also be found for $\beta = 8$. In the case of the dynamic comparison, no further trend can be found in the scatter plots, except for the value choices already discussed in the previous subsection.

For w_{persprev}, there are only value gaps for $\alpha = 3$ and $\beta = 8$. Especially for the pairing with β, there is a slight trend towards smaller values for w_{persprev} for smaller β values, with an interesting area for the lowest dynamic intensity of $C = 0.1$ and a $\beta = 7$ using unusually low values for w_{persprev} between 0 and 0.2. The combination of w_{persprev} and w_{persbest} shows a significant avoidance of a medium-value area where both parameters are around 0.5, with only four exceptions. Therefore, it is implied that these two parameters

are preferred in pairings near the value boundaries, with a small bias towards a low-low-combination. In particular, for a medium dynamic environment of $C = 0.25$, lower values for w_{persprev} below 0.2 are most often used with similarly low values for w_{persbest}.

The parameter $w_{\text{parentbest}}$ shows no clear trend when interacting with α or β. As with the interaction between the other two weights, the pairing with both w_{persprev} and w_{persbest} shows the same avoided mid-area around values of 0.5 for both parameter, although not as pronounced. The bias towards higher values for $w_{\text{parentbest}}$ above 0.2, especially at low dynamic intensities, can also be seen in these scatter plots. Another interesting area to note is present in the pairing with w_{persprev}, where a clear notch in the distribution is visible for $w_{\text{parentbest}} \in [0.8, 1]$ and $w_{\text{persprev}} \in [0.2, 0.5]$. However, this does not justify an indication of a possible correlation and could just be a random influence.

Finally, the detection threshold θ is paired with all other parameters. With $\alpha = 2$, there is a preference for smaller values of $\theta < 0.3$ for high dynamic intensities of $C = 0.5$. The opposite can be observed at low dynamics of $C = 0.1$ with smaller $\beta \leq 7$ that are being paired with θ values greater than 0.3. The interaction with any of the three weights does not show any obvious correlation or tendency towards certain areas.

For completeness, Figure A.3 in Appendix A also shows the ungrouped version of the scatter plot that includes the dynamic reaction parameter H. However, other than avoiding the full reset method (H_{full}) for low detection threshold θ values between 0.1 and 0.15, nothing noteworthy is added by this matrix.

In conclusion, only the interaction between α and β seems to suggest some kind of correlation between these parameters. In particular, since α is used as an exponent for the sum of the three weights, one might expect some kind of relationship between these parameters. But even when α was plotted in a semi-logarithmic scatter plot over the sum of all three weights (see Figure A.4), no particular distribution became apparent. Therefore, at least for this specific case of HPO using GBRT, there does not appear to be a direct relationship between these parameters. The same can be said for any other combination of parameters where no zero-order correlation is apparent. However, a higher order correlation across multiple parameters may still be possible and could be further investigated with a different test setup that also samples parameter areas that lead to sub-optimal solutions, i.e., that is not driven by HPO goals.

Partial Dependence

Although we have already discussed the abundance of strong correlations between parameter pairings, the following partial dependence plots, as described in Section 5.5.2, take advantage of the predictive capabilities of the final surrogate model generated during

HPO. Since each of the 90 optimizer runs produced one model, only the model with the best solution quality was used, following the same reasoning as for the selection of the optimal parameter sets described at the beginning.

For each parameter (dimension), 60 points were evaluated across their value ranges, with each point averaged over 250 prediction samples. In the one-dimensional case, shown in the diagonal line plots, this allows for an analysis of the effect of a single dimension on the predicted objective function, i.e., which parameter values most affected the output of the surrogate model. In the two-dimensional case shown below the diagonal, this effect is shown for a combination of two parameters using contour plots. Light colored (yellow) areas represent better solutions (smaller prediction for tour length L), while dark colored (blue) areas represent worse solution areas. The black dots represent the observed points during the HPO, and the red star in each contour plot marks the best observed parameters. Due to the qualitative nature of these visualizations, and since this is only a strong estimate of what might be really important for the actual H-SPPBO metaheuristic, no explicit quantitative color-map is used, and the PD values on the diagonal one-dimensional plots are to be understood only as relative orientation.

Due to the size of these partial dependence plots, not all 15 possible figures are shown in the following, but only the two most interesting ones, showing the results for the problem instance `berlin52` and for dynamic intensities $C = 0.25, 0.5$, are discussed. However, these two chosen figures are most representative, especially in the context of the analysis from previous subsections, and necessary remarks for the other unused figures will be included if necessary. These other 13 figures are also available in the GitHub repository of this thesis.

Figure 6.13 shows the partial dependence for $C = 0.25$. Starting from the diagonal, a low value for PD, i.e. a good solution quality, was obtained by the model for α values between 1 and 4, with a steep edge around $\alpha = 5$. The solution quality also seems to improve dramatically for higher values of β, with $\beta \geq 8$ as the optimum. The resulting combination of the two, shown in the contour plot to the left and below these two one-dimensional plots, confirms similar value areas as described previously in Figure 6.11, where the light-colored, good areas are in the upper left corner of the plot, i.e., the high β, low α value range. The opposite dark blue corner also confirms that this area of high values for α and low values for β also leads to predicted bad solutions for the model, which is why this area was never part of a good parameter set. This observation for α and β can be made for almost all of the other 14 partial dependence plots, with the only difference that α sometimes has significantly less influence on a good solution, resulting in the contour plot suggesting good solutions for any α value paired with $\beta \geq 8$.

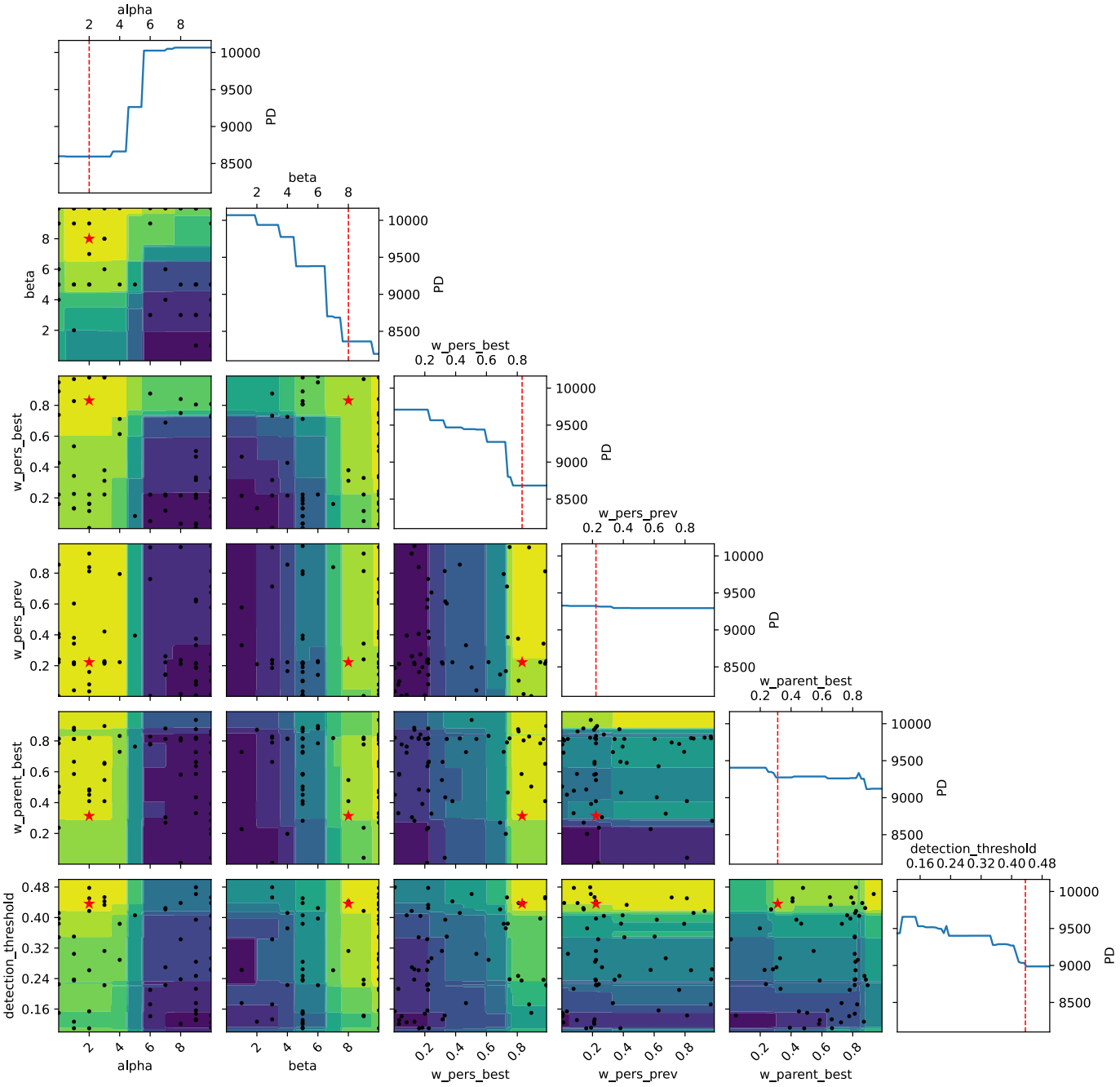


Figure 6.13: Partial dependence plot using the surrogate model of the optimizer run with the best parameter set for problem instance `berlin52` and dynamic intensity $C = 0.25$. Better (smaller) solutions are lightly colored, dark areas represent worse solution areas.

Continuing on the diagonal, high values for w_{persbest} seem to have a large impact on the solution quality for the predictive model, with $w_{\text{persbest}} > 0.8$ as the optimum. Interestingly, this result differs greatly from previous observations from the statistical box plots

grouped by problem instance, where `berlin52` had a median of 0.35. The contour plots also show clear preferences for high values of w_{persbest} in combination with high values of β and especially with low values of α.

On the other hand, the model predicts the same PD regardless of any value of w_{persprev}. This is also evident in the contour plots of the same row, where the light-colored good solution areas are only affected by well-performing α and β values, respectively. This is also true for the combination of w_{persbest}, where the avoided area shown in the scatter matrices is also absent.

The parameter $w_{\text{parentbest}}$ has a small influence on the model prediction, with values > 0.85 being most beneficial for a low PD. However, this influence does not appear to be sufficient when looking at the contour plots for the first three parameters from the left, where a similarly distinct edge can be seen that is mainly influenced by these more influential parameters. In combination with w_{persprev}, almost no area seems to be of real interest except for a narrow region in the upper right, suggesting that the previously mentioned high values for $w_{\text{parentbest}}$ should be paired with also high values for w_{persprev}.

The detection threshold θ showed its lowest value for PD at $\theta > 0.44$. This value range is also seen in the contour plots, where the light-colored areas are always above 0.4, except in combination with β values close to 10, where lower values for θ are also possible, suggesting that the higher heuristic influence could compensate for scarcer dynamic detection and thus triggered dynamic reaction mechanisms. Furthermore, by looking at the red stars, we can see that the HPO process almost always chose a parameter in the areas favored by the surrogate model, except for the pairing of w_{persprev} and $w_{\text{parentbest}}$, where the predicted best, yellow area in the upper right corner, was never even sampled by the aggregation function.

The second partial dependence plot, Figure 6.14, shows the results for a higher dynamic intensity of $C = 0.5$. The same results hold for α and β, but with the aforementioned lesser influence of smaller α values on PD in the one-dimensional plot. The diagonal plot for β also shows a less significant decrease in PD between 4 and 8 than before, instead resembling more of an exponential function. The weight w_{persbest} is also much less influential for the surrogate model. In fact, the diagonal plots for all three weights show that no particular value in the entire range between 0 and 1 has a significant impact on the prediction of the surrogate model, at least compared to β. This is also true for the detection threshold θ. The corresponding contour plots for the weights and θ show similar edges, separating light-colored good solution areas from darker areas, as with $C = 0.25$, but with less intense dark-colored spots, suggesting that more areas could result in satisfactory performance. The remaining 13 partial dependence plots show very similar results for the weights and θ.

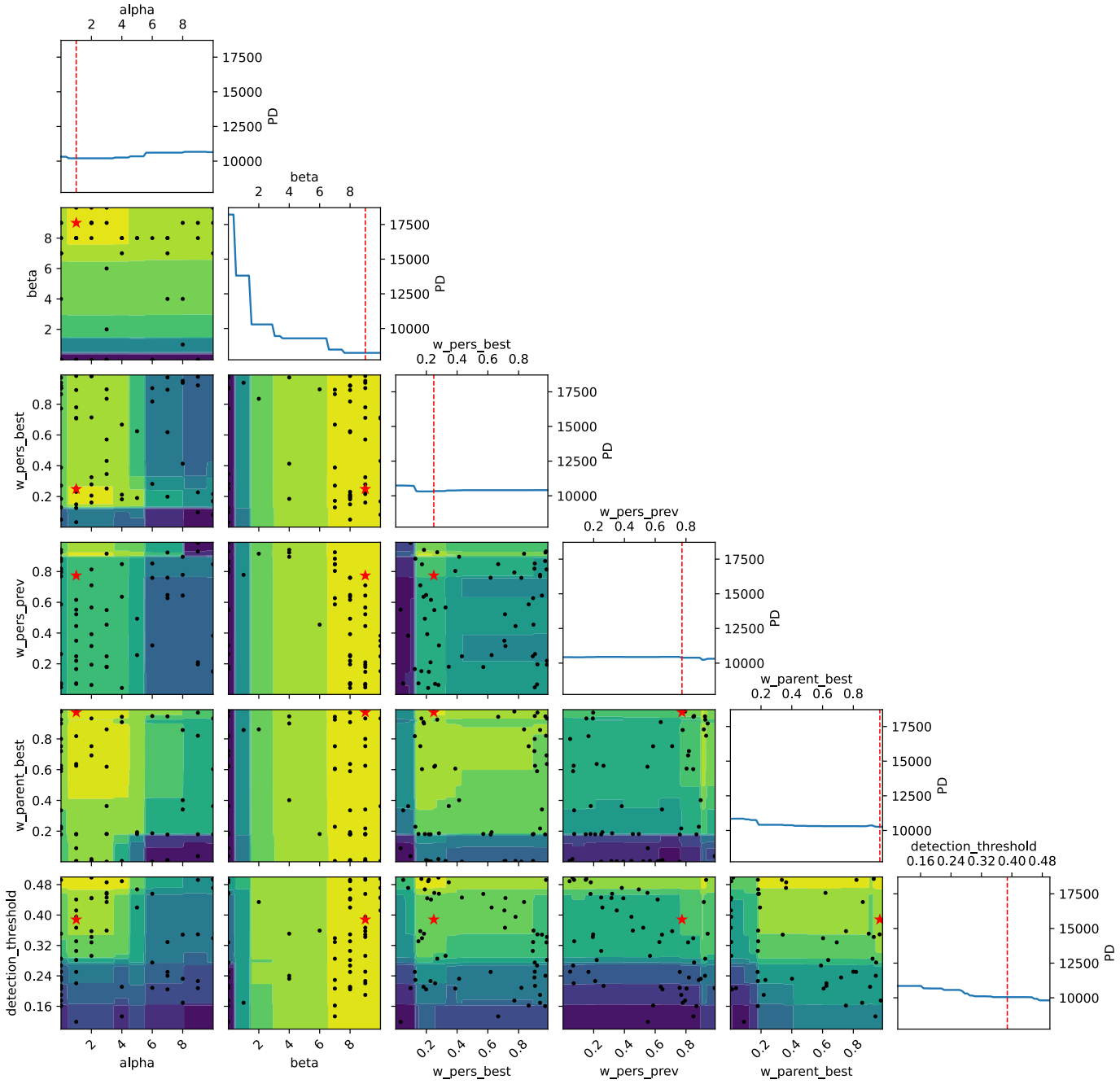


Figure 6.14: Partial dependence plot using the surrogate model of the optimizer run with the best parameter set for problem instance `berlin52` and dynamic intensity $C = 0.5$. Better (smaller) solutions are lightly colored, dark areas represent worse solution areas.

