TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

From Parameter Tuning to Dynamic Heuristic Selection

Yevhenii Semendiak

Yevhenii.Semendiak@tu-dresden.de
Born on: 7th February 1995 in Izyaslav
Course: Distributed Systems Engineering
Matriculation number: 4733680
Matriculation year: 2020

Master Thesis

to achieve the academic degree

Master of Science (M.Sc.)

Supervisors
MSc. Dmytro Pukhkaiev
Dr. Sebastian Götz

Supervising professor
Prof. Dr. rer. nat habil. Uwe Aßmann

Submitted on: 11th May 2020

Technische Universität Dresden/Fakultät Informatik

Aufgabenstellung für die Masterarbeit

Name, Vorname: Semendiak, Yevhenii

Studiengang: Master DSE Matr. Nr.: 4 7 3 3 6 8 0

Thema:

From Parameter Tuning to Dynamic Heuristic Selection

Zielstellung :

Metaheuristic-based solvers are widely used in solving combinatorial optimization problems. A choice of an underlying metaheuristic is crucial to achieve high quality of the solution and performance. A combination of several metaheuristics in a single hybrid heuristic proved to be a successful design decision. State-of-the-art hybridization approaches consider it as a design time problem, whilst leaving a choice of an optimal heuristics combination and its parameter settings to parameter tuning approaches. The goal of this thesis is to extend a software product line for parameter tuning with dynamic heuristic selection; thus, allowing to adapt heuristics at runtime. The research objective is to investigate whether dynamic selection of an optimization heuristic can positively effect performance and scalability of a metaheuristic-based solver.

For this thesis, the following tasks have to be fulfilled:

- Literature analysis covering closely related work.
- Development of a strategy for online heuristic selection.
- Implementation of the developed strategy.
- Evaluation of the developed approach based on a synthetic benchmark.
- (Optional) Evaluation of the developed approach with a problem of software variant selection and hardware resource allocation.

Betreuer: M.Sc. Dmytro Pukhkaiev, Dr.-Ing. Sebastian Götz

Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. habil. Uwe Aßmann

Institut: Software- und Multimediatechnik

Beginn am : 01.10.2019

Einzureichen am : 09.03.2020

Unterschrift des verantwortlichen Hochschullehrers

Verteiler: 1 x SCIS, 1x HSL, 1x Betreuer, 1x Student

Contents

List of Figures

List of Tables

0.1 abstract

Abstract will be available in final versions of thesis.

1 Introduction

Intent and content of chapter. This chapter is an self-descriptive, shorten version of thesis.

1.1 Motivation

Structure:

- optimization problem(OP) → exact or approximate (+description to both) → motivation to use **approximate solvers** →
- impact of parameters, their tuning on solvers → motivation of **parameter control** (for on-line solver) →
- but what if we want to solve a class of problems (CoP) → algorithms performance is different →
- user could not determine it [63] → exploration-exploitation balance
- no-free-lunch (NFL) theorem [114] → motivation of the thesis

thesis motivation The most related research field is Hyper-heuristics optimizations [21], that are designed to intelligently choose the right low lewel heuristics (LLH) while solving the problem. But the weak side of hyper-heuristics is the luck of parameter tuning of those LLHs [links]. In the other hand, meta-heuristics often utilize parameter control approaches [links], but they do not select among underlying LLHs. The goal of this thesis is to get the best of both worlds - algorithm selection from the hyper-heuristics and parameter control from the meta-heuristics.

1.2 Research objective

Yevhenii: Rename: Problem definition?

The following steps should be completed in order to reach the desired goal:

Analysis of existing studies of algorithm selection. *(find a problem definition, maybe this will do [63])*

Analysis of existing studies in field of parameter control and algorithm configuration problems *(find a problem definition)* [69]

Formulation and development of combined approach for LLH selection and parameter control.

Evaluation of the developed approach with

Yevhenii: family of problems??? since it is a HH, maybe we should think about it...

.

Research Questions At this point we define a Research Questions (RQ) of the Master thesis.

- **RQ 1** Is it possible to select an algorithm and it hyper-parameters while solving an optimization problem *on-line*?
- **RQ 2** What is the gain of selecting and tuning algorithm while solving an optimization problem?
- **RQ 3?** How to solve the problem of algorithm selection and configuration simultaneously?

1.3 Solution overview

Yevhenii: Rename: Problem solution?

- described problems solved by HH, highlight problems of existing HHs(off-line, solving a set of homogeneous problems in parallel)
- create / find portfolio of MHs (Low level Heuristics)
- define a search space as combination of LLH and their hyper-parameters (highlight as a contribution)
- solve a problem on-line selecting LLH and tuning hyper-parameters on the fly. (highlight as a contribution? need to analyze it.)

Thesis structure The description of this thesis is organized as follows. First, in chapter 2 we refresh readers background knowledge in the field of problem solving and heuristics. In this chapter we also define the scope of thesis. Afterwards, in chapter 2 we describe the related work and existing systems in defined scope. In Chapter 4 one will find the concept description of dynamic heuristics selection. Chapter 5 contains more detailed information about approach implementation and embedding it to BRISE. The evaluation results and analysis could be found in Chapter 6. Finally, Chapter 7 concludes the thesis and Chapter 8 describe the future work.

2 Background and Related Work Analysis

In this Chapter we provide the reader with a review of the basic knowledge in fields of optimization problems and approaches for solving them. A reader, experienced in field of optimization and search problems, may consider this chapter as an obvious discussion of well-known facts. If such notions as a *parameter tuning* and a *parameter control* are not familiar to you or seems the same, we highly encourage you to spend some time reading this chapter carefully. In any case, it is worth for everyone to refresh the knowledge with coarse-grained description of topics, mentioned in this section and examine the examples of hyper-heuristics in Section 2.2.5 and systems for parameter tuning in Section 2.3.2.

The structure of this Chapter is defined as follows. Firstly, we give an informal definition of optimization problem and enumerate possible solver types in Section 2.1. Secondly, we pay attention to the heuristic solvers, their weak points and *No Free Lunch Theorem* in Section 2.2. Afterwards, in Section 2.3 we discuss the influence of parameter setting and possible approaches to set the parameters. Section 2.4, dedicated to *Combined Algorithm Selection and Hyper-parameter Tuning* problem, is followed by conclusion on the literature analysis outlining the thesis' scope in Section 2.5.

2.1 Optimization Problems and their Solvers

Our life is full of different difficult and sometimes contradicting choices. Optimization is an art of making good decisions.