In addition, compared to the previous, less dynamic representation of the partial dependence, the best observed parameters, indicated by the red stars, are less often in the areas indicated as optimal by the model, suggesting a mismatch between HPO and

the produced model. In this case, potentially more objective calls or a reduction of the parameter dimension could improve this situation. However, it is quite possible that this mismatch is only produced by the random sampling of the model prediction and is not significantly present in repeated predictions with a different random initialization.

Concluding, we can confirm some relationships between parameters, especially that of α and β. However, even by using the trained model for its predictive capabilities, we did not gain any further insight into possible direct correlations between parameters. Nevertheless, the importance of certain parameters for the surrogate model is an interesting addition to this analysis.

6.2.3 Parameter Importance

In the previous subsection, regarding the partial dependence plots, we already discussed the influence of the parameters on the surrogate model and its prediction of solution quality. This was achieved by explicitly sampling the parameter search space using the surrogate model. For some simpler ML models, mostly decision tree-based, the feature, or in this case the parameter importance, can be extracted directly from the structure of the model. The previously required intermediate sampling is now replaced by analyzing the structure of the regression trees of the GBRT surrogate model used for the experiments. The resulting relative parameter importance, or so-called Mean Decrease in Impurity (MDI), as explained in Section 5.5.2, expresses how important a particular parameter is for the prediction of the underlying model. For example, for a simple linear regression, $y = a_0 \cdot x_1 + a_1 \cdot x_1 + ... + a_n \cdot x_n$, the values for a_i would indicate the relative importance for their corresponding parameter x_i.

As mentioned earlier in the experimental setup chapter, the procedure for calculating the MDI has some limitations. It has a bias towards parameters with high cardinality and therefore tends to underestimate the importance of parameters such as H, which has only two values. Furthermore, it can only reflect the importance resulting from the 60 objective calls during the optimizer run, i.e., it is not a statement about the importance of the entire parameter search space and, as such, of the H-SPPBO. Nevertheless, the MDI is still an indicator of what the model considered important enough to base its predictions on, which then guides the successful search for well-performing parameter sets.

Table 6.7 shows the averaged MDI for all H-SPPBO parameters with their standard errors over all 90 optimizer runs. The table is also ordered from the most highest value (left) to the lowest value (right), and all values sum to 1. The importance of the parameter β is by far the highest at 0.303, indicating that about a third of each model-prediction is explained by this value. The standard error is also the highest by comparison, but still very low and

Table 6.7: The relative parameter importance (MDI) computed via the surrogate model over all 90 optimizer runs with the standard error.

β	α	$w_{\text{parentbest}}$	θ
0.303 ± 0.007	0.185 ± 0.006	0.132 ± 0.005	0.132 ± 0.005

w_{persprev}	w_{persbest}	H
0.115 ± 0.004	0.108 ± 0.004	0.025 ± 0.002

far away from the margin of error for the second most important parameter, α, at 0.185. This difference of more than 10% between the first and second most important parameters suggests that although this MDI metric is biased, β is definitely an important parameter for the H-SPPBO metaheuristic. Including the importance of α may also explain why only these two parameters were the most robust and consequently allowed for a reasonably meaningful value suggestion.

The next most important parameters are $w_{\text{parentbest}}$ and θ, both with $MDI = 0.132$, also with very low standard errors when averaged over all optimizer runs. In particular, $w_{\text{parentbest}}$ showed a slightly more robust behavior in the box plots and also had a higher median than the other two weights, thus implying more importance to the solution creation process. Therefore, it seems reasonable that $w_{\text{parentbest}}$ has a higher MDI than the other weights.

This is followed by w_{persprev} at $MDI = 0.115$ and w_{persbest} at $MDI = 0.108$, both of which are very close to the previous two parameters, with at most 3% less importance. Their standard errors are also very small, at around 3.5% relative to their values. The least important parameter, as already indicated by the bias, is the dynamic reaction type H. With only $MDI = 0.025$, it is four times less important than the next most important parameter w_{persbest} and 12 times less important than the most relevant parameter β.

The order of parameter importance does not change when the optimizer runs are grouped by their respective dynamic intensities C, as shown by the bar plot in Figure 6.15. Here, the values are also discussed qualitatively for comparison, rather than giving the absolute MDI values as before. Most of the differences in MDI are within the margin of error, represented by the black whiskers at the right end of each bar. Only an increase in the importance of α and a slight decrease of w_{persprev} and $w_{\text{parentbest}}$ are noticeable for medium dynamic intensities ($C = 0.25$) compared to the other dynamics. Also, the detection threshold becomes slightly more important as the dynamic intensity increases, which could be expected from an increasingly changing problem environment, where

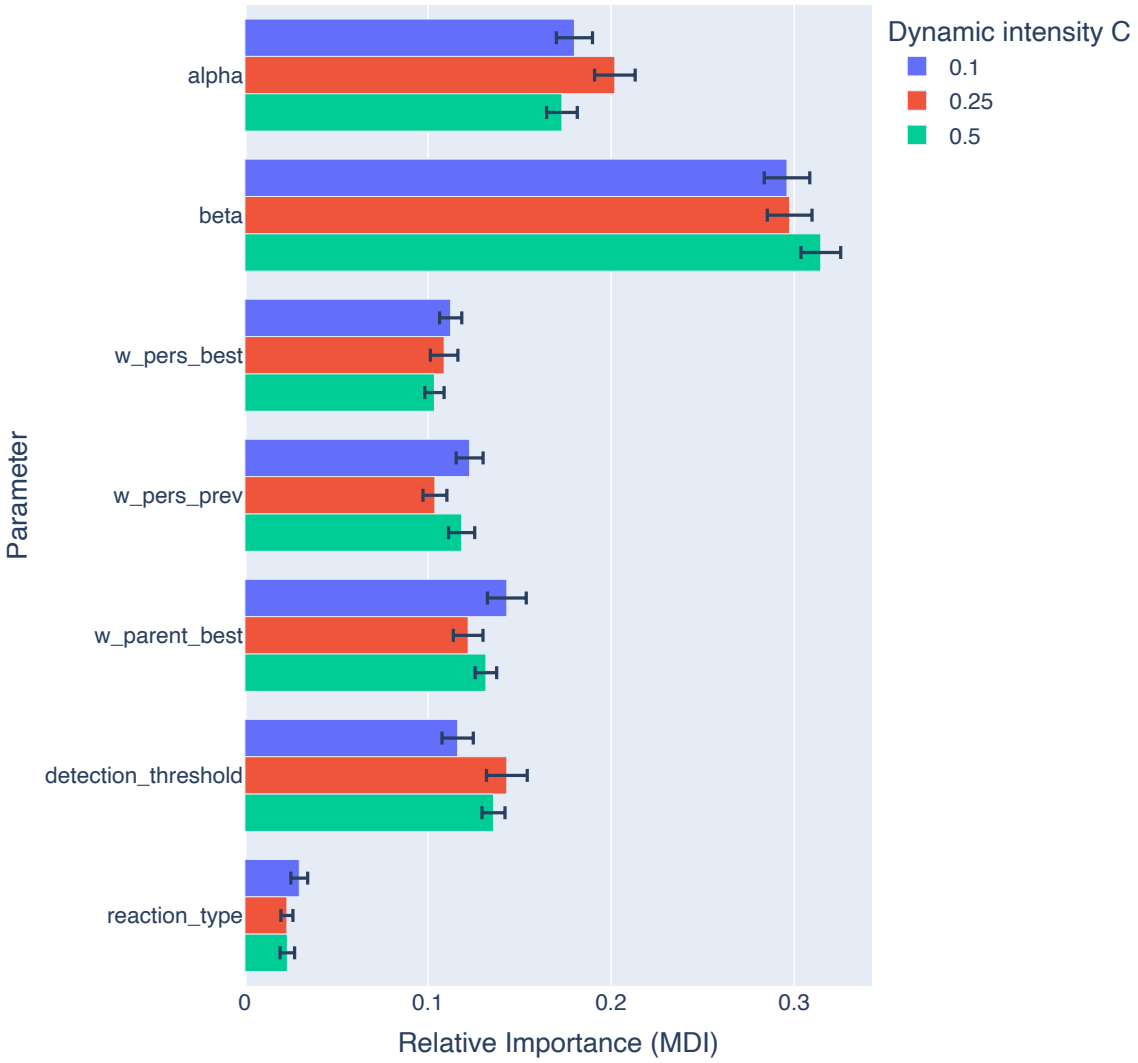


Figure 6.15: Bar plot of the parameter importance (MDI) computed via the surrogate model over all 90 optimizer runs and compared by dynamic intensity C.

improved change detection leads to better solutions. However, all these results are very close to the error margins of their groups and, as already mentioned, do not significantly change their relative importance compared to the values discussed in the previous table.

Similar findings can be made when grouping across problem instances, where the order parameter of parameter importance remains mostly the same. Figure 6.16 shows only a few problem instances that change the importance for certain parameters. In particular, the highly clustered instance `pr226` has an importance of β of over 0.35, more than any other instance, with a relatively low standard error. This instance also has α at about the same importance as the overall average, but the weights and detection threshold θ at even lower importance. In contrast, the very small and semi-regular instance `eil51` has the

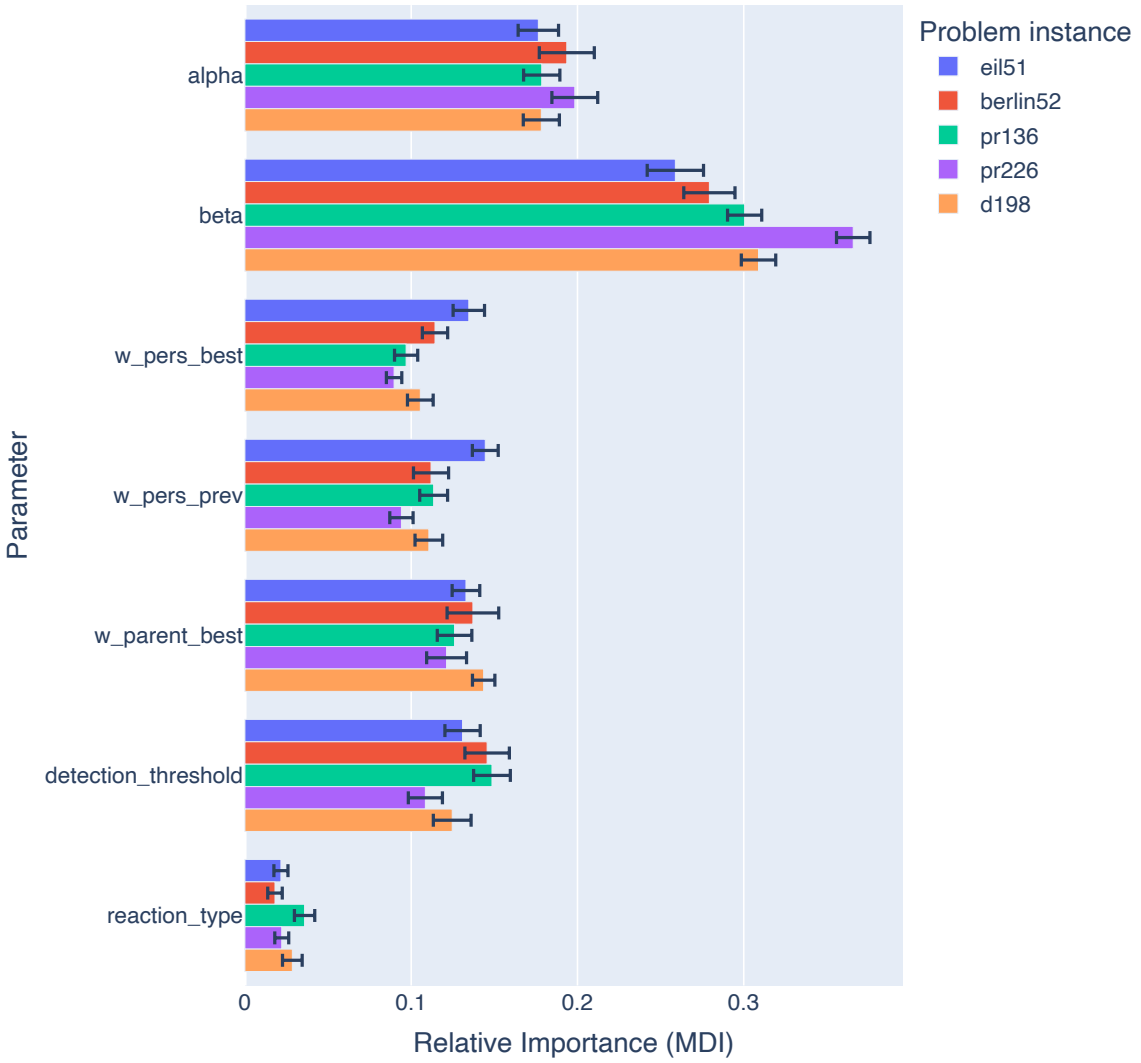


Figure 6.16: Bar plot of the parameter importance (MDI) computed via the surrogate model over all 90 optimizer runs and compared by problem instance.

lowest importance out of all for β and the highest MDIs for the parameters w_{persbest} and w_{persprev}, therefore suggesting an increasing relevance of the personal SCE populations for the solution creation process. In addition, instance `pr136` evaluated the reaction type H as more important than any other problem instance.

In summary, β is undoubtedly the most important parameter for the surrogate model in every possible problem scenario, followed by α with about 10% less relative importance. The three weights and the detection threshold θ are evaluated with varying importance depending on a certain dynamic intensity and problem instance, so that the importance

gap to α is even smaller under some conditions. Finally, the reaction type appears to be the least important parameter, although this statement may be significantly invalidated by the cardinality bias of the MDI method.

6.3 Part III - Evaluating the Parameter Sets

The third part of the evaluation is based on the experimentation data of all ten TSP instances, small and large, with all three dynamic intensities $C = \{0.1, 0.25, 0.5\}$, and each combination of instance and intensity was repeated $r_{\text{exp}} = 20$ using their specific optimal parameter set acquired by HPO. As explained, all the parameters that have been acquired from smaller instances were used for the larger instance in the same structural group (see Table 6.5). In addition to the HPO parameter sets, a generally well-performing reference set based on the results of the original H-SPPBO paper [10] was used. In visualizations and comparisons, this is denoted as "Ref". In contrast to the HPO experiments, the parameter set for these reference experiments was the same for all combinations of instance and dynamics.

Furthermore, all graph and table data were computed by averaging the 20 repeated experimentation runs for each instance and dynamic intensity, and for aggregated or grouped data, they were also averaged across different dynamic intensities and/or TSP instances. However, this is be mentioned where relevant. Since only *TSPLIB* instances were used, the *RPD*, i.e. the relative difference to the optimal solution, was used to compare the solution quality, as in the first part of the results chapter. Also, as mentioned in Section 5.4, dynamic events start at iteration 2000 and occur at every next 100th iteration up until the last iteration at 2599. The time between each dynamic events is referred to as dynamic interval.

6.3.1 Solution Quality

After the first two result parts answered the two main research questions, we proceed to the fundamental, underlying question of whether or not these HPO parameter sets produce satisfactory results at all. Therefore, the solution quality in the form of the previously used *RPD* is shown over the course of the dynamic runtime, starting at iteration 1950, just before the first dynamic change is triggered at iteration 2000, and continuing until iteration 2399, omitting the last two dynamic events since they do not add any relevant data.

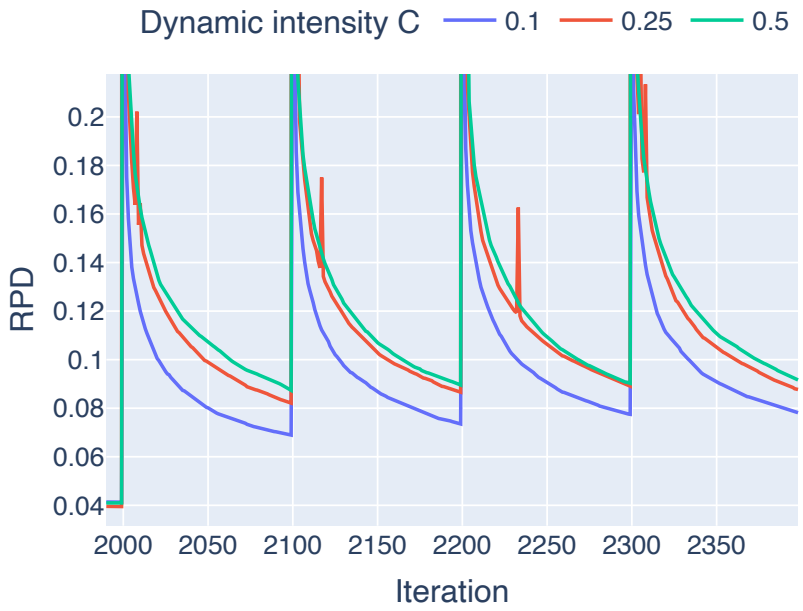


Figure 6.17: Line plots showing the relative solution quality *RPD* over the first 400 dynamic iterations (starting just before iteration 2000) for all three dynamic intensities C averaged over all instances that used the HPO parameter sets.

Figure 6.17 shows one of these run plots with different lines for each dynamic intensity C and averaged over all TSP instances. The first impressive aspect to note is that the solution quality just before the first dynamic event at iteration 2000 is exceptionally good for all dynamic intensities around $RPD = 4\%$. Although the solution quality of the first 2000 iterations, representing the static TSP problem, was only weighted with 1/7, compared to the other six solution qualities taken just before the next dynamic event, the influence must have been sufficient for the HPO process to produce good results here as well.

Continuing to the first dynamic event at iteration 2000, there is an expected huge spike in solution quality for all three intensities, which continues along the y-axis up to $RPD = 3$ and was truncated to increase the differentiation of the lines between the dynamic events. Interestingly, the medium dynamic intensity $C = 0.25$ reached a higher value of $RPD = 3$, while the solution quality for the higher dynamic $C = 0.5$ did not spike as high to only $RPD = 0.7$. This is mainly because the majority of parameter sets at $C = 0.25$ used the full reset (H_{full}) as the change handling procedure, which resulted in significantly worse solution quality immediately after the change was detected, rather than the partial reset, where better performing solutions were still stored in the SCE tree. At iteration 2008 there is another spike, indicating a delayed execution of the change handling procedure,

which can also be observed at different iterations in between each following dynamic interval. These spikes are also due to some specific instances that will be further analyzed at a later time.

After these first 10 iterations following the dynamic event we observe a rapid increase in solution quality up to iteration 2099, with the parameter sets produced for the least dynamic problems $C = 0.1$ clearly leading in *RPD* value, achieving a solution only about 3% worse than the static one. The *RPD* values just before the next three dynamic events at 2200, 2300, and 2400 also show leading performance, although not as good as in this first interval. The solution qualities for the other two dynamic intensities also managed to improve to $RPD_{C=0.25} = 0.8$ and $RPD_{C=0.5} = 0.9$ just before the next dynamic event, which is also the case for the next three dynamic intervals.

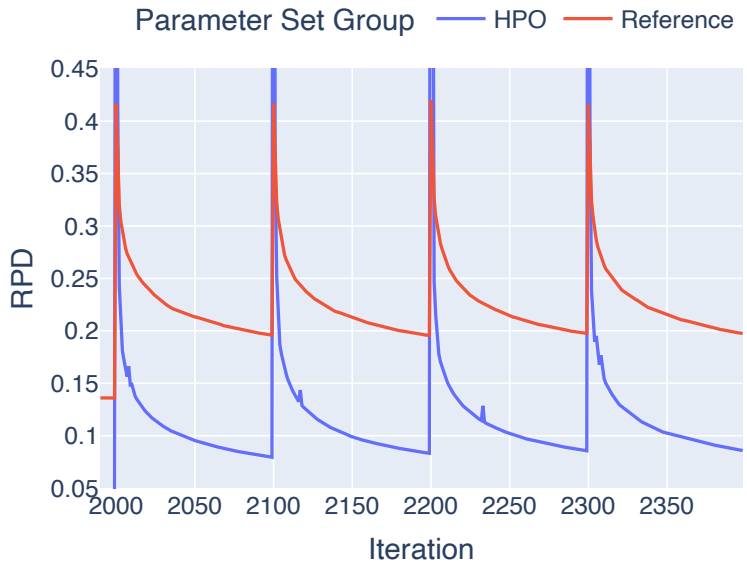


Figure 6.18: Line plots showing the relative solution quality *RPD* over the first 400 dynamic iterations (starting just before iteration 2000) for both parameter set groups, HPO and the reference set, averaged over all three dynamic intensities C all ten TSP instances.

To put these solution quality values into perspective, Figure 6.18 shows the results for the reference parameter set in the same graph as the HPO results, averaged over all dynamic intensities and TSP instances. Although the reference set also produces acceptable results, the parameters obtained by HPO clearly outperform them at every point in the dynamic runtime. Not only is the static solution more than three times better up to iteration 2000 than the reference parameter set ($RPD_{\text{HPO}} = 0.4$ vs. $RPD_{\text{Ref}} = 0.14$)), the solutions obtained during each dynamic interval, just before the next dynamic event, are also significantly better with more than two times lower *RPD* values of 0.08 compared to

$RPD_{\text{Ref}} = 0.2$. However, the reference parameter set appears to have no noticeable false detections of dynamic change in its dynamic intervals and also spikes to much lower *RPD* values of just over 0.4.

When viewed separately for each dynamic intensity, as shown in Figure 6.19, the impression does not change much. Just before each dynamic event, the reference parameter set consistently produced almost the same solution quality of $RPD_{\text{Ref}} = 0.2$, while the HPO parameter sets produced better results for lower dynamic intensities of $C = 0.1$, as discussed above.

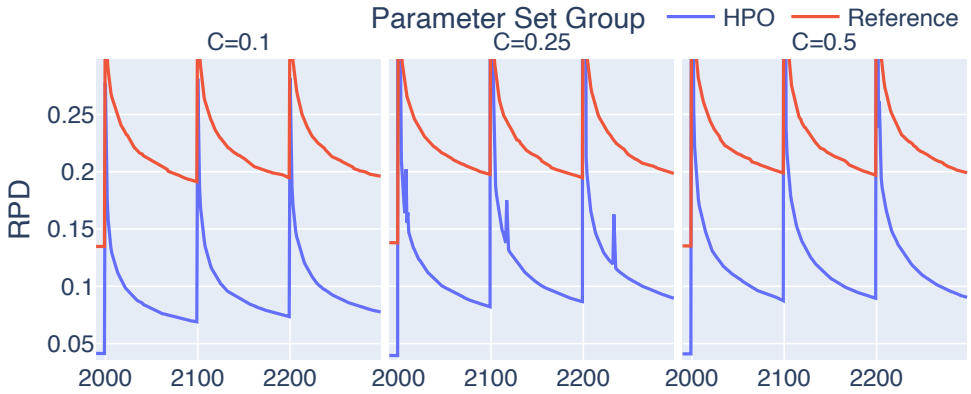


Figure 6.19: Line plots showing the relative solution quality *RPD* over the first 300 dynamic iterations (starting just before iteration 2000) for both parameter set groups, HPO and the reference set, with a plot for each dynamic intensity$C$, averaged over all ten TSP instances.

Problem Instance Generalization

Next, we want to evaluate the problem classification and the proposed generalization of parameter sets obtained from smaller problem instances to larger instances of the same structural group. For this purpose, Figure 6.20 shows the run plots for each individual TSP instance and each dynamic intensity C. The instance names are at the top of each plot and they are ordered in the same way as before in Figure 5.2, so that the smaller instances are in the left column and the larger instances are in the right column, with each row corresponding to the predefined structural groups. Thus, each row of plots uses the same set of parameters when comparing dynamics.

The first row shows instances with random to nearly regular distribution. Although `rat195` has a dimension almost four times larger than `eil51`, the solution qualities for the static problem and the following dynamic intervals are very similar. With only a few percent worse *RPD* values for `rat195` the overall performance just before the end of

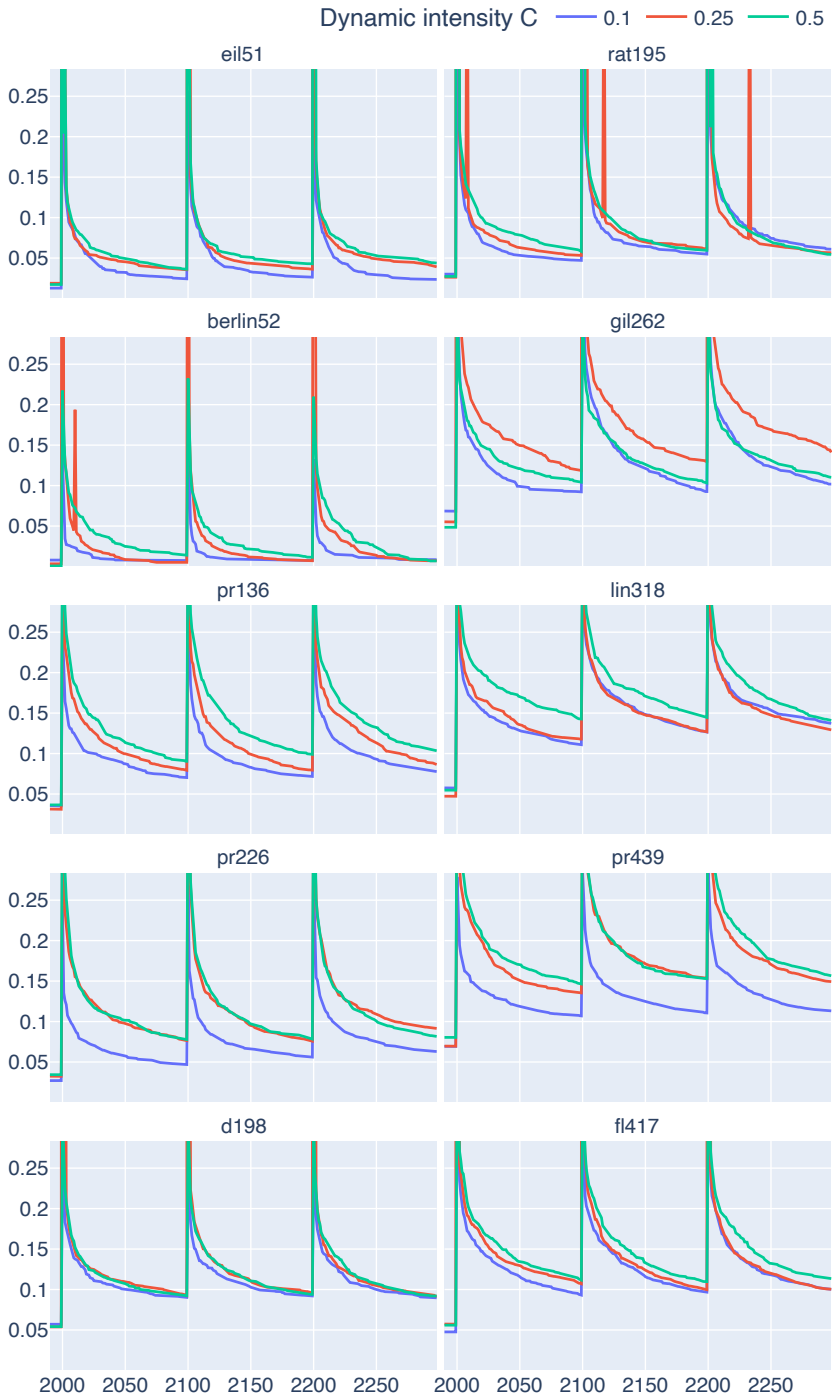


Figure 6.20: Line plots showing the relative solution quality *RPD* over the first 300 dynamic iterations (starting just before iteration 2000) for all three dynamic intensities C and all ten TSP instances for the HPO experiments. The smaller instances are in the left column and the larger instances are in the right column. Each row corresponds to the predefined structural groups (see Figure 5.2).

each dynamic interval is very satisfactory at around $RPD = 0.05 \pm 0.01$. Furthermore, the static performance for `eil51` shows that it was almost perfectly solved for all three dynamic intensities at $RPD_{C=0.1} = 0.01$ and $RPD_{C=0.25,0.5} = 0.02$, which is even more interesting considering the use of very different parameters for each dynamic intensity. The larger pendant `rat195` also got very good static performance at around $RPD = 0.025$, but has some significant cases of false dynamic detection, even resulting in spikes five iterations ($L_{\text{pause}} = 5$) after the dynamic event. Although both instances used the same parameter sets for $C = 0.25$, the higher problem complexity resulted in an increase in SCE tree swaps many iterations after the dynamic event occurred, which then resulted in a large increases in RPD because H_{full} was used as the change handling procedure. The differences between the dynamic intensities are very small for both instances, which could be explained by the regular distribution of city nodes and therefore easier solution adaptation process after the problem change.

The second row contains TSP instances with smaller, slightly clustered areas with an otherwise random structure. However, the two instances produced very different solution qualities. The H-SPPBO algorithm almost always found the optimal solution in the static part for the smaller instance, `berlin52`, while the RPD for `gil262` is only about 5% just before the dynamic event at iteration 2000. Also, the parameter set used for the low dynamic intensity of $C = 0.1$ produced a worse static solution quality than with the other two higher dynamic intensities. Looking at the parameter set in Table 6.5, this could be explained by the use of the highest value for α across all parameter sets of $\alpha = 3$, paired with the lowest $w_{\text{parentbest}}$ value of $w_{\text{parentbest}} = 0.196$. This could be due to the fact that `berlin52` has only one fifth of the city nodes of `gil262`, making it easier to find a good static solution, and the positive influence of these parameters on the dynamic performance. Another interesting observation for `gil262` is the worse dynamic performance of the parameter set for $C = 0.25$ compared to $C = 0.5$, although the results for `berlin52` do not show this behavior. Again, by looking at the parameters used, we can see that for $C = 0.25$ a full reset of the SCE tree (H_{full}) was used with a very high dynamic detection threshold of $\theta = 0.436$, thus making the algorithm less sensitive to dynamic changes in the problem instance. The instance `eil51` also used a similarly high $\theta = 0.46$ with H_{full}, but for a the high dynamic intensity of $C = 0.5$, which produced better results because the dynamic detection threshold was likely reached more often.

The third row shows more consistent results when comparing the smaller instance, `pr136`, and the larger instance, `lin318`. These problems are artificially structured with certain patterns of medium clustered regions, with small distinct holes in the city distribution. The corresponding parameter sets use $\alpha = 2$, low w_{persbest}, and H_{partial} for all three dynamic intensities. This resulted in good static performance just before the first dynamic event in iteration 1999, with RPD values of just under 5% for `pr136` and around 5% for `lin318`. The following first dynamic event shows very low spikes in RPD with the highest value

being 4% for $C = 0.5$. The increase in solution quality (and the decrease of the curve) seems to be less significant than for `berlin52` or the first row, and the *RPD* values just before the next dynamic event do not come close to the static performance, especially for $C = 0.5$ where the metaheuristic only obtained a solution of $RPD = 0.1$ for `pr136` and $RPD = 0.15$ for `lin318`, which is one of the worst results for any problem instance. The parameter sets for the two lower dynamic intensities can only marginally improve the *RPD* values between dynamic intervals. The artificial pattern of city placement appears to make it more difficult to find good solutions after a dynamic change in a short amount of time.