A decision between working hard or going home earlier, to buy cheaper goods or to follow brands, to isolate ourselves or to visit friends during the quarantine, to spend more time for planning trip or to start it instantly. Each decision that we make, has its consequences.

Figure 2.1 outlines the trade-off between a decision quality and an amount of effort spent. The underlying idea of the research in optimization problems solving sphere is to squash this curve simultaneously down and to the left thus, deriving a better result with less cost when solving the optimization problem.

Yevhenii: axes labels should be distinguishable from axes values (Dima review)

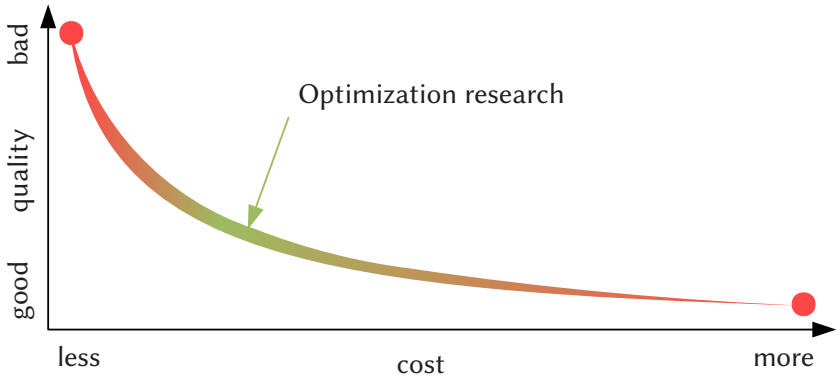


Figure 2.1 Optimization trade-off.

2.1.1 Optimization Problems

While the *search problem* (SP) defines the process of finding a possible solution for the *computation problem*, the *optimization problem* (OP) defined as a special case of the SP, focused on the process of finding the *best possible* solution for computation problem [47].

The focus of this thesis is the optimization problems.

Most studies conducted in this field have tried to formalize the OP concept, but the underlying notion is so vast that it is hard to exclude the application domain from the definition. The description of every possible optimization problem and all approaches to its solving are not in the scope of this thesis, while we consider it necessary to present a coarse-grained review in order to make sure that readers are familiar with all the terms and notions mentioned in the thesis.

To begin with, let us define the optimization *subject*. Analytically, it could be represented as the function $Y = f(X)$ that accepts some input X and reacts to it, providing an output Y. Informally, it could be imagined as the *target system* f (TS), shown in Figure 2.2. It accepts the input information with its *inputs* X_n, which are sometimes called variables or parameters, processes them performing some *task* and produces the result on its *outputs* Y_m.

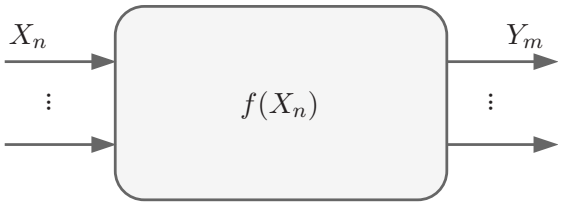


Figure 2.2 Optimization Target System.

Each (unique) pair of sets X_n^i and respective Y_m^i form the $Solution^i$ for computational problem. All possible inputs X^i, where $i = 1...N$ form the *search space* of N size, while all outcomes Y^i, where $i = 1...M$ form an *objective space* of M size.

The solution is characterized by the *objective value(s)* — a quantitative measure of TS performance that we want to minimize or maximize in the optimization problems. We could obtain those value(s) directly, by reading the output on Y_m, or indirectly, for instance, noting the wall clock time TS took to produce the output Y^i for given X^i. The solution objective value(s) form the *object* of optimization. For the sake of simplicity we here use Y_m, *outputs* or *objectives* interchangeably as well as X_n, *variables* or *parameters*.

Yevhenii: figure for classification

Next, let us highlight the target system characteristics. In works [2, 14, 29, 42] dedicated to solving the OPs, the authors distinguished OP characteristics that overlap through each of these works. Among them, we found the following properties to be the most important ones:

- **Input data type** of X_m, which is a crucial characteristic. All input variables could be (1) *discrete*, where representatives are binary strings, integer-ordered, or categorical data, (2) *continuous*, where variables are usually a range of real numbers, or (3) *mixed*, as the mixture of the previous two cases.

- **Constraints**, which describe the relationships among inputs and explain the dependencies in allowable values for them. As an example, imagine that having X_n equal to $value$ implies that X_{n+k} should not appear at all, or could take only some subset of all possible values.

- **Type of target system**, which is an amount of exposed knowledge about the dependencies $X \to Y$ before the optimization process starts. Taking this into consideration, an optimization could be of several types: *white box* — it is possible to derive the algebraic model of TS, *gray box* — the amount of exposed knowledge is significant but not enough to build the algebraic model and *black box* — the exposed knowledge is mostly negligible.
- **Determinism of TS**, which is one of possible challenges, when the output is uncertain. TS is *deterministic*, when it each time provides an equal output for the same input. However, in most real-life challenges engineers tackle *stochastic* systems, the output of which is affected by random processes happened inside TS.
- **Cost of evaluation**, which is an amount of resources (energy, time, money, etc.) TS should spend to produce the output for particular input. It varies from *cheap*, when TS could be an algebraic formula and task evaluation is a simple mathematic computation, to *expensive*, when the TS is a pharmaceutical company, and the task is to perform a whole bunch of tests for a new drug, which may last years.
- **Number of objectives**, which is a size of the output vector Y_m^i. With regard to this, the optimization could be either single- ($m = 1$), or multi- ($m = 2...M$) objective, where the result is one single solution, or a set of non-dominated (Pareto-optimal) solutions.

Most optimization problem types could be obtained by combining different types of each characteristic listed above.

In this thesis we tackle practical combinatorial problems, where the most prominent examples are *bin packing* [77], *job-shop scheduling* [17] or *vehicle routing* [106] optimization problems. All combinatorial problems are *NP-Complete* meaning they are in both *NP* and *NP-Hard* complexity classes[44]. NP complexity implies that the solution is verifiable in the polynomial time, while in the NP-Hard case, the problem can be transformed to other NP-Complete problem in polynomial time, allowing to use a different solving algorithm.