The fourth row shows TSP instances with few highly clustered areas, `pr226` and `pr439`, which produce a run curve very similar to the previous row. Interestingly, both structural groups share the same value range for the regularity index, which may explain their similar solution quality convergence behavior. They also show similarities in static performance, with `pr439` having a slightly worse solution quality, and the larger instance `pr439`, again showing one of the worst dynamic interval performances, only reaching $RPD_{C=0.25,0.5} = 0.15$. However, the solution quality for the lowest dynamic $C = 0.1$ is much lower for both small and large instances, with differences of up to 5% in *RPD*, but only in dynamic situations. This suggests that the parameters seem to be highly tuned to perform well for small changes in the problem. Notable values in this set are very low $w_{\text{persbest}} = 0.112$ and $w_{\text{persprev}} = 0.051$, comparatively higher $w_{\text{parentbest}} = 0.436$, and a low detection threshold of $\theta = 0.183$. Especially the last parameter could have a big impact on the good dynamic performance. However, other instances, such as `d198`, with a similar θ value at $C = 0.1$ do not show the same good performance. This suggests that the structural characteristics of both problem instances may have had an influence.

The fifth and last row, with instances `d198` and `fl417`, represent a dispersed city placement, with highly clustered areas with few or no city nodes in between. The regularity index here is different from the previous two rows (groups). The convergence behavior between dynamic intervals is now more similar to the first row, with a rapid increase in solution quality in the first few iterations after the dynamic event, followed by a stagnation period up until the next dynamic event. Also, the difference between smaller and larger instance is the smallest out of all structural groups, with a gap of only 1-2% for $C = 0.5$. Interestingly, the static performance of the larger instance, `fl417`, is better than that of the smaller instance, `d198`, for $C = 0.1$, which is also true for the first dynamic interval.

Compared to the same plots for the reference parameter set, shown in the Appendix A, Figure A.5, we can immediately confirm the conclusion from the aggregated plot above. The HPO clearly outperforms the reference parameter set, with some instances showing significantly worse solution quality between dynamic events with *RPD* values over 25%.

Also, the difference between dynamic intensities is not nearly as significant as for the HPO parameters, suggesting that the tuning process specific to each dynamic intensity also seems to improve performance.

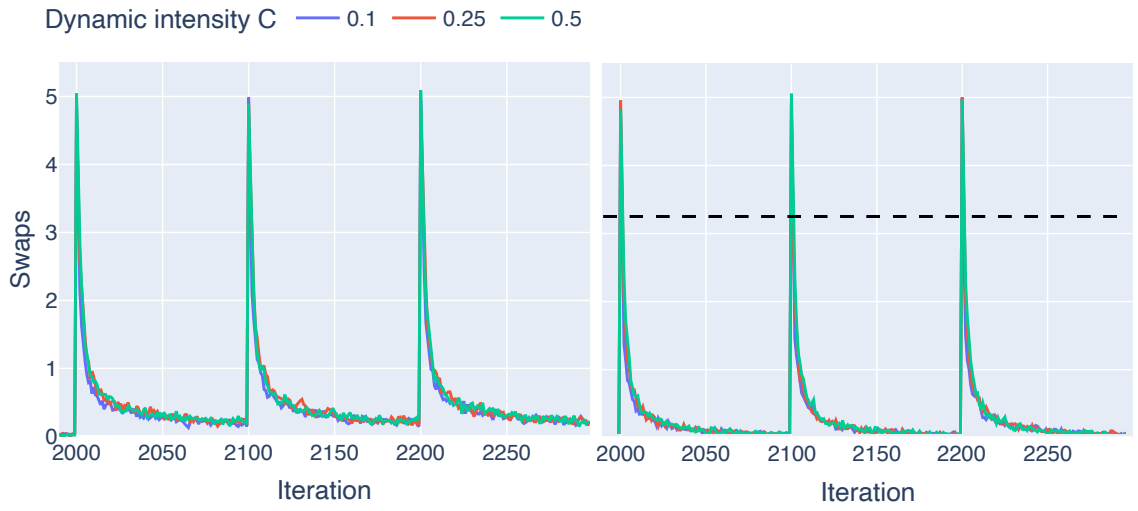
In summary, all ten TSP instances show their own specific solution quality behavior and response to dynamic changes. Although a similar convergence behavior can be found within each structural group, which is also true for the plots for the reference parameter set, the data is not sufficient to suggest a clear relationship to the city placement characteristic. However, the generalization assumption can be confirmed to some extent, since all parameter sets for smaller instances also produced very good results for their larger counterparts. For a more profound evaluation of this aspect, a cross-validation of these parameter sets over different TSP instances and dynamic intensities would have been necessary, but would have increased the number of experiments by a factor of 8.5.

6.3.2 Detection of Dynamic Changes

A very important aspect of the H-SPPBO metaheuristic is its ability to detect changes in the problem it solves and, in response, trigger a change handling procedure to respond optimally. Its performance in this regard depends solely on the dynamic detection threshold θ and its response mechanism on the H mechanism, H_{full} or H_{partial}. Without any prior knowledge of the way the HPO method chooses these parameters, one would assume that, since the main goal is to obtain good solutions in dynamic environments, the correct detection of these changes is of high importance to the algorithm and thus to the HPO method. However, we have already discussed the minor importance of these two dynamic-relevant parameters in the previous results section (see Section 6.2.3).

Since the results in this respect deviate very strongly from this assumption, the general conclusion should already be anticipated. With a few exceptions, related to specific combinations of instances and dynamics, and thus individual parameter sets, the correct detection of dynamic changes in the problem does not seem to have a particularly large effect on the solution quality of the H-SPPBO algorithm when using HPO parameter sets. More surprisingly, the reference parameter set outperforms the HPO parameter sets in every discipline in terms of dynamic detection. This puts into perspective the exceptionally good solution quality performance of the HPO parameter sets discussed before and raises the question of how relevant the correct detection of dynamic changes really is for good solutions to the DTSP. To support these statements, the following graphs and tables provide performance metrics for many aspects of the dynamic detection behavior.

First, the number of swaps, i.e., the changes within the SCE tree that eventually exceed the dynamic detection threshold θ, is plotted over the first three dynamic intervals, similar to the solution quality plots from above. Figure 6.21 shows the results averaged over all ten TSP instances, but separated by dynamic intensity C, represented by a line for each value. The left line plot shows the HPO parameter sets, the right plot shows the reference parameter set. An additional dashed line was drawn for the reference parameter set, indicating the detection threshold $\theta = 0.25$, which results in $0.25 \cdot |A| = 3.25$ swaps required to trigger the change handling procedure. A similar line cannot be drawn for the HPO plot because every TSP instance uses its own θ value. A later plot, separated by problem instance, will show more details.



(a) HPO **(b)** Reference

Figure 6.21: Line plots showing the relative solution quality swaps within the SCE tree over the first 300 dynamic iterations (starting just before iteration 2000) with a plot for each parameter set group, HPO and the reference set, each showing all three dynamic intensities C, and averaged over all ten TSP instances.

The first thing that is immediately noticeable is the similar number of swaps for both parameter groups, with an average of 5 swaps right at the dynamic event for all three dynamic intensities. This makes sense when considering the three levels of the SCE tree, where the first two levels are very likely to lose their high position when using the partial reset strategy H_{partial}, resulting in 4 swaps. However, the H-SPPBO algorithm using the reference parameter set has its threshold at a reasonable level, which allows the dynamic response mechanism H to be almost always correctly triggered immediately after the dynamic event. This partial reset allows the algorithm to find new, good solutions, which greatly reduces the number of swaps around 25 iterations after the dynamic event. Eventually, after the 50th iteration in each dynamic interval, the amount of swaps has been

reduced to about 0.10 ± 0.05 for all the dynamic intensities. Compared to the H-SPPBO results, the amount of swaps after this 50th iteration after each dynamic event is much higher at 0.30 ± 0.1, indicating that even long after the dynamic event has occurred, the algorithm is still finding better solutions than its direct parent node at about every third iteration. This suggests that either, the change handling procedure was not triggered, allowing old, possibly bad, solutions to influence the search for a longer time, or that the particular combination of parameters just performed enough exploratory behavior to still find increasingly better solutions at a later time. While the latter theory can be dismissed by looking at the almost identical convergence behavior of the solution quality comparison in Figure 6.18, the former theory is discussed later.

Continuing with Figure 6.22, which shows the swaps over the first three dynamic intervals, this time also separated by problem instance, and only for the HPO parameter sets, we can also draw dashed line representing the dynamic threshold that triggers H. As before, the instances are arranged according to their structural group and dimension. Looking at the first three rows, the thresholds for at least two of the dynamic intensities are very high at around 5 swaps, or even 6 in the case of `eil51` and `rat195`, using $\theta = 0.460$ for the highest dynamic intensity $C = 0.5$. This threshold is almost never reached, and even the two thresholds for the lower dynamic intensities are only just passed in some cases. Another interesting aspect in the third and fifth row is that here the HPO process chose the lowest detection threshold for the highest dynamic intensity $C = 0.5$ at $\theta_{\text{pr136}} = 0.139$ and $\theta_{\text{d198}} = 0.216$, which is consistent with the results from the previous discussion in Section 6.2.1. Furthermore, the amount of swaps after the 50th iteration in each dynamic interval remains high as seen before, with the smallest instances, `eil51` and `berlin52`, reaching a lower value level than other instances, such as `lin318` and `rat195`, which show a much higher average amount of swaps in the later stages of the dynamic interval. Thus, even though the same parameter sets were used in each row for each dynamic intensity, either the higher dimension or influences of the city placement characteristic had an impact on the number of swaps within each structural group.

In addition to the number of swaps, two performance metrics from machine learning classification, also used in information retrieval, were calculated on the data. As explained in Section 5.5.3, these metrics are precision and recall. The precision describes how many detected iterations (TP + FP) were actually TP, and thus describing how relevant the detected iterations are. The recall describes how many possible correctly detectable iterations (positives) were actually detected by the H-SPPBO algorithm (TP). Both metrics span a range between 0 and 1, where 0 means that no correct detections (TP) were found at all, and 1 means that either every dynamic event correctly triggered the change handling procedure, but allowing for FPs (recall), or that all change handling procedures triggered were in response to a dynamic event, but allowing for dynamic events without

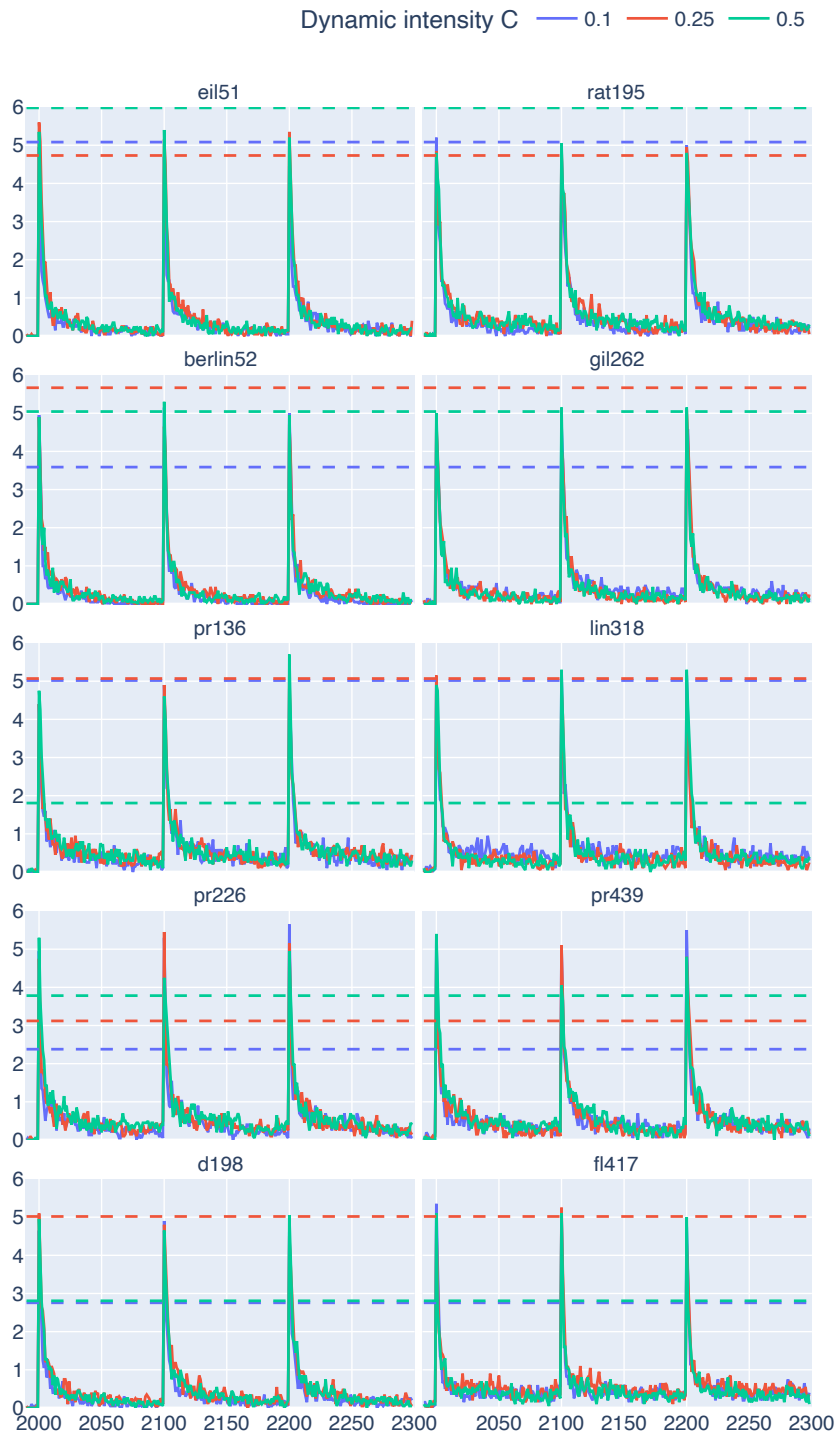


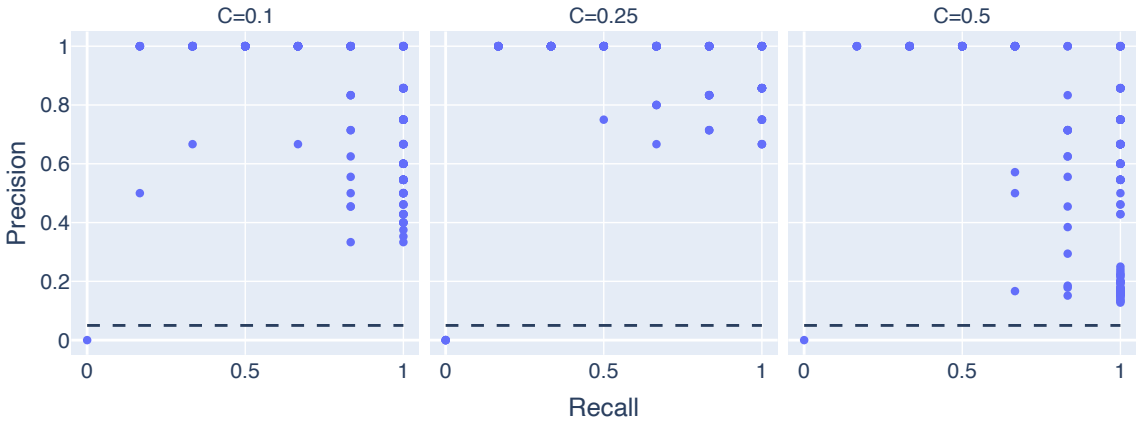
Figure 6.22: Line plots showing the relative solution quality swaps within the SCE tree over the first 300 dynamic iterations (starting just before iteration 2000) for the HPO parameter set showing all three dynamic intensities C with a plot for each TSP instance. The smaller instances are in the left column and the larger instances are in the right column. Each row corresponds to the predefined structural groups (see Figure 5.2).

any response (precision). The direct results of these metrics can also be found in Table B.4 for the HPO parameter set and Table B.5 for the reference parameter set, both inducing each combination of TSP problem instance and dynamic intensity C.

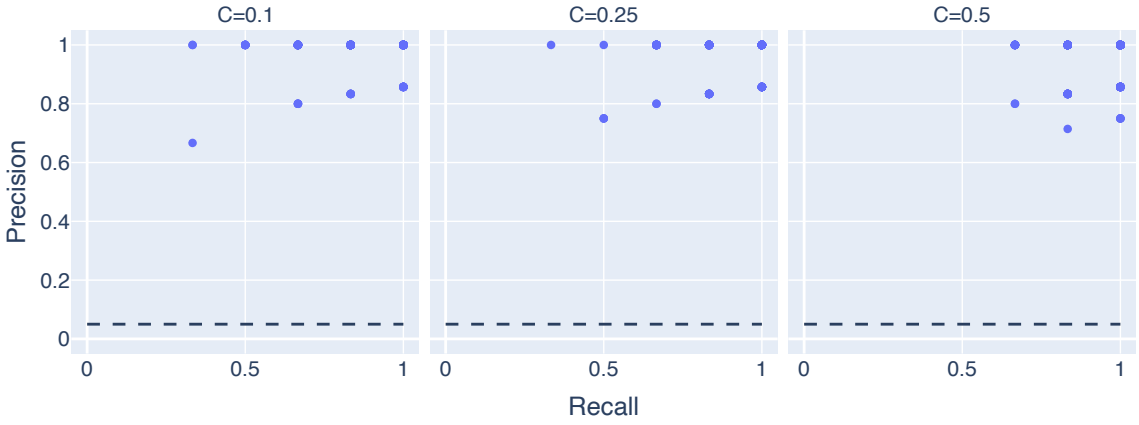
In ML visualization, these metrics are often presented in precision-recall curves for certain thresholds of classification confidence. A very good classifier will have a PR curve close to $precision = 1$ for varying recall values. A common baseline in these plots is a line parallel to the recall axis for the precision value representing the fraction of all positives to all negatives, with a good classifier staying above this line. In an example with equal amounts of positives and negatives, this line is drawn at $precision = 0.5$, thus a random classifier could achieve this precision on average. When applied to this problem, the dynamic detection threshold of H-SPPBO is not comparable to a classifier threshold, making it impractical to draw a curve or area plot from these precision and recall values. Instead, scatter plots were drawn with the baseline at $precision = 30/570 \approx 0.053$.

Figure 6.23 shows a PR scatter plot for each dynamic intensity C, averaged over all TSP instances, and compared for both the HPO parameter set (6.23a) and the reference parameter set (6.23b). Again, the better dynamic detection performance of the reference set is clearly evident, with PR points staying well above the precision baseline and mostly populating the upper right corner of the high precision and recall values, meaning that almost all dynamic events were found while avoiding false positives. Interestingly, the performance on the least dynamic instances $C = 0.1$ shows the worst PR values at $recall = 0.33$ and $precision = 0.67$. This suggests that fewer dynamic events were detected in this case, which is to be expected since the static threshold of $\theta = 0.25$ was reached less often. On the contrary, the highest dynamic intensity of $C = 0.5$ showed the best PR values.

Looking at the results for the HPO parameter sets, the PR values are more scattered, while generally performing worse, and sometimes coming close to the precision baseline. There even seemed to be multiple runs for all dynamic intensities where no dynamic events were detected ($PR = (0, 0)$). The parameter sets for the medium dynamic intensity $C = 0.25$ are able to obtain the best values for precision, when compared to the reference set. Although the highest dynamic intensity shows that many H-SPPBO runs have detected all the dynamic events, reaching $recall = 1$, the precision of these averaged runs was as low as $precision = 0.13$. This indicates a very high amount of false detections (FP) and thus false triggers of the change handling procedure. This is also partially true for the lowest dynamic intensity $C = 0.1$, but the precision is slightly better, with the lowest value being $precision = 0.33$, meaning that every third change handling procedure triggered was not a correct response to the dynamic event. This behavior could indicate that only in the case of medium dynamic intensity the HPO considered a correct dynamic response as important enough for the goal of solution quality that it properly optimized for it.



(a) HPO



(b) Reference

Figure 6.23: Precision-Recall scatter plots showing the values for recall on the x-axis and precision on the y-axis separated for each dynamic intensity C (annotation at the top of the plots) for the HPO (left) and the reference (right) parameter sets.

In contrast, especially with the choice of parameter values discussed before in mind (see Figure 6.9), the robust, high β values for dynamic intensity $C = 0.5$ and the robust, comparatively high α values for dynamic intensity $C = 0.1$ suggest that for these dynamic environments, other parameters may be more important for good solution quality. This is also suggested in Figure 6.15, although the higher importance of the detection threshold for $C = 0.25$ lies in the error margin of the other two dynamic intensities.

This relationship between high precision and recall values and the detection threshold seems to hold even more information about the dynamic detection capabilities of the H-SPPBO metaheuristic. Figure 6.24 shows the precision over the recall values, but with an underlying heatmap showing the detection threshold θ for the HPO parameter set. Each heatmap cell represents the average θ value of all PR points falling in that region, also called a bin. This is why the background appears to be covered by low θ values of

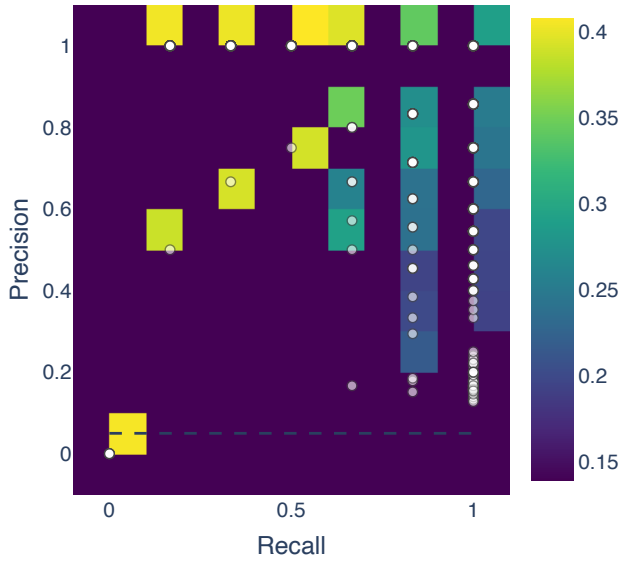


Figure 6.24: Precision-Recall heatmap showing the precision over the recall value with the z-axis displaying the detection threshold θ as a color map for the HPO experiments. The θ values for points in the same heatmap cell are averaged. In addition, the actual precision-recall points are overlayed to better seperate the darker areas.

0.15, because this is the minimum value available for averaging and can be ignored if no point lies directly on that cell. The worst performing runs with $PR = (0, 0)$ used a very high detection threshold of $\theta \approx 0.4$, which requires 5 or more swaps in the SCE tree to trigger H. This value seems to be too high in almost all scenarios, because even the runs with $precision > 0.5$ using these high θ values only achieved suboptimal recall scores of $recall < 0.5$. This means that even though most of the H triggers were correct responses to a dynamic event, more than half of the dynamic events went undetected. Looking at the near perfect upper right corner of $PR = (1, 1)$, a dynamic detection value of $\theta = 0.29$ seems to be the best choice, which is very similar to the median seen in Figure 6.8. While the recall remains high in the right section of the plot, the precision decreases with decreasing θ values.

Examining the PR plot for each individual TSP instance for the HPO parameter set, shown in Appendix A with Figure A.6, it appears that some instances achieved very good scores (e.g., the first two rows), while other instances had more problems with detecting dynamic events. However, this may very well be influenced by the parameters used rather than the problem instances themselves. Therefore, this visualization is not particularly significant and is only meant to further illustrate the effect of θ on the PR scores.

Another interesting observation that proves that the dynamic detection works as expected is the relationship between false positives (FP), which are triggers of the change handling procedure that were not in correct response to a dynamic event, and the detection

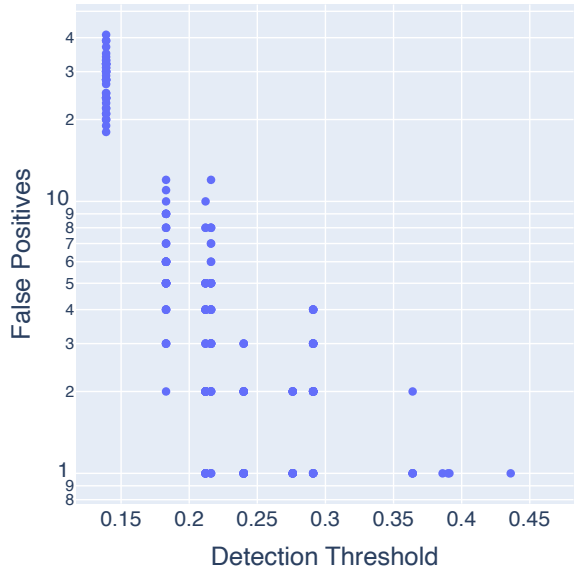


Figure 6.25: Scatter plot of all false dynamic detections by H-SPPBO (false positives) over the detection threshold θ for all 600 experimental runs using the HPO parameter sets, using a logarithmic scale for the y-axis.

threshold θ, shown in Figure 6.25. A logarithmic y-axis was chosen because the FPs increase exponentially with decreasing θ. Therefore, we can clearly see an almost linear diagonal line from the upper left to the lower right of the scatter plot, suggesting that to greatly reduce false detections (to a fraction of 1/4) from 40 to 10, we only need to slightly increase θ by 0.05 to $\theta = 0.19$,. However, any further reduction of FPs can only be achieved by even higher increases in θ until a point is reached where we only have no false positives because we are not detecting anything at all.

Finally, all of the precision and recall values are combined into a harmonic mean, the so-called F_1 score, which is high, if both precision and recall are good, while a low F_1 score indicates poor performance in at least one of these aspects. It is particularly useful when the two metrics are of equal importance, as is the case here. The score is shown in Table 6.8 for each combination of TSP instance and dynamic detection threshold C, their respective means $\overline{\text{TSP}}$ and $\overline{C}$, compared between the HPO parameter sets and the reference set, with additional standard error from averaging. Each pair of rows separated by a line represents a structural group.

The data shows better dynamic detection by the reference parameter set than the HPO sets in almost every direct comparison. Even more impressive, the reference value of $\theta = 0.25$ achieves values of $F_1 \geq 0.9$ in almost every combination of problem instance and dynamic intensity, with a total average over all values of $F_{1_{\text{Ref}}} = 0.92$. The total average of the HPO parameter sets is 26% lower at only $F_{1_{\text{HPO}}} = 0.68$. Only a few combinations of

Table 6.8: The F_1 score, i.e. the harmonic mean of the metrics precision and recall, and its standard error averaged over all $r_{\text{exp}} = 20$ experimental runs for each combination of TSP instance (rows) and dynamic intensity C (columns). Each element contains the values for the hyperparameter optimized parameters (HPO, see 6.5) and the reference parameter set (see 5.3).

TSP	$C = 0.10$		$C = 0.25$		$C = 0.50$		$\overline{C}$	
	HPO	Ref.	HPO	Ref.	HPO	Ref.	HPO	Ref.
eil51	0.56 ± 0.05	0.88 ± 0.02	0.89 ± 0.03	0.91 ± 0.02	0.65 ± 0.03	0.96 ± 0.01	0.70 ± 0.03	0.92 ± 0.01
rat195	0.63 ± 0.04	0.89 ± 0.02	0.86 ± 0.02	0.94 ± 0.01	0.65 ± 0.03	0.94 ± 0.01	0.71 ± 0.02	0.92 ± 0.01
berlin52	0.92 ± 0.01	0.84 ± 0.03	0.57 ± 0.05	0.95 ± 0.02	0.59 ± 0.05	0.93 ± 0.02	0.69 ± 0.03	0.91 ± 0.02
gil262	0.89 ± 0.03	0.91 ± 0.02	0.55 ± 0.05	0.88 ± 0.03	0.66 ± 0.04	0.96 ± 0.01	0.70 ± 0.03	0.92 ± 0.01
pr136	0.61 ± 0.04	0.89 ± 0.02	0.56 ± 0.05	0.92 ± 0.02	0.27 ± 0.01	0.96 ± 0.02	0.48 ± 0.03	0.92 ± 0.01
lin318	0.67 ± 0.03	0.87 ± 0.03	0.59 ± 0.04	0.94 ± 0.01	0.32 ± 0.01	0.94 ± 0.02	0.53 ± 0.03	0.92 ± 0.01
pr226	0.66 ± 0.01	0.91 ± 0.02	0.88 ± 0.02	0.91 ± 0.02	0.84 ± 0.03	0.94 ± 0.02	0.79 ± 0.02	0.92 ± 0.01
pr439	0.68 ± 0.02	0.95 ± 0.01	0.89 ± 0.02	0.92 ± 0.01	0.84 ± 0.02	0.93 ± 0.02	0.80 ± 0.02	0.93 ± 0.01
d198	0.87 ± 0.02	0.96 ± 0.01	0.58 ± 0.03	0.91 ± 0.03	0.76 ± 0.02	0.91 ± 0.02	0.73 ± 0.02	0.93 ± 0.01
fl417	0.75 ± 0.02	0.90 ± 0.02	0.47 ± 0.04	0.91 ± 0.02	0.76 ± 0.04	0.92 ± 0.02	0.66 ± 0.03	0.91 ± 0.01
$\overline{\text{TSP}}$	0.72 ± 0.01	0.90 ± 0.01	0.68 ± 0.02	0.92 ± 0.01	0.63 ± 0.02	0.94	0.68 ± 0.01	0.92

TSP instance and dynamic intensity, and therefore specific parameter sets, were able to achieve satisfactory results. These exceptions are the first structural group with $C = 0.25$ using a $\theta = 0.364$, the second structural group with $C = 0.1$ using $\theta = 0.276$, the fourth group with $C = 0.25$ using $\theta = 0.240$, and the fifth group with $C = 0.1$ using $\theta = 0.212$. However, these exceptions are still worse than the scores in the same category obtained by the reference set, except for the second structural group, specifically `berlin52` with $C = 0.1$, where $\theta = 0.276$, resulted in a F_1 score slightly better than that of the reference set.

In summary, the dynamic detection of the reference set using $\theta = 0.25$ almost always outperforms the parameter sets acquired by HPO. This suggests that by using only the solution quality as a metric for the response to dynamic changes, the HPO process, or at least the chosen GBRT method, generally does not evaluate the correct dynamic detection as relevant to the search process, and thus, as expected in hindsight, values good solutions above all else. The superior solution quality has already been discussed before, leaving no doubt about the success of the HPO process in that regard. However, it appears that unless the correct response to dynamic changes is explicitly made a requirement in the scoring function $f(\lambda)$, the process will not automatically include this goal in its search for well-performing parameters. The question then becomes whether this is even necessary if the solutions to the DTSP are still remarkably good.

7 Conclusion

In this thesis we successfully applied methods from Hyperparameter Optimization (HPO) to tune the parameters of the Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO) metaheuristic algorithm to solve the Dynamic Traveling Salesperson Problem (DTSP). Not only did the parameters obtained by HPO perform incredibly well in the dynamic phase of the DTSP, but the static performance, just before the problem was changed, also showed admirable performance, especially when compared to a standard set of reference parameters for H-SPPBO.