As an example, let us grasp these characteristics for *traveling salesman problem* (TSP) [3] — an instance of the vehicle routing problem [68] and one of the most frequently studied a combinatorial OP (here we consider deterministic and symmetric TSP). The informal definition of TSP is as follows: "Given a set of N cities and the distances between each of them, what is the shortest path that visits each city once and returns to the origin city?" With respect to our previous definition of the optimization problem, the target system here is a function that evaluates the length of proposed path. The TSP distance (or cost) matrix is used in this function for the evaluation and it is clear that this TS exposes all internal knowledge therefore, it is a white box. The input X_n is a vector of city indexes as a result, the type of input data is non-negative integers. There are two constraints for the path: it should contain only unique indexes (visit each city only once) and it should start and end from the same city: $[2 \to 1 \to ... \to 2]$. Since the cost matrix is fixed and not changing during the solving process, the TS is considered to be deterministic and costs of two identical paths are always the same. Nevertheless, there exist Dynamic TSP where the cost matrix changes at runtime to reflect a more realistic real-time traffic updates[25]. It is cheap to compute a cost for a given path using the cost matrix therefore, overall solution evaluation in this OP is cheap, and $n = N!$ is the overall number of solutions. Since we are optimizing only the route distance, this is a single-objective OP.

2.1.2 Optimization Problem Solvers

Most of the optimization problems could be solved by an *exhaustive search* — trying all possible combinations of the input variables and choosing the one, which provides the best objective value. This

approach guarantees finding a globally optimal solution of the OP. But when the search space size significantly increases, the brute-force approach becomes infeasible and in many cases solving even the relatively small problem instances takes too much time.

Here, different optimization techniques come into play. Characteristics exposed by target system could restrict and sometimes strictly define the applicable approach. For instance, imagine you have a white-box deterministic TS with a discrete constrained input data and a cheap evaluation. The OP in this case could be solved using the *Integer Linear Programming* (ILP), or a heuristic approaches. But if this TS turned out to be a black-box, the ILP approaches will not be applicable anymore and one should consider using the heuristics [14].

Evidently, there exist a lot of different facets for optimization problem solvers classification, but they are a subject of many surveying works [14, 39, 58]. In this thesis, as the point of interest we highlight only two of them.

- **Solution quality** perspective:
  1. **Exact** solvers are those algorithms that always provide an optimal OP solution.
  2. **Approximate** solvers produce a sub-optimal output with guarantee in quality (some order of distance to the optimal solution).
  3. **Heuristics** solvers do not give any worst-case guarantee for the final result quality.
- **Solution availability** perspective:
  1. **Completion** algorithms report the results only at the end of their run.
  2. **Anytime** algorithms are designed for stepwise solution improvement thus, could expose intermediate results.

Each of these algorithm characteristics provide their own advantages, having, however, their own disadvantages. For instance, if solution is not available at any time, one will not be able to control the optimization process. On the contrary, if it is available, the overall performance may decrease. If the latter features are more or less self-explanatory, the former require more detailed explanation.

Solution Quality

Exact Solvers. As was stated above, the exact algorithms are those, which always solve OP to guaranteed optimality. For some OP it is possible to develop an effective algorithm that is much faster than the exhaustive search — they run in a super-polynomial time, instead of exponential, still providing an optimal solution. As authors claimed in [113], if the common belief $P \neq NP$ is true, the super-polynomial time algorithms are the best we can hope to get when dealing with the NP-complete combinatorial problems.

According to the definition in [43], the objective of an exact algorithm is to perform much better (in terms of running time) than the exhaustive search. In both works [43, 113] the authors enumerated main techniques for designing the exact algorithms. Each of these techniques contributes in this 'better' independently and later they could be combined.

You may find a brief explanation of them below:

- **Branching and bounding** techniques, when applied to the original problem, split the search space of all possible solutions (e.g. exhaustive enumeration) to a set of smaller sub-spaces. More formally, this process is called *branching the search tree into sub-trees*. This is done with an intent to prove that some of sub-spaces never lead to an optimal solution and thus could be rejected.

- **Dynamic programming across sub-sets** technique could be combined with the branching techniques. After forming the sub-trees, the dynamic programming attempts to derive the solutions for the smaller subsets and later combine them into the solutions for the lager subsets. This process repeats until the solution for original search space obtained.

- **Problem preprocessing** could be applied as an initial phase of the solving process. This technique is dependable upon the underlying OP, but when applied properly, it significantly reduces the running time. A simple example from [113] elegantly illustrates this technique: imagine a problem of finding a pair of two integers x_i and y_i in X_k and Y_k sets of unique numbers (k here denotes the size of sets), that sum up to an integer S. The exhaustive search approach implies enumerating all $x - y$ pairs. The time complexity in this case is $O(k^2)$. But if we firstly consider the data preprocessing by sorting and afterwards, using the bisection search repeatedly in these sorted arrays to find k values $S - y_i$, then the overall time complexity reduces to $O(k \log(k))$.

Approximate Solvers. When the OP cannot be solved to optimal in polynomial time, the only solution is to start thinking of the alternative ways to tackle it. A common decision is to apply the requirement *relaxation techniques* [93] to derive the approximated solution. Approximate algorithms are representatives of the theoretical computer science. They were created in order to tackle the computationally difficult (not solvable in super-polynomial time) white-box OP. Words of Garey and Johnson (computer scientists, authors of *Computers and Intractability* book [44]) could pay a perfect description of such approaches: "I can't find an efficient algorithm, but neither can all of these famous people."

Unlike exact, approximate algorithms relax the quality requirements and solve the OP effectively with the provable assurances on the result distance from an optimal solution [112]. The worst-case results quality guarantee is crucial in the approximation algorithms design and involves the mathematical proofs.

How do these algorithms guarantee on quality, if the optimal solution is unknown beforehand? — is a reasonable question arises at this point. Certainly, it sounds contradicting, but the comprehensive answer to this question requires an explanation of the key approximation algorithms design techniques that is not in the scope of this thesis. Nevertheless, let us briefly describe these techniques.

In [112] the authors provided several techniques of the approximate solvers design. For instance, the *Linear Programming* (LP) relaxation plays a central role in approximate solvers. It is well known, that solving the ILP is *NP-hard* problem. However, it could be relaxed to the polynomial-time solvable linear programming. Later, a fractional solution for the LP will be rounded to obtain a feasible solution for the ILP. Different rounding strategies define separate approximate solver techniques [112]:

- **Deterministic rounding** follows a predefined strategy.

- **Randomized rounding** performs a round-up of each fractional solution value to the integer uniformly.

In contrast to rounding, another technique requires building a *Dual Linear Program* (DLP) for given linear program. This approach utilizes the *weak* and *strong duality* properties of DLP to derive the distance of the LP solution to the original ILP optimal solution. Other properties of DLP form a basis for the *Primal-dual* algorithms. They start with a dual feasible solution and use the dual information to derive the primal linear program solution (possibly infeasible). If the primal solution is not feasible, the algorithm modifies the dual solution increasing the dual objective function values. In any case, these approaches are far beyond the thesis scope, but in case of an interest reader could start his own investigation from [112].