By planning and implementing all aspects necessary to answer the research questions, a new framework was derived, the EXperimentation Framework and (Hyper-)Parameter Optimization for Metaheuristics (XF-OPT/META). Its main goal is to combine a variety of HPO methods, such as Random Search (RS), Bayesian Optimization (BO), and Gradient Boosted Regression Trees (GBRT), with the H-SPPBO metaheuristic, all implemented in the *Python* programming language. Moreover, it provides modularity and interfaces for each of the main modules, the problem, the optimizer, and the metaheuristic, allowing easy extensibility and ensuring potential future development based on the XF-OPT/META framework. In light of the results discussed, it stands as a promising example of the potential that lies in the combination of HPO and metaheuristics.

The effort to select appropriate test environments to efficiently perform all the necessary experiments and to analyze the huge amount of data obtained also proved to be worthwhile. The ten Traveling Salesperson Problem (TSP) instances and the three dynamic intensities tested provided a well-rounded subset of problems. While the partitioning of the TSP instances from the *TSPLIB* benchmark set into structural groups was ultimately not observed in the data analyzed, the goal of using disjoint subsets to reduce the number of experiments was still achieved, with all groups successfully generalizing their parameter sets to their larger counterparts. The parameter constraints postulated and used in the optimization process also proved to be correct, with the exception of β, which showed a clear ceiling effect. And the four HPO methods tested presented a diverse selection, performing differently under given circumstances, each with its own advantages. The three parts of the experiments that were conducted all successfully served their purpose of answering the research questions, and during the data analysis no data was felt to be missing, due in large part to the automation of the process using a logging module

that captured all the raw data outputs of the processes, and an analysis module that was able to calculate complex tabular results as well as visualize large amounts of data with different aggregations. However, due to the many repetitions of the HPO procedure, and despite using two servers in parallel to compute the workload, the experiments took almost three months to complete.

The first research question, the choice of the ideal HPO method for the H-SPPBO metaheuristic, was answered in the first part of the results chapter. The arguments provided by convergence plots, area under the curve (AUC) and minimum solution quality metrics, and statistical tests made it clear that GBRT was the best choice for H-SPPBO, since it almost always found the best solutions, while converging to them the fastest out of the the four methods compared. Furthermore, Gaussian Process (GP) and Extra-Trees (ET) also showed very promising performance, with GP in particular, excelling in TSP instances of larger dimension.

The second question of what these best parameter sets look like was answered by the second part of the experiments. Here the results were more ambivalent. Although well-performing parameter sets were found for all specific combinations of problem instance and dynamic intensity, no clear recommendations for parameter values could be made under most considerations. Except for α and β, which also seemed to be the most important parameters for the trained GBRT surrogate model, and the favored use of the partial reset as the change handling procedure, only improved parameter value ranges were given for the remaining parameters, the three weights, and the detection threshold. In addition, no significant zero order correlation was found between these parameters.

In the final results chapter, all of these findings were combined - using the best HPO method to generate the best parameter values for each combination of problem instance and dynamic intensity - and compared to a well-performing reference parameter set. In terms of solution quality, the HPO parameters showed excellent results across all problem descriptions and throughout the runtime. However, it was found that this good solution quality was not due to the correct detection of dynamic changes, but was achieved despite their absence. The reference parameter set outperformed the HPO parameters in every discipline regarding dynamic detection, which was explained by either too low or too high choices for the detection threshold θ.

In summary, HPO has great potential for tuning the parameters of H-SPPBO and possibly metaheuristics in general. Different optimization methods have different effects on the resulting parameter values and on the problems on which they are obtained on. No single set of parameter values can be recommended to solve all DTSP problem descriptions, be they instance or dynamic. However, through the efficient application of HPO methods, such a general-purpose parameter set can easily be replaced by a parameter set specifically tuned for a given problem. In the case of H-SPPBO, it is

debatable whether good performance on the DTSP should be achieved by constructing highly greedy-influenced solution creating entities (SCEs), or by correct detection of dynamic events, and also which parameters influence the solution quality the most.

7.1 Future Work

On many occasions throughout the thesis, thoughts were given about possible future work and research. To further improve the solution quality, new parameter value ranges and fixed value recommendations were presented (see Section 6.2.1), which are an ideal starting point for new experiments with any kind of HPO method. In this context, the good performance of the GP method for TSP instances of larger dimension (> 150) could be further analyzed, especially in comparison with GBRT. Also, a cross-validation of HPO parameter sets obtained for different problem descriptions could be done to investigate, how specific they are tied to their original problem and how well they can be generalized. Since the optimization results focus on the parameters that lead to the best solution quality, the experiments were also prepared with this goal in mind. This raises two interesting topics that could not be answered with the available data. First, the tuning of the HPO process itself could significantly reduce the time needed for a parameter set. Thus, experiments should be conducted on the minimum number of objective calls required, different initial sampling methods, and other ways to make HPO more suitable for online use. The second possibility for future work to optimize the HPO process is the evaluation of different solution scoring functions $f(\lambda)$, which could strongly influence what kind of "good" solutions are found. This is also where the lack of dynamic performance can be addressed by including the correct detection of dynamic events in the evaluation function that gives feedback to HPO. Implementing the precision and recall metrics described above in the function would be one way to do this.

Finally, the data sets obtained in this thesis offer an intriguing possibility of replacing HPO by using machine learning (ML) models for parameter inference in a metaheuristic algorithm. The experiments resulted in numerous parameter sets that produced near-optimal solutions, along with detailed meta-information about each run, problem instance, city placement properties, and dynamic aspects. While the problem description was found to have a significant influence on the optimal parameters, direct correlations between parameter values and problems were not observed, suggesting higher-order relationships. Therefore, a ML model capable of capturing these complex relationships is proposed. The training set would include experimental data, particularly from the second part, with input features represented by meta-information about the problem environment (R, λ_1, or CQV). The dynamics could be described by specific measures (C or T_d) or generalized categories. The output or label would be the chosen parameters

for a given problem description. Two possible approaches for training the model can be used: supervised learning using the best parameter sets only, or reinforcement learning that also incorporates poorly performing sets. Supervised learning, while limited by available HPO data, offers easier preprocessing and implementation, potentially achieving satisfactory accuracy with simple methods. On the other hand, reinforcement learning requires more data and training time but may lead to better accuracy and generalization. Regardless of the chosen ML method, the trained model would enable inferring optimal parameter values for any TSP problem description.

However good this machine learning option might be, training such a model would be a challenge in and of itself, not to mention the training data that would be required. In contrast, the possibility of HPO for metaheuristics, discussed here and supported by evidence, should be the primary focus of future work.

Abstract

This master's thesis in computer science explores the application of Hyperparameter Optimization (HPO) methods to tune the parameters of the Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO) metaheuristic algorithm for solving the Dynamic Traveling Salesperson Problem (DTSP). The work combines various HPO methods with the H-SPPBO metaheuristic. The experiments were conducted using ten TSP instances and three dynamic intensities to realize the DTSP. The first research question about the ideal HPO method was answered by comparing the performance of four different methods and found GBRT to be the leading method. The second question investigated the best parameter sets for each combination of the problem instance and dynamic intensity, yielding interesting results about the robustness and importance of certain parameters, as well as the insight that no single parameter set is the best choice for all problem descriptions. The third part of the results compared the performance of the HPO parameters to a reference parameter set, showing that the HPO parameters performed excellently in the DTSP, but the reference set outperformed them in dynamic detection. Overall, the work demonstrates the potential of HPO in tuning metaheuristic parameters.

***Keywords*—** Hyperparameter Optimization (HPO), Metaheuristics, Parameter Tuning, Dynamic Traveling Salesperson Problem (DTSP), Hierarchical Simple Probabilistic Population-Based Optimization (H-SPPBO), Python, Random Search (RS), Bayesian Optimization (BO), Gaussian Process (GP), Extra-Trees (ET), Gradient Boosted Regression Trees (GBRT)

Bibliography

[1] U. Vogel, "A flexible metaheuristic framework for solving rich vehicle routing problems: Formulierung, implementierung und anwendung eines kognitionsbasierten simulationsmodells," Ph.D. dissertation, Universität zu Köln, Diss., 2011.

[2] V. Sharma, A. K. Tripathi, "A systematic review of meta-heuristic algorithms in IoT based application," *Array*, vol. 14, p. 100 164, 2022.

[3] 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," *Philippine Computing Journal*, vol. 4, no. 2, pp. 1–9, 2009.

[4] L. Yang, A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[5] D. H. Wolpert, W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[6] K. Sörensen, M. Sevaux, F. Glover, "A history of metaheuristics," in *Handbook of Heuristics*, Springer, 2018, pp. 791–808.

[7] Y.-C. Lin, M. Clauss, M. Middendorf, "Simple probabilistic population-based optimization," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 2, pp. 245–262, 2015.

[8] G. Reinelt, "TSPLIB—a traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.

[9] R. E. Burkard, S. E. Karisch, F. Rendl, "QAPLIB – a quadratic assignment problem library," *Journal of Global Optimization*, vol. 10, pp. 391–403, 1997.

[10] 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.

[11] S. Janson, M. Middendorf, "A hierarchical particle swarm optimizer," in *The 2003 Congress on Evolutionary Computation.*, IEEE, vol. 2, 2003, pp. 770–776.

[12] 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.

[13] H. N. Psaraftis, "Dynamic vehicle routing: Status and prospects," *Annals of Operations Research*, vol. 61, no. 1, pp. 143–164, 1995.

[14] M. Feurer, F. Hutter, "Hyperparameter optimization," in *Automated Machine Learning: Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, J. Vanschoren, Eds. Springer International Publishing, Cham, 2019, pp. 3–33.

[15] H. N. Psaraftis, "Dynamic vehicle routing problems," *Vehicle Routing: Methods and Studies*, vol. 16, pp. 223–248, 1988.

[16] G. A. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958.

[17] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245–2269, 1965.

[18] S. Lin, B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, no. 2, pp. 498–516, 1973.

[19] 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)*, IEEE, vol. 2, 2001, pp. 1283–1288.

[20] 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.

[21] 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.

[22] 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.

[23] C. A. Silva, T. A. Runkler, "Ant colony optimization for dynamic traveling salesman problems," *ARCS 2004–Organic and pervasive computing*, 2004.

[24] M. Mavrovouniotis, S. Yang, "Ant colony optimization with immigrants schemes for the dynamic travelling salesman problem with traffic factors," *Applied Soft Computing*, vol. 13, no. 10, pp. 4023–4037, 2013.

[25] M. Mavrovouniotis, F. M. Müller, S. Yang, "Ant colony optimization with local search for dynamic traveling salesman problems," *IEEE Transactions on Cybernetics*, vol. 47, no. 7, pp. 1743–1756, 2016.

[26] 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.

[27] 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.

[28] 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.

[29] S. Janson, M. Middendorf, "A hierarchical particle swarm optimizer for noisy and dynamic environments," *Genetic Programming and Evolvable Machines*, vol. 7, pp. 329–354, 2006.

[30] Á. E. Eiben, R. Hinterding, Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999.

[31] E.-G. Talbi, *Metaheuristics: from design to implementation* (Wiley Series on Parallel and Distributed Computing). Hoboken, NJ: Wiley-Blackwell, 2009.

[32] 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.

[33] M. Dorigo, A. Colorni, V. Maniezzo, "Ant system: An autocatalytic optimizing process," *Dipartimento Di Elettronica, Politecnico Di Milano, Milan, Italy*, 1991.

[34] M. Dorigo, V. Maniezzo, A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[35] M. Dorigo, T. Stützle, *Ant Colony Optimization*. Cambridge, MA: The MIT Press, Jun. 2004. DOI: 10.7551/mitpress/1290.001.0001.

[36] D. Gaertner, K. L. Clark, "On optimal parameters for ant colony optimization algorithms," vol. 1, 2005, pp. 83–89.

[37] 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 Cham, 2018, pp. 144–161.

[38] T. Stützle, M. López-Ibánez, P. Pellegrini, M. Maur, M. Montes de Oca, M. Birattari, M. Dorigo, "Parameter adaptation in ant colony optimization," in *Autonomous Search*, Y. Hamadi, E. Monfroy, F. Saubion, Eds. Berlin, Heidelberg: Springer, 2012.

[39] 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*, O. Castillo, P. Melin, J. Kacprzyk, Eds. Berlin, Heidelberg: Springer, 2013, pp. 259–271.

[40] 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.

[41] P. Li, H. Zhu, "Parameter selection for ant colony algorithm based on bacterial foraging algorithm," *Mathematical Problems in Engineering*, vol. 2016, 2016.

[42] M. Randall, "Near parameter free ant colony optimisation," in *Ant Colony Optimization and Swarm Intelligence*, Springer, 2004, pp. 374–381.

[43] M. Birattari, J. Kacprzyk, *Tuning Metaheuristics: A Machine Learning Perspective*. Berlin, Heidelberg: Springer, 2009.

[44] O. Maron, A. Moore, "Hoeffding races: Accelerating model selection search for classification and function approximation," in *Advances in Neural Information Processing Systems*, vol. 6, Morgan-Kaufmann, 1993.

[45] 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.

[46] M. Packianather, P. Drake, H. Rowlands, "Optimizing the parameters of multilayered feedforward neural networks through taguchi design of experiments," *Quality and Reliability Engineering International*, vol. 16, no. 6, pp. 461–473, 2000.

[47] A. Tortum, N. Yayla, C. Çelik, M. Gökdağ, "The investigation of model selection criteria in artificial neural networks by the taguchi method," *Physica A: Statistical Mechanics and its Applications*, vol. 386, no. 1, pp. 446–468, 2007.

[48] J.-R. Jung, B.-J. Yum, "Artificial neural network based approach for dynamic parameter design," *Expert Systems with Applications*, vol. 38, no. 1, pp. 504–510, 2011.

[49] E. Yin, K. Wijk, "Bayesian parameter tuning of the ant colony optimization algorithm: Applied to the asymmetric traveling salesman problem," Bachelor's Thesis, 2021.

[50] J. Robinson, "On the hamiltonian game (a traveling salesman problem)," RAND Project Air Force Arlington VA, Tech. Rep., 1949.

[51] A. Schrijver, "On the history of combinatorial optimization (till 1960)," *Handbooks in Operations Research and Management Science*, vol. 12, pp. 1–68, 2005.

[52] E. L. Lawler, *The traveling salesman problem: a guided tour of combinatorial optimization* (Wiley Series in Discrete Mathematics & Optimization). Chichester, England: John Wiley & Sons, 1985.

[53] 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*, Association for Computing Machinery, 1976, pp. 1–9.

[54] A. P. Punnen, "The traveling salesman problem: Applications, formulations and variations," in *The Traveling Salesman Problem and Its Variations*, G. Gutin, A. P. Punnen, Eds. Boston, MA: Springer US, 2007, pp. 1–28.

[55] 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*, G. Gutin, A. P. Punnen, Eds. Boston, MA: Springer US, 2007, pp. 445–487.

[56] K. Sörensen, F. Glover, "Metaheuristics," in *Encyclopedia of Operations Research and Management Science*, S. I. Gass, M. C. Fu, Eds. Boston, MA: Springer US, 2013, vol. 62, pp. 960–970.

[57] I. Boussaid, J. Lepagnot, P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82–117, 2013.

[58] M. Gendreau, J.-Y. Potvin, *et al.*, *Handbook of Metaheuristics*. New York, NY: Springer, 2010, vol. 2.

[59] C. Blum, A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, 2003.

[60] S. Goss, S. Aron, J.-L. Deneubourg, J. M. Pasteels, "Self-organized shortcuts in the argentine ant," *Naturwissenschaften*, vol. 76, no. 12, pp. 579–581, 1989.

[61] M. Dorigo, T. Stützle, "Ant colony optimization: Overview and recent advances," in *Handbook of Metaheuristics*, M. Gendreau, J.-Y. Potvin, Eds. Springer, Cham, 2019, pp. 311–351.

[62] M. Guntsch, M. Middendorf, "A population based approach for ACO," in *Applications of Evolutionary Computing*, Springer Berlin Heidelberg, 2002, pp. 72–81.

[63] J. Kennedy, R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, IEEE, vol. 4, 1995, pp. 1942–1948.

[64] W.-C. Yeh, "A two-stage discrete particle swarm optimization for the problem of multiple multi-level redundancy allocation in series systems," *Expert Systems with Applications*, vol. 36, no. 5, pp. 9192–9200, 2009.

[65] W.-C. Yeh, "Simplified swarm optimization in disassembly sequencing problems with learning effects," *Computers & Operations Research*, vol. 39, no. 9, pp. 2168–2177, 2012.

[66] J. Bergstra, Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of Machine Learning Research*, vol. 13, no. 2, 2012.

[67] 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," *arXiv preprint arXiv:1012.2599*, 2010.

[68] C. K. Williams, C. E. Rasmussen, *Gaussian Processes for Machine Learning*. Cambridge, MA: MIT Press, 2006, vol. 2.

[69] J. Snoek, H. Larochelle, R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc., 2012.

[70] H. J. Kushner, "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise," *Journal of Basic Engineering*, vol. 86, no. 1, pp. 97–106, 1964.

[71] D. R. Jones, M. Schonlau, W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[72] M. L. Stein, *Interpolation of Spatial Data: Some Theory for Kriging*. New York, NY: Springer, 1999.

[73] A. Cutler, D. R. Cutler, J. R. Stevens, "Random forests," in *Ensemble Machine Learning: Methods and Applications*, C. Zhang, Y. Ma, Eds. New York, NY: Springer, 2012, pp. 157–175.

[74] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 2001.

[75] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, pp. 123–140, 1996.

[76] T. K. Ho, "A data complexity analysis of comparative advantages of decision forest constructors," *Pattern Analysis & Applications*, vol. 5, pp. 102–112, 2002.

[77] P. Geurts, D. Ernst, L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, pp. 3–42, 2006.

[78] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, pp. 1189–1232, 2001.

[79] H. Deng, Y. Zhou, L. Wang, C. Zhang, "Ensemble learning for the early prediction of neonatal jaundice with genetic features," *BMC Medical Informatics and Decision Making*, vol. 21, pp. 1–11, 2021.

[80] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[81] T. Hastie, R. Tibshirani, J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York, NY: Springer, 2009, vol. 2.

[82] A. Mohan, Z. Chen, K. Weinberger, "Web-search ranking with initialized gradient boosted regression trees," in *Proceedings of the Learning to Rank Challenge*, ser. Proceedings of Machine Learning Research, vol. 14, PMLR, 2011, pp. 77–89.

[83] 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," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[84] S. AI, *MPIRE (multiprocessing is really easy)*, https://github.com/Slimmer-AI/mpire, 2023.

[85] X. Chen, *Treelib*, https://github.com/caesar0301/treelib, 2018.

[86] P. J. Clark, F. C. Evans, "Distance to nearest neighbor as a measure of spatial relationships in populations," *Ecology*, vol. 35, no. 4, pp. 445–453, 1954.

[87] G. C. Crişan, E. Nechita, D. Simian, "On randomness and structure in euclidean TSP instances: A study with heuristic methods," *IEEE Access*, vol. 9, pp. 5312–5331, 2021.

[88] 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," *The Journal of Problem Solving*, vol. 4, no. 1, pp. 1–17, 2012.

[89] A. Hagberg, P. Swart, D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[90] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, I. Shcherbatyi, "Scikit-optimize/scikit-optimize: V0. 8.1," *Zenodo*, 2020.

[91] 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," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[92] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, M. A. Aivazis, "Building a framework for predictive science," *arXiv preprint arXiv:1202.1056*, 2012.

[93] D. G. Bonett, "Confidence interval for a coefficient of quartile variation," *Computational Statistics & Data Analysis*, vol. 50, no. 11, pp. 2953–2957, 2006.

[94] 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," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[95] D. Cvetković, Z. Dražić, V. Kovačević-Vujčić, M. Čangalović, "The traveling salesman problem," *Bulletin (Académie serbe des sciences et des arts. Classe des sciences mathématiques et naturelles. Sciences mathématiques)*, no. 43, pp. 17–26, 2018.

[96] L. Lovász, "Eigenvalues of graphs," *Lecture notes*, 2007.

[97] S. Hougardy, X. Zhong, "Hard to solve instances of the euclidean traveling salesman problem," *Mathematical Programming Computation*, vol. 13, pp. 51–74, 2021.

[98] D. Arthur, S. Vassilvitskii, "K-means++: The advantages of careful seeding," Stanford, Tech. Rep., 2006.

[99] S. Lloyd, "Least squares quantization in PCM," *IEEE transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[100] C. Cervellera, D. Macciò, "Learning with kernel smoothing models and low-discrepancy sampling," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 24, no. 3, pp. 504–509, 2013.

[101] W. P. Vogt, B. Johnson, *Dictionary of statistics methodology: A nontechnical guide for the social sciences*, 4th ed. Thousand Oaks, CA: SAGE Publications, 2011.

[102] T. Head. "Comparing surrogate models." (2016), [Online]. Available: https://scikit-optimize.github.io/stable/auto_examples/strategy-comparison.html#comparing-surrogate-models (visited on 03/23/2023).

[103] W. J. Conover, R. L. Iman, "On multiple-comparisons procedures," *Los Alamos Sci. Lab. Tech. Rep. LA-7677-MS*, vol. 1, p. 14, 1979.

[104] W. J. Conover, "Practical nonparametric statistics," in (Wiley Series in Probability and Statistics), 3rd ed., Wiley Series in Probability and Statistics. Nashville, TN: John Wiley & Sons, 1999, ch. 5.3.

[105] M. Terpilowski, "Scikit-posthocs: Pairwise multiple comparison tests in Python," *The Journal of Open Source Software*, vol. 4, no. 36, p. 1169, 2019. DOI: 10.21105/joss.01169.

[106] C. Strobl, A.-L. Boulesteix, A. Zeileis, T. Hothorn, "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC Bioinformatics*, vol. 8, no. 1, pp. 1–21, 2007.

Acronyms

ACO Ant Colony Optimization. 9, 18

ANN artificial neural network. 11

AUC area under the curve. 75, 130

BO Bayesian Optimization. 11, 33, 45, 129

CDF cumulative distribution function. 37

CQV coefficient of quartile variation. 49

DoE Design of Experiment. 11, 31

DTSP Dynamic Traveling Salesperson Problem. 6, 15, 129

EA Evolutionary Algorithm. 18

EI expected improvement. 37

ET Extra-Trees. 40, 130

FP false positives. 78, 148

GA Genetic Algorithm. 5

GBRT Gradient Boosted Regression Trees. 35, 129

GP Gaussian Process. 35, 130

GS Grid Search. 33

HPO Hyperparameter Optimization. 7, 32, 45, 129

H-PSO Hierarchical Particle Swarm Optimization. 6, 23

H-SPPBO Hierarchical Simple Probabilistic Population-Based Optimization. 6, 26, 45, 129

IQR interquartile range. 88

LCB lower confidence bound. 66

MDI Mean Decrease in Impurity. 78

ML machine learning. 5, 31, 131

PDF probability density function. 37

PI probability of improvement. 37

P-metaheuristic Population-Based Metaheuristic. 17

PACO Population-Based Ant Colony Optimization. 6, 21, 147

PR Precision-Recall. 79

PSO Particle Swarm Optimization. 5, 21, 147

QAP Quadratic Assignment Problem. 5, 27

RF Random Forests. 35

RS Random Search. 11, 33, 129

SA Simulated Annealing. 17

S-metaheuristic Single-Solution Based Metaheuristic. 16

SCE solution creating entity. 6, 25, 131

SPPBO Simple Probabilistic Population-Based Optimization. 5, 25

SSO Simplified Swarm Optimization. 6, 24

TP true positives. 78

TS Tabu Search. 17

TSP Traveling Salesperson Problem. 5, 13, 129

XF-OPT/META EXperimentation Framework and (Hyper-)Parameter Optimization for Metaheuristics. 52, 129

List of Figures

List of Tables

List of Listings

List of Algorithms

A Additional Figures

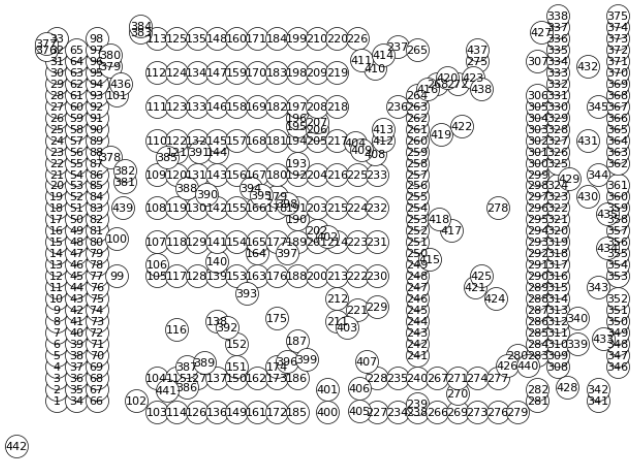


Figure A.1: Visualization of the TSP instance *pcb442*.

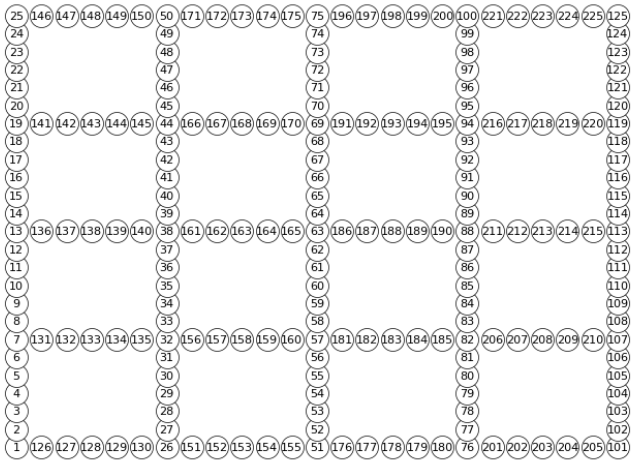


Figure A.2: Visualization of the TSP instance *ts225*.

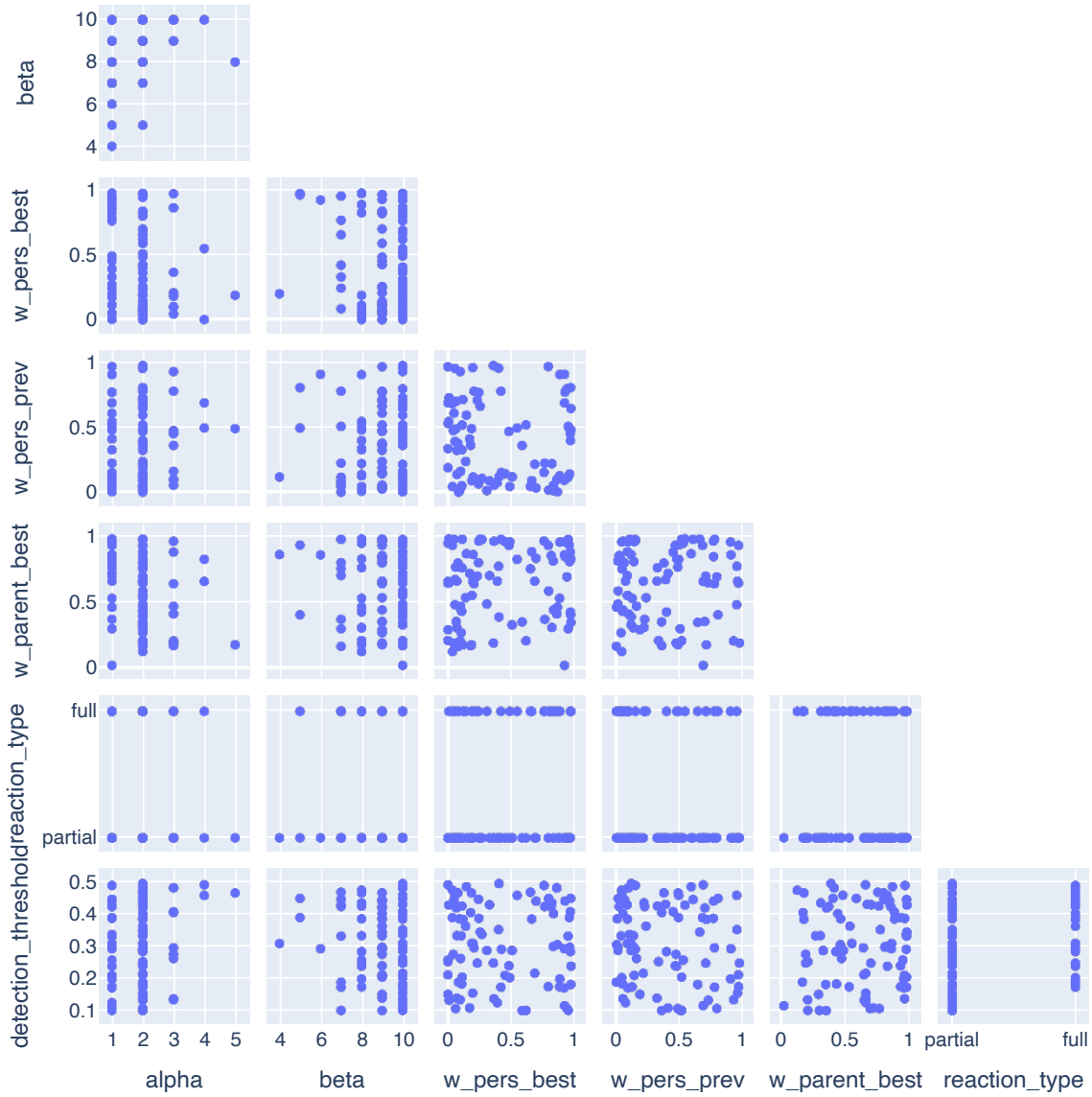


Figure A.3: Scatter matrix plot of all H-SPPBO parameters (except H) over all 90 optimizer runs with reaction type.

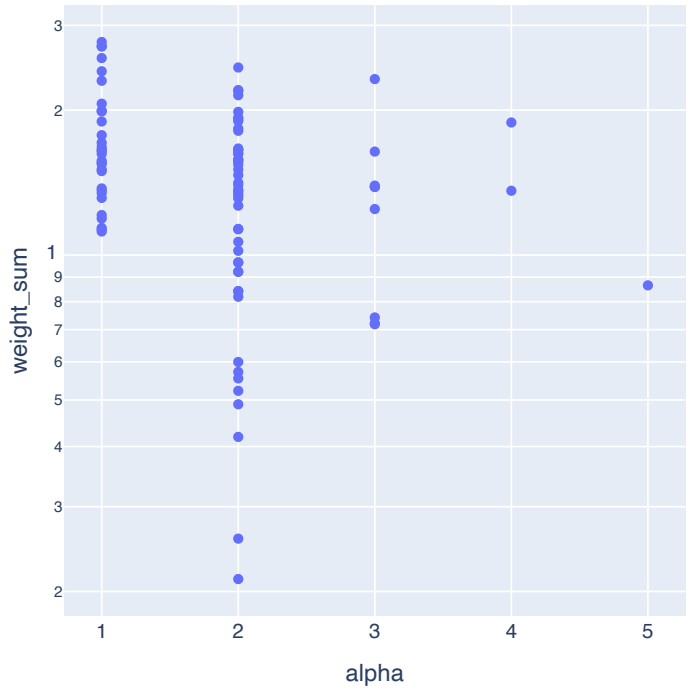


Figure A.4: Scatter plot of all α-values over the sum of the three weights on a logarithmic scale over all 90 optimizer runs.