Heuristics. As opposed to the solvers mentioned above, heuristics do not provide any guarantee on the solution quality. They are applicable not only to the white-box TS but also to the black-box cases. These approaches are sufficient to quickly reach an immediate, short-term goal in such cases, when the finding an optimal solution is impossible or impractical because of the huge search space size.

As in the reviewed above approaches, here exist many facets for classification. We start from the largest one, namely the *level of generality*:

- **Simple heuristics** are the specifically designed to tackle the concrete problem algorithms. They fully rely on the domain knowledge, obtained from the optimization problem. Simple heuristics do not provide any mechanisms to escape a local optimum therefore, could be easily trapped to it [85].
- **Meta-heuristics** are the high-level heuristics, that being domain knowledge-dependent, also provide some level of generality to control the search. They could be applied to broader range of the OPs. They are often nature-inspired and comprise mechanisms to escape the local optima but may converge slower than the simple heuristics. For the more detailed explanation we refer to survey [12].
- **Hybrid-heuristics** arise as the combinations of two or more meta-heuristics. They could be imagined as the recipes merge from the cook book, combining the best decisions to create something new and presumably better.
- **Hyper-heuristics** are the algorithms that operate in the search space of *Low Level Heuristics* (LLH). Instead of tackling the original problem, they choose (or construct) LLHs, which will tackle this problem for them [21].

In the upcoming Section 2.2, dedicated to heuristics, we provide more detailed information on each of the approaches mentioned above.

The Most Suitable Solver Type

"Fast, Cheap or Good? Choose two."

The old engineering slogan.

At this point, we have reached the crossroads and should make a decision, which way to follow.

Firstly, we have the exact solvers for the optimization problems. As mentioned above, they always guarantee to derive an optimal solution. Today, tomorrow, maybe in the next century, but eventually the exact solver will find it. The only thing we need is to construct the exact algorithm. This approach definitely offers the best final solution quality however, it sacrifices the solver construction simplicity and the speed in problem solving.

Secondly, we have the approximate solvers. They do not guarantee finding the one and only optimal solution but suggest a provably good instead. From our perspective, the required effort for constructing the algorithm and proving its preciseness remains the same as for the exact solvers. However, this approach outperforms the previous one in the speed of problem solving, sacrificing a reasonably small amount of the result quality. It sounds like a good deal.

Finally, the remaining heuristic approaches. They quickly produce a solution, in comparison to the previous two. In addition, they are much easier to apply for the specific problem — there is no need to build a complex mathematical models or prove the theorems. However, the biggest flaw in these approaches is an absence of the solution quality guarantee.

As we mentioned in Section 2.1.1, this thesis is dedicated to facing the practical combinatorial problems, such as the TSP. They are NP-complete, that is why we are not allowed to apply the exact solvers. In both approximate and heuristic solvers we are sacrificing the solution quality, though in different quantities. Nevertheless, the heuristic algorithms repay in the development time and provide the first results faster. The modern world is highly dynamic, in the business survive those, who are faster and stronger. In the most cases, former plays the crucial role for success. The great products are built iteratively, enhancing existing results step-by-step and leaving the unlucky decisions behind. It motivates us stick to the heuristic approach within the scope of the thesis.

In the following Section 2.2 we shortly survey different heuristic types and examples. We analyze their properties, weaknesses and ways to deal with them. As the result, we select the best suited class of heuristics for solving the TSP problem.

2.2 Heuristic Solvers for Optimization Problems

We base our descriptions of heuristics and their examples on the mentioned in Section 2.1.1 traveling salesman problem. The input data X to our heuristics will be the problem description in form of a distance matrix (or coordinates to build this matrix), while as an output Y from heuristics we expect to obtain the sequence of cities, depicting the route plan.

Most heuristic approaches utilize the following concepts:

- **Neighborhood**, which defines a set of solutions that could be derived performing a single step of the heuristic search.
- **Iteration**, which could be defined as an action (or a set of actions) performed over the solution in order to derive a new, hopefully better one.
- **Exploration** (diversification), which is the process of discovering previously unvisited and presumably high quality parts of the search space.
- **Exploitation** (intensification), which is the usage of already accumulated knowledge (solutions) to derive a new solution but similar to existing one.

2.2.1 Simple Heuristics

As we mentioned above, the simple heuristics are domain-dependent algorithms, designed to solve a particular problem. They could be defined as the rules of thumb, or strategies to utilize the information, exposed by the TS and obtained from the previously found solutions, to control the problem-solving process [85].

Scientists draw the inspiration for heuristics creation from all aspects of our being: starting from the observations of how humans tackle daily problems using intuition, and proceeding to the mechanisms discovered in nature. The two main types of simple heuristics were outlined in [22]: *constructive* and *perturbative*.

The first type aggregates the heuristics which construct the solutions from its parts step by step. A prominent example of constructive approach is a *greedy algorithm*, which can also be called the *best improvement local search*. When applied to TSP, it tackles the path construction simply accepting the next closest city from currently discovered one. Generally, the greedy algorithm follows the logic of making a sequence of locally optimal decisions therefore, it ends up in a local optimum after constructing the very first solution.

The second type, called a *local search*, implies heuristics which operate on the complete solutions, perturbing them. A simple example of the local search is a *hill climbing algorithm*, also known as a *first improvement local search* [111]. This heuristic accepts a better solution as soon as it finds it, during the neighborhood evaluation. This approach plays a central role in many high-order algorithms however, it could be very inefficient, since in some cases the neighborhood could be enormously huge.

Indeed, since the optimization result is fully dependent from the starting point. The use of simple local search heuristics might not lead to a globally optimal solution. Nevertheless, in this case the advantage will be the implementation simplicity [112].

2.2.2 Meta-Heuristics

Meta-Heuristic (MH) is an algorithm, created to solve a wider range of complex optimization problems with no need to deeply adapt it to each problem.

The research in MHs field arose even before 1940s, when the MHs were already actively applied. However, there were no all-embracing and complex studies of MHs at that time. The first formal studies appeared between 1940s and 1980s. Deep and profound research in this field reaches its most active stage in the late 1990s, when the numerous MHs popular nowadays were invented. The period from 2000 and up till now the authors in [100] call the framework growth time, when the meta-heuristics widely appear in form of frameworks, providing a reusable core and requiring only the domain-specific adaptation.

The prefix *meta-* indicates the algorithms to be of the *higher level* when compared to simple problem-dependent heuristics. A typical meta-heuristic structure could be imagined as *n-T—H* (template and hook) framework variation pattern.

Yevhenii: do not have a cite since it is more my observation + guess, better to remove?

The template part is stable and problem independent, it forms the core of an algorithm and usually exposes *hyper-parameters*, which could be used for the algorithm tuning. The hook parts are domain-dependent and should be adapted for problem at hand. Later, T operates on the set of Hs to perform an optimization. Many MHs contain stochastic components, which provide abilities to escape from local optimum. However, it also means that the output of meta-heuristic is non-deterministic and it could not guarantee the result preciseness [18].

The meta-heuristic optimizer success on a given OP depending on the *exploration vs exploitation balance*. If there is a strong bias towards diversification, the solving process could naturally skip a good solution while performing huge steps over the search space, but in case of intensification domination, the process will quickly settle in the local optima. The disadvantage of the simple heuristic approaches mentioned above is a high exploitation dominance, since they simply do not have the components contributing to exploration. In the most cases, it is possible to decompose MH into simple components and clarify, to which of competing processes contributes each component. Often, the simple heuristics are used as the intensification component.

In general, the difference between existing meta-heuristics lays in a particular way how they are trying to achieve this balance, but the common characteristic is that the most of them are inspired by real-world processes — physics [110], biology [96], ethology [31, 95, 103], and even evolution [11, 35].

Meta-Heuristics Classification

When the creation of novel methodologies has slowed down, the research community began to organize and classify the created algorithms.

Yevhenii: Im not sure about wiki-based pic, but cannot find anything else more or less attractive..

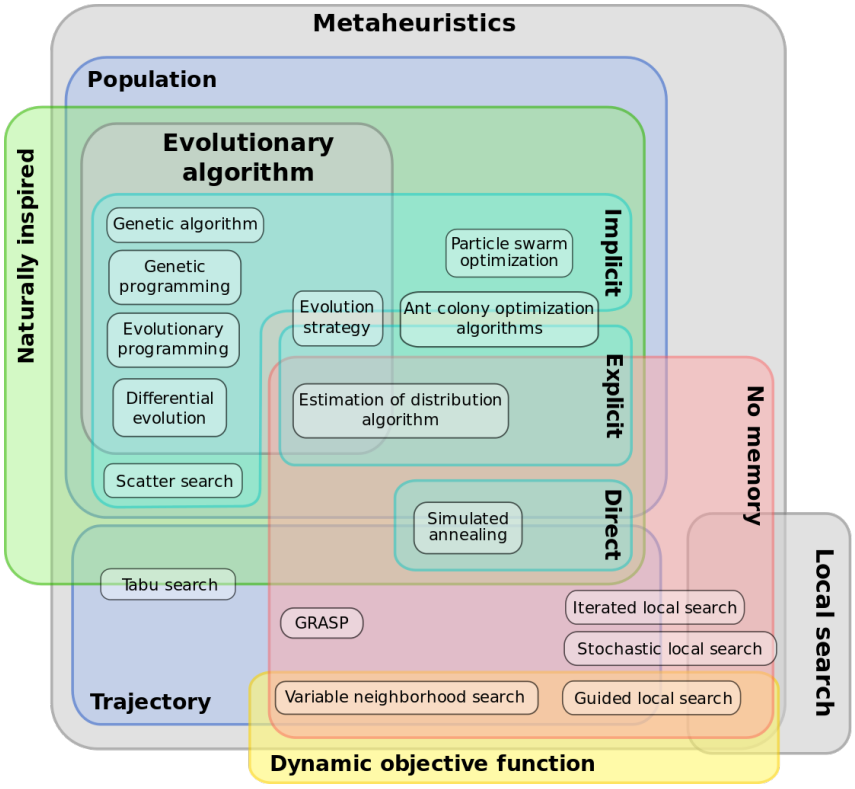


Figure 2.3 Meta-heuristics Classification [26].

As an example, [15] highlights the following classification facets:

- The **walk-through search space method** could be either trajectory-based or discontinuous. The first one corresponds to a closed walk through the neighborhood where such prominent examples as *iterated local search* [76] or *tabu search* [46] do exist. The second one allows large jumps in the search space, where the examples are such MHs as *variable neighborhood search* [49] or *simulated annealing* [64].
- The **number of concurrent solutions** could be either single or multiple. Such approaches as tabu search, simulated annealing or iterated local search are the examples of algorithms with a single concurrent solution. Evolutionary algorithms [35], ant colony optimization [31] or particle swarm optimization [62] are the instances of algorithms with multiple concurrent solutions (the population of solutions).
- From the **memory usage** perspective, we distinguish those approaches which do and do not utilize the memory. Tabu search explicitly uses memory in form of tabu lists to guide the search, but simulated annealing is memory-less.
- The **neighborhood structure** could be either static or dynamic. Most local search algorithms, such as simulated annealing and tabu search are based on a static neighborhood. Variable neighborhood search is an opposite case, where various structures of neighborhood are defined and interchanged while the algorithm solves the OP.

There are many more classification facets, with are not in the scope of this thesis. Figure 2.3 illustrates the summarized classification including some characteristics and well-known meta-heuristic instances

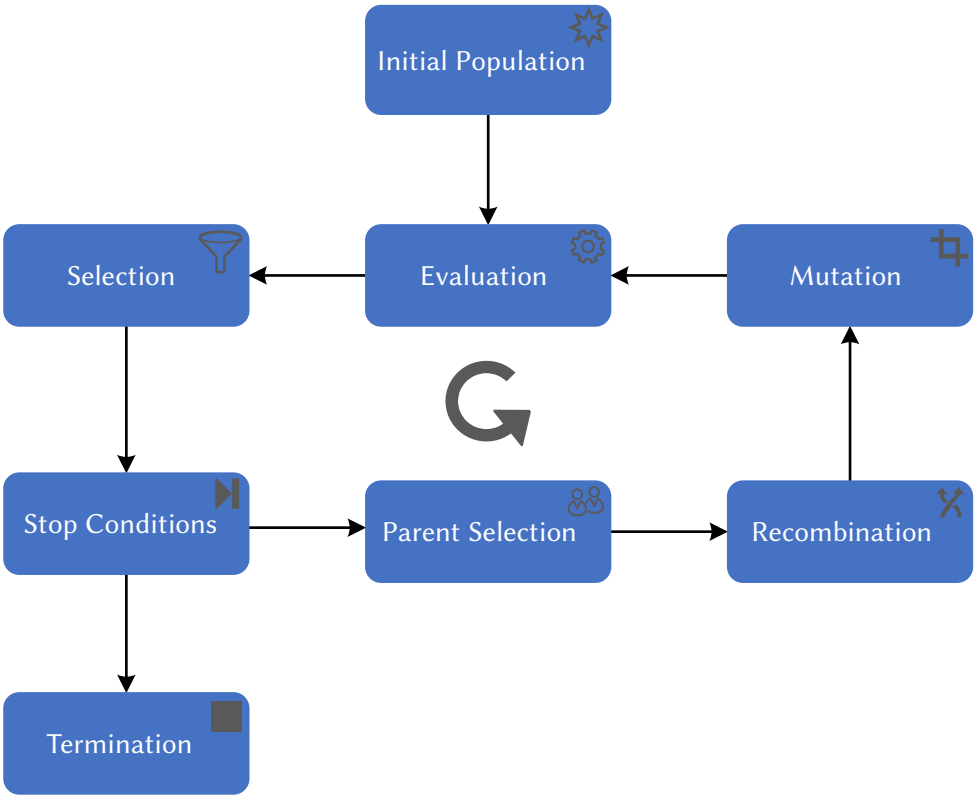


Figure 2.4 Evolutionary Algorithms Workflow.

we did not mention.

Meta-Heuristics Examples

At this point, let us briefly describe some of the most prominent and widely used meta-heuristics. It is motivated by the later usage of them in our approach, described in Section 4.4.

Evolutionary Algorithms (EAs) are directly inspired by the processes in nature, described in evolution theory. The common underlying idea in all of these methods is as follows: if we put a population of individuals (solutions) into an environment with limited resources (population size limit), a competition processes cause natural selection, where only the best individuals survive [35].

Three basic actions are defined as operators of EAs: a *recombination* operator selects the parent solutions, which later will be combined to produce the new ones (offspring); a *mutation* operator, when applied to solution, creates a new and very similar one. Applying both operators,the algorithm creates a set of new solutions — the offspring, whose quality is then evaluated with the TS. After that, a *selection* operator is applied to all available solutions (parents and offspring) to keep the population size within the defined boundaries. This process is repeated, until some termination criteria is fulfilled. For instance, the maximal iterations counter was reached, the number of TS evaluations exceeds the defined maximal value, or the solution with the required quality is found. The high-level work-flow of EA is depicted in Figure 2.4.

The well-known examples of EAs include the *genetic algorithm* [96], *genetic/evolutionary programming* [67], *evolution strategies* [11], and many other algorithms.

Genetic Algorithm (GA) is the first of all associated with the Evolutionary Algorithms. GA traditionally has a fixed workflow: given an initial population of μ usually randomly sampled individuals, the parent selection operator creates pairs of parents, where the probability of each solution to become a parent depends on its objective value (fitness, or results). After that, the crossover operator is applied to every created pair with a probability p_c and produces children. Then, newly created solutions undergo the mutation operator with an independent probability p_m. The resulting offspring perform a tournament within the selection operator and μ survivals replace the current population [34]. Distinguishable characteristic of vanilla GA is the usage of the following operators: bit-string solution representation, one-point crossover recombination, bit-flip mutation and generational selection (only children survive).

Evolution Strategy (ES), comparing to GA, is working in a vector space of the solution representation. However, they also use the population size of μ individuals and λ offspring generated in each iteration. While the general workflow for all EAs remains the same, they mostly differ in underlying operators. In ES, the parent selection operator takes a whole population into consideration uniformly, the recombination scheme could involve more than two parents to create one child. To construct a child, the recombination operator joins parents alleles in two possible ways: (1) with uniform probability for each parent (discrete recombination), or (2) averaging the weights of alleles by parent solution quality (intermediate recombination). There are two selection schemes, used in such algorithms. (μ, λ): discard all parents and select only among offspring highly enriching the exploration, and $(\mu + \lambda)$: include also the predecessor solutions into selection, which is often called the *elitist selection* [34]. In many cases, the ES utilizes a very useful feature of *self-adaptation*: changing the mutation step sizes at runtime, which we will discuss in Section 2.3.3.

Simulated Annealing (SA). This is the other type of meta-heuristics, inspired by the technique used in metallurgy to obtain 'well-ordered' solid state of metal [110]. An annealing technique imposes a globally minimal internal energy state and avoids locally minimal semi-stable structures.

The SA treats the search process as a metal with a high temperature at the beginning and lowering it to minimum while approaching the end. It starts with an initial solution S creation (randomly or using some other heuristic) and temperature parameter T initialization. At each iteration, a new solution candidate is sampled within a neighborhood of the current solution: $S^* \leftarrow N(S)$. The newly sampled solution replaces the older one, if (1) optimization objective $f(S^*)$ dominates over $f(S)$ or (2) with a probability that depends on a quality loss and current value of T, see Equation (2.1).

$$p(T, f(S^*), f(S)) = \exp(-\frac{|f(S^*) - f(S)|}{T}) \tag{2.1}$$

At each iteration the temperature parameter T value is decreased following some type of annealing schedule, which is also called as *cooling rate* [18]. The weak side here is that the quality of each annealing schedule is the problem dependent and cannot be determined beforehand. Nevertheless, the SA algorithms with adaptive parameters do exist and address this problem changing the cooling rate or temperature parameter T during the search process. Later, we will shortly review these techniques in Section 2.3.3.

2.2.3 Hybrid-Heuristics

The hybridization of different systems often provides a positive effect — taking the advantages of one system and merging them with characteristics of the other, getting the best from the both systems. The same idea is applicable in case of meta-heuristics. Imagine you have two algorithms, one is biased towards exploration, the other — towards exploitation. Applying them separately, the expecting results in most cases may be far away from the optimal as the outcome of disrupted diversification-intensification

balance. However, when merging them into, for example, repeated stages of hybrid heuristic, one will obtain the advantages of both escaping a local optima and finding a good quality result.

Most of available hybridization algorithms are created with the help of this idea of two heuristics staging combination, one of which is suited for the exploration and other is better for the exploitation.

The methods to construct the hybrids are mostly defined by the underlying heuristics. Therefore, to the best of our knowledge they could not be generalized and classified in an appropriate way. The only one commonly shared characteristic is the usage of *staging approach*, where the output of one algorithm is used as the initial state of the other.

As during the simple heuristics review, here we also introduce examples of performed hybridization in order to provide the reader a better understanding how can be combined different algorithms components and what is the effect on the aforementioned balance.

Hybrid-Heuristics Examples

Guided Local Search and Fast Local Search. The main focus of guided local search (GLS) in this case, lies on the search space exploration and the guidance of process using incubated information. To some extent, the GLS approach is closely related to the frequency-based memory usage in tabu search. During the runtime, GLS modifies the problem cost function to include penalties and passes this modified cost function to the local search procedure. These penalties form a memory that characterizes a local optimum and guide the process out of it. The more time algorithm spends in the local optimum, the higher penalties. A local search procedure is carried out by fast local search (FLS) algorithm, where the main advantage is a quick neighborhood traversal. It is done by breaking it up into a number of small sub-neighborhoods. Afterwards, by performing the depth-first search over these sub-neighborhoods, it ignores those, without an improving moves. At some point of time FLS reaches the local optimum and passes back the control in GLS to update the penalties and repeat the iteration [107].

Direct Global and Local Search. This hybridization consists of two stages: the stochastic global coarse pre-optimization and the deterministic local fine-optimization. In the first stage, the authors apply one of the two mentioned earlier meta-heuristics: genetic algorithm or simulated annealing [51]. A transition from global to local search happens after reaching the predefined conditions. For instance, when the number of TS evaluations exceeds a boundary, or when no distinguishable improvement was made in the last few iterations. Then, the pattern search algorithm also known as the direct, derivative-free, or black-box search performs the fine-optimization. The hybrid-heuristic terminates when the pattern search converges to the local optima [102].

Simulated Annealing and Local Search. After the brief explanation of previous two hybrids, an observant reader hopefully guesses, how the next hybridization works. The authors in their work [78] called this method 'Chained Local Optimization'. Thus, it is a yet another representative of staged hybridization. Iteration starts with the current solution perturbation, called *kick* in [78], referring a dramatic change of a current position within the search space. Afterwards, the hill climbing algorithm is applied to intensify the obtained solution. When reached the local optimum, hill climber passes the control flow back to the simulated annealing for acceptance criteria evaluation, which finishes the iteration.

EMILI. Easily Modifiable Iterated Local search Implementation (EMILI) is a framework system for the automatic generation of new hybrid stochastic local search algorithms [84]. EMILI is a solver for permutation flow-shop problems (PFSP), also known as flow shop scheduling problems [91]. In PFSP

the search of an optimal sequence of steps to create products within a workshop is performed. In this framework, the authors have implemented both generic algorithmic- and problem-specific building blocks. They also have defined grammar-based rules for those blocks composition and used an automatic parameter tuning tool IRACE [74] to find the high performing algorithm configurations. The workflow of EMILI could be split in three steps: (1) adaptation of the grammar rules to specific PFSP objective representations (makespan, sum completion times and total tardiness), (2) generation of all possible hybrid heuristics for each PFSP representation and (3) execution of IRACE to select the best performing hybrid for each problem.

From our perspective, the described approach of automatic algorithm generation is an example of construction hyper-heuristics, which we describe in the upcoming Section 2.2.5. However, we are not authorized to change the system class (from hybrid- to hyper-heuristic) defined by the EMILI authors.

2.2.4 No Free Lunch Theorem

A nature question could arise "If we have all this fancy and well-performing heuristics, why should we put an effort in developing new algorithms, instead of using the existing?" And the answer to this question is quite simple — the perfect algorithm suited for all OP does not exist and can not exist. The empirical research has shown that some meta-heuristics perform better with some types of problems but poorly with others. In addition to that, for different instances of the same problem type, the same algorithm could result in unexpected performance metrics. Moreover, even in different stages of same problem solving process the dominance of one heuristic over another could change.

All search algorithms perform exactly the same, when the results are averaged over all possible optimization problems. If an algorithm is gaining the performance in one problems class, it loses in another class. This is a consequence of a so-called *No Free Lunch Theorem for Optimization* (NFLT) [114].

In fact, one could not predict, how exactly one or another algorithm will behave with problem at hand. A possible and the most obvious way is to probe one algorithm and compare its performance to another one during the problem solving process. In this case simple heuristics and meta-heuristics are out of the competition, since once you solved the Optimization Problem you probably wouldn't optimize a second time. Here, the *Hyper-Heuristics* come into play to intelligently pick heuristics that are suitable to problem at hand. We will proceed with their description and how they deal with the NFLT consequences in the following section.

2.2.5 Hyper-Heuristics

A lot of state-of-the-art heuristics and meta-heuristics are developed in a complex and very domain-dependent way, which causes problems in its reuse. It is motivated the research community to raise the level of generality at which the optimization systems can operate and still provide good quality solutions for various optimization problems.

The term **Hyper-Heuristic** (HH) was defined to describe an approach of using some *High-Level-Heuristics* (HLH) to select over other *Low-Level-Heuristics* (LLH) and apply them to solve the *class* of optimization problems rather than a particular instance. Indeed, scientists report that the combination of different HLH produces better results than if they were applied separately [32]. This behavior can be explained by the way of how the search process evolves in time. When you apply a heuristic, it sooner or later converges to some extreme point, hopefully global optimum. But it is 'blind' to other, not visited regions of the search space. Changing the trajectory of investigation by (1) drastically varying the neighborhood, (2) changing the strategy of neighborhood exploration and exploitation could (1) bring you to those previously unreachable zones (2) in more rapid ways. However, usually it is hard

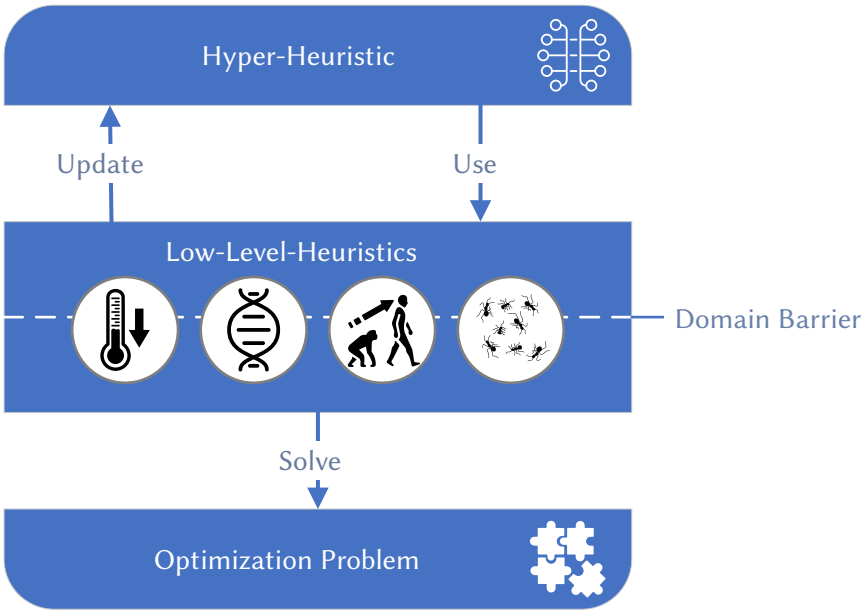


Figure 2.5 Hyper-Heuristics.

to predict how one LLH will behave in every stage of the search process in comparison to another. In hyper-heuristics, this job was encapsulated into the HLH and performed automatically.

In [81] the authors made an infer that HHs can be viewed as a form of reinforcement learning (RL), which is a logical conclusion, especially if we rephrase it to *hyper-heuristics utilize reinforcement learning methodologies*.

The new concept which was implicitly used in meta-heuristics but explicitly pointed out in hyper-heuristics is the *domain barrier* (see Figure 2.5[1]). As we told previously, HH do not tackle the OP directly but use LLH instead. This means, that usually HH are minimally aware of the domain details, such as what are data types, relationships, etc. within a domain. This information is rather encapsulated in LLHs therefore, HHs could be used to a broader range of optimization problems.

With this idea, many researchers started to create not only hyper-heuristics to tackle a concrete optimization problem class, but also frameworks with building blocks for their creation.

Classification

Although, the research in hyper-heuristics field is actively ongoing, many algorithm instances were already created and some trials to organize approaches were conducted in [22, 32, 94].

Researchers in their surveys classify HHs by different characteristics, some of which overlap, but it also happens that important (from our perspective) features were not highlighted in all works.

In this section we present a union of those important hyper-heuristics classification facets to better justify the goal of this thesis.

We begin with the two broadest classes, which differentiate HH *routine*, also called as *a nature of high-level-heuristic search space* [22, 23, 32]. The first class are hyper-heuristics to *select* low-level-heuristic, in other words *selection hyper-heuristic*. In our discussions above, dedicated to the HHs we were referencing to this concrete type. These algorithms operate in the defined by complete and rather simple low-level-heuristics search space. The task of HLH here is to pick the best suited LLH (or sequence

[1] Icons from `https://thenounproject.com/`

of LLHs) based on an available prior knowledge and apply it to the OP underway. Note, that staging HHs could be viewed as solutions of selection HHs. Hyper-heuristics of the second class seek to *construct* LLH, following some predefined receipt and using the atomic components of other heuristics as Lego bricks. The other commonly used name here is a *construction hyper-heuristic*. These approaches often lead to a creation of new and unforeseen heuristics that are expected to reveal good performance while solving the problem at hand.

Next, the distinction in *nature of LLH search space* arises. In other words: "How does the LLH derive a solution for the OP?" The authors in [22, 23, 32] distinguished *construction* LLHs where the solution creation happens each time from scratch and *perturbation* LLHs where the new solutions are created from parts of already existing ones.

The other broadly used characteristic is the *memory usage* method. From this perspective we distinguish hyper-heuristics in which the learning happens *on-line*, *off-line* or learning mechanisms are *not present* at all [22, 94]:

- In **on-line** case, the HH derives an information, used to select among LLH, while those LLH are solving the problem.
- In **off-line** case, the learning happens before solving a concrete OP. Here one should first train the HH solving other homogeneous problem instances by underlying LLHs (off-line learning phase). After that, the HLH will be able to properly choose among LLHs therefore, be applicable to problems at hand (on-line use phase). Note, that this approach also requires a creation of meta-features extraction mechanism and its application to every optimization problem.
- There also exist **mixed** cases, where the learning happens firstly in the off-line and later also in the on-line phase. Definitely, it is a promising (in terms of results quality) research direction, despite its high complexity.
- In the last case **no learning** mechanisms are present. Therefore, HLH here performs to some extent a random search over LLH search space. At the first sight it may look like a weak approach however, there exist meta-heuristics, which are similar to HH and perform well (variable neighborhood search).

For a more detailed analysis, description, other classification facets and respective hyper-heuristic examples we encourage the reader to look into recent classification and surveying researches [21, 22, 32, 94].

Hyper-Heuristics Instance Examples

Hyper-heuristic for integration and test order problem (HITO). HITO [48] is an example of construction HH. LLHs in this case are presented as a composition of basic EAs operators — crossover and mutation forming multi-objective evolutionary algorithms (MOEA). HH selects those components from $jMetal$ framework [33] using interchangeably *choice function* (form of weighted linear equation) and multi-armed-bandit-based heuristic to balance an exploitation of good components and an exploration of new promising ones.

Markov Chain Hyper-Heuristic (MCHH). MCHH [79] is an on-line selection hyper-heuristic for multi-objective continuous problems. It utilizes reinforcement learning techniques and Markov chain approximations to provide an adaptive heuristic selection method. While solving the OP, MCHH updates the prior knowledge about the probability of producing Pareto dominating solutions by each underlying

LLH using Markov chains, guiding an LLH selection process. Applying on-line reinforcement learning techniques, this HH adapts transition of weights in the Markov chains constructed from all available LLHs, updating prior knowledge for LLH selection.

Hyper-Heuristics Frameworks Examples

Hyper-Heuristics Flexible Framework (HyFlex). HyFlex [82] is a software skeleton, built specifically to help other researchers creating hyper-heuristics. It provides the implementation of components for 6 problem domains: boolean satisfiability, bin packing, personnel scheduling, permutation flow shop and vehicle routing problems. Thereby, problem and solution descriptions, evaluation functions and adaptations for set of low-level-heuristics are provided out-of-the-box. The set benchmarks and comparison techniques to other built HHs on top of HyFlex are included into framework as well.

The intent of HyFlex creators was to provide low-level features that enable the users to focus directly on HLHs implementation without a need to challenge other minor details. It also brings a clear comparison among created HLH performance, since the other parts are mostly common.

From the classification perspective, all derivatives from the HyFlex framework are selection hyper-heuristics however, they utilize different learning approaches. Algorithms, built on top of HyFlex framework could be found in many reviews [32, 80, 94] or on the CHeSC 2011 challenge website[1] (CHeSC is dedicated to choosing the best HH built on top of HyFlex).

Along with HyFlex, a number of hyper-heuristic-dedicated frameworks is growing, some of them are under active development while others are abandoned:

- **Hyperion** [101] is a construction hyper-heuristic framework, aiming to extract an information from the OP search domain for identification of promising components in form of object-oriented analysis.
- **hMod** [109] framework allows not only to rapidly prototype an algorithm using provided components, but also to construct those components using predefined abstractions (such as *IterativeHeuristic*). In current development stage, developers of *hMod* are focusing on creation of development mechanisms rather than providing a set of pre-