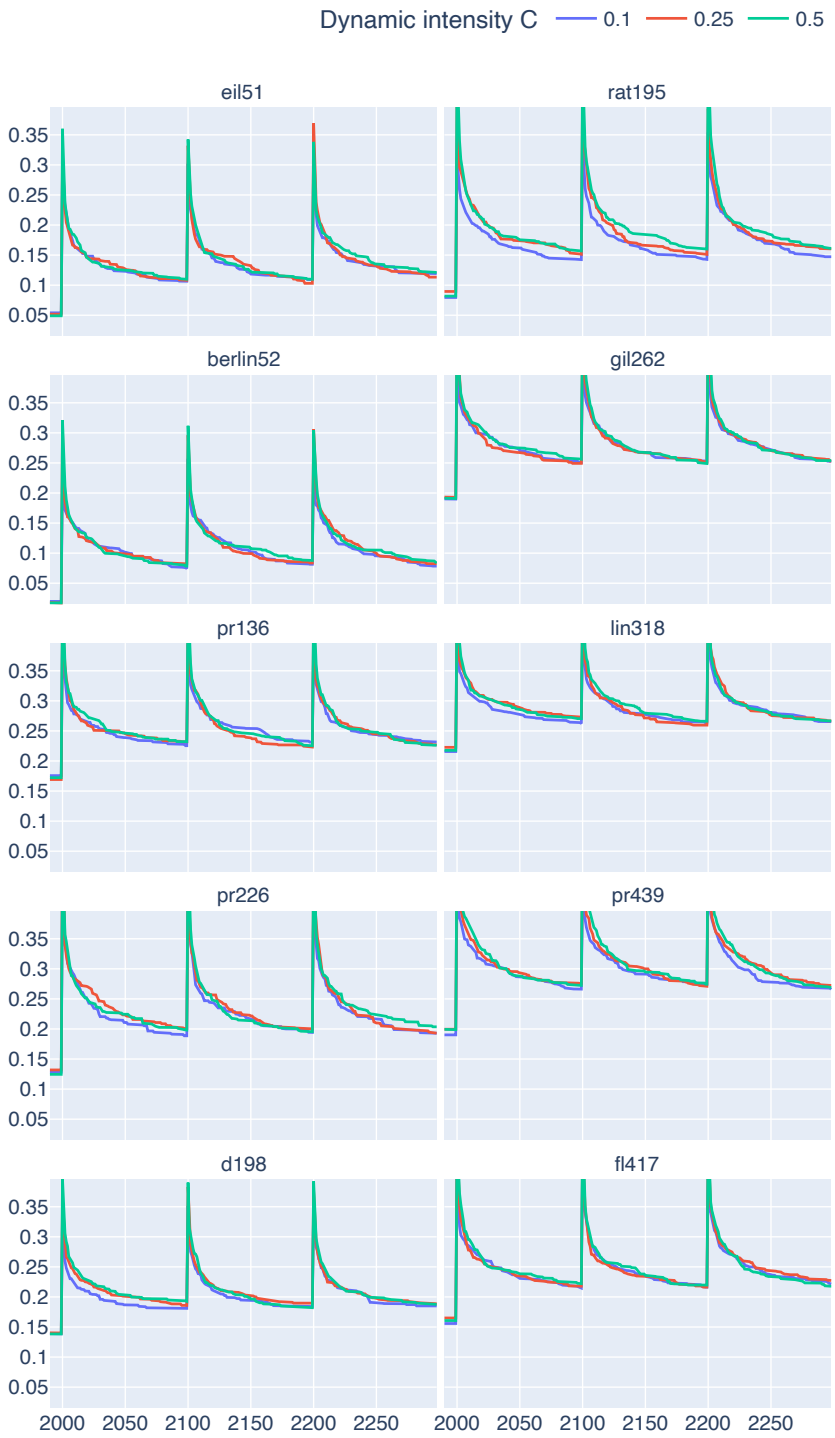


Figure A.5: Line plots showing the relative solution quality *RPD* over the first 300 dynamic iterations (starting just before iteration 2000) for all three dynamic intensities C and all ten TSP instances for the reference experiments. The smaller instances are in the left column and the larger instances are in the right column. Each row corresponds to the predefined structural groups (see Figure 5.2).

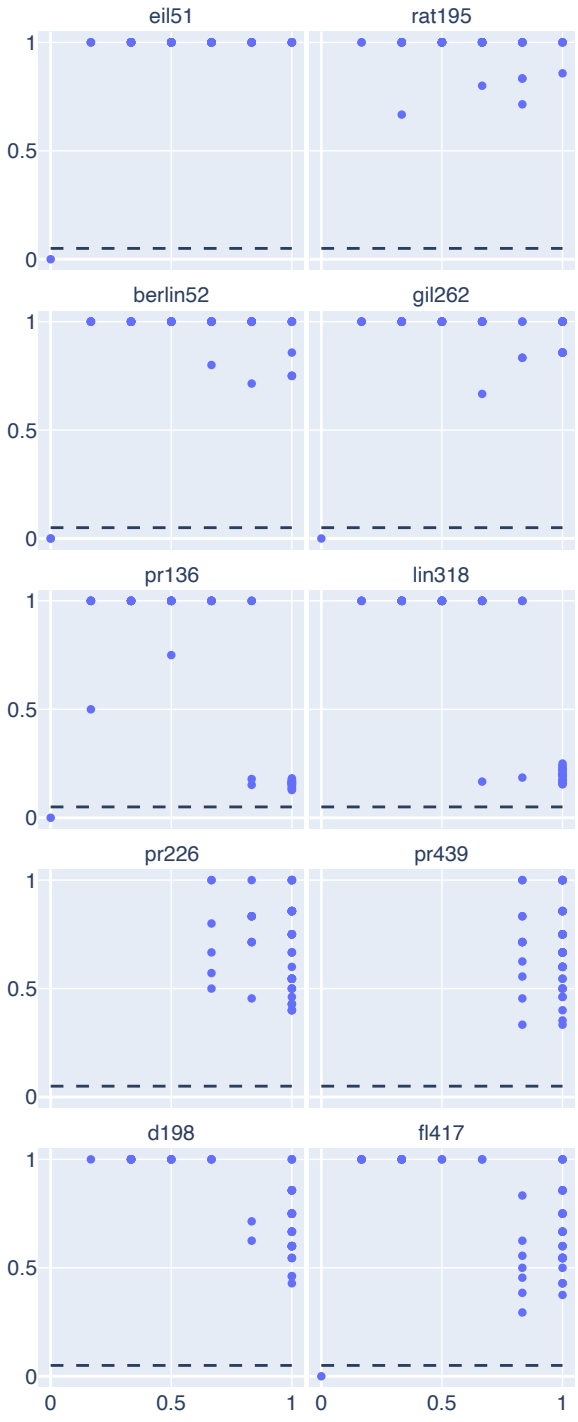


Figure A.6: Precision-Recall scatter plots showing the values for recall on the x-axis and precision on the y-axis separated for each of the 10 TSP instances (name atop the plots) for the HPO parameter sets. The smaller instances are in the left column and the larger instances are in the right column. Each row corresponds to the predefined structural groups (see Figure 5.2).

B Additional Tables

Table B.1: Statistical values from the parameter box plots in Figure 6.8.

Quartile	α	β	w_{persbest}	w_{persprev}	$w_{\text{parentbest}}$	θ
Q_0	1	4	0.002	0.001	0.024	0.100
Q_1	1	8	0.108	0.098	0.398	0.203
Q_2	2	9	0.367	0.350	0.667	0.308
Q_3	2	10	0.821	0.662	0.874	0.426
Q_4	5	10	0.984	0.982	0.989	0.497

Table B.2: Statistical values from the parameter box plots in Figure 6.9, grouped over dynamic intensity C.

C	Quartile	α	β	w_{persbest}	w_{persprev}	$w_{\text{parentbest}}$	θ
0.1	Q_0	1	4	0.002	0.008	0.181	0.106
	Q_1	2	7	0.101	0.103	0.422	0.266
	Q_2	2	10	0.252	0.436	0.671	0.349
	Q_3	2	10	0.693	0.686	0.814	0.446
	Q_4	5	10	0.980	0.982	0.988	0.493
0.25	Q_0	1	7	0.002	0.001	0.130	0.100
	Q_1	1	8	0.076	0.063	0.300	0.239
	Q_2	2	9	0.281	0.230	0.550	0.349
	Q_3	2	10	0.820	0.512	0.927	0.417
	Q_4	3	10	0.984	0.975	0.986	0.497
0.5	Q_0	1	6	0.009	0.003	0.024	0.100
	Q_1	1	9	0.217	0.131	0.488	0.172
	Q_2	1	10	0.442	0.382	0.667	0.266
	Q_3	2	10	0.888	0.708	0.968	0.330
	Q_4	3	10	0.982	0.967	0.989	0.491

Table B.3: Statistical values from the parameter box plots in Figure 6.10, grouped over TSP problem instance.

TSP	Quartile	α	β	w_{persbest}	w_{persprev}	$w_{\text{parentbest}}$	θ
eil51	Q_0	1	5	0.009	0.001	0.170	0.101
	Q_1	1	7	0.101	0.292	0.281	0.194
	Q_2	2	9	0.195	0.504	0.597	0.333
	Q_3	2	10	0.961	0.792	0.883	0.404
	Q_4	5	10	0.984	0.967	0.989	0.467
berlin52	Q_0	1	4	0.045	0.003	0.196	0.100
	Q_1	1	9	0.163	0.086	0.412	0.300
	Q_2	2	9	0.354	0.188	0.775	0.404
	Q_3	2	10	0.732	0.579	0.867	0.444
	Q_4	3	10	0.972	0.962	0.986	0.497
pr136	Q_0	1	7	0.002	0.008	0.174	0.100
	Q_1	2	8	0.084	0.099	0.339	0.193
	Q_2	2	9	0.349	0.276	0.503	0.298
	Q_3	2	10	0.677	0.376	0.781	0.402
	Q_4	3	10	0.979	0.982	0.986	0.468
pr226	Q_0	1	8	0.003	0.014	0.024	0.106
	Q_1	1	9	0.074	0.156	0.373	0.158
	Q_2	2	10	0.217	0.442	0.713	0.222
	Q_3	2	10	0.759	0.621	0.952	0.290
	Q_4	3	10	0.982	0.912	0.988	0.476
d198	Q_0	1	7	0.002	0.017	0.417	0.137
	Q_1	2	9	0.245	0.063	0.606	0.224
	Q_2	2	10	0.606	0.102	0.686	0.362
	Q_3	3	10	0.804	0.708	0.882	0.457
	Q_4	4	10	0.935	0.975	0.985	0.493

Table B.4: Information retrieval or classification performance metrics averaged over all $r_{\text{exp}} = 20$ experimental runs for each combination of TSP instance (rows) and dynamic intensity C (columns) for the HPO parameter sets (see 6.5).

TSP	C	True Positives	False Positives	False Positive Rate	Recall	Precision	F_1
eil51	0.10	3	0	0.000	0.42	0.95	0.56 ± 0.05
	0.25	5	0	0.000	0.82	1.00	0.89 ± 0.03
	0.50	3	0	0.000	0.50	1.00	0.65 ± 0.03
rat195	0.10	3	0	0.000	0.48	0.98	0.63 ± 0.04
	0.25	5	0	0.001	0.80	0.95	0.86 ± 0.02
	0.50	3	0	0.000	0.50	1.00	0.65 ± 0.03
berlin52	0.10	6	0	0.001	0.92	0.94	0.92 ± 0.01
	0.25	3	0	0.000	0.43	0.94	0.57 ± 0.05
	0.50	3	0	0.000	0.44	0.95	0.59 ± 0.05
gil262	0.10	5	1	0.001	0.89	0.92	0.89 ± 0.03
	0.25	2	0	0.000	0.40	0.95	0.55 ± 0.05
	0.50	3	0	0.000	0.53	1.00	0.66 ± 0.04
pr136	0.10	3	0	0.000	0.47	0.98	0.61 ± 0.04
	0.25	3	0	0.000	0.43	0.94	0.56 ± 0.05
	0.50	6	32	0.056	0.98	0.16	0.27 ± 0.01
lin318	0.10	3	0	0.000	0.52	1.00	0.67 ± 0.03
	0.25	3	0	0.000	0.44	1.00	0.59 ± 0.04
	0.50	6	25	0.043	0.98	0.20	0.32 ± 0.01
pr226	0.10	6	6	0.011	0.99	0.49	0.66 ± 0.01
	0.25	5	1	0.001	0.90	0.88	0.88 ± 0.02
	0.50	6	2	0.003	0.93	0.77	0.84 ± 0.03
pr439	0.10	6	6	0.010	0.98	0.53	0.68 ± 0.02
	0.25	6	1	0.002	0.96	0.84	0.89 ± 0.02
	0.50	6	2	0.004	0.96	0.76	0.84 ± 0.02
d198	0.10	6	2	0.003	0.98	0.78	0.87 ± 0.02
	0.25	3	0	0.000	0.42	1.00	0.58 ± 0.03
	0.50	6	4	0.007	1.00	0.62	0.76 ± 0.02
fl417	0.10	6	4	0.007	0.98	0.62	0.75 ± 0.02
	0.25	2	0	0.000	0.33	0.95	0.47 ± 0.04
	0.50	6	4	0.007	0.97	0.65	0.76 ± 0.04

Table B.5: Information retrieval or classification performance metrics averaged over all $r_{\text{exp}} = 20$ experimental runs for each combination of TSP instance (rows) and dynamic intensity C (columns) for the reference parameter set (see 5.3).

TSP	C	True Positives	False Positives	False Positive Rate	Recall	Precision	F_1
eil51	0.10	5	0	0.000	0.82	0.98	0.88 ± 0.02
	0.25	5	0	0.000	0.86	0.98	0.91 ± 0.02
	0.50	6	0	0.000	0.94	0.98	0.96 ± 0.01
rat195	0.10	5	0	0.000	0.83	0.98	0.89 ± 0.02
	0.25	6	0	0.001	0.94	0.94	0.94 ± 0.01
	0.50	6	0	0.001	0.95	0.94	0.94 ± 0.01
berlin52	0.10	5	0	0.000	0.76	0.98	0.84 ± 0.03
	0.25	6	0	0.000	0.93	0.98	0.95 ± 0.02
	0.50	5	0	0.000	0.89	0.98	0.93 ± 0.02
gil262	0.10	5	0	0.000	0.87	0.97	0.91 ± 0.02
	0.25	5	0	0.001	0.85	0.95	0.88 ± 0.03
	0.50	6	0	0.000	0.95	0.98	0.96 ± 0.01
pr136	0.10	5	0	0.000	0.83	0.99	0.89 ± 0.02
	0.25	5	0	0.000	0.87	0.98	0.92 ± 0.02
	0.50	6	0	0.000	0.95	0.98	0.96 ± 0.02
lin318	0.10	5	0	0.000	0.83	0.96	0.87 ± 0.03
	0.25	5	0	0.000	0.91	0.99	0.94 ± 0.01
	0.50	6	0	0.001	0.95	0.94	0.94 ± 0.02
pr226	0.10	5	0	0.000	0.85	1.00	0.91 ± 0.02
	0.25	5	0	0.000	0.87	0.98	0.91 ± 0.02
	0.50	5	0	0.000	0.91	0.98	0.94 ± 0.02
pr439	0.10	6	0	0.000	0.93	0.98	0.95 ± 0.01
	0.25	5	0	0.001	0.91	0.94	0.92 ± 0.01
	0.50	5	0	0.001	0.91	0.96	0.93 ± 0.02
d198	0.10	6	0	0.000	0.93	1.00	0.96 ± 0.01
	0.25	5	0	0.000	0.87	0.97	0.91 ± 0.03
	0.50	5	0	0.001	0.89	0.93	0.91 ± 0.02
fl417	0.10	5	0	0.000	0.84	0.98	0.90 ± 0.02
	0.25	5	0	0.000	0.87	0.98	0.91 ± 0.02
	0.50	5	0	0.000	0.88	0.98	0.92 ± 0.02

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Ort:	Datum:	Unterschrift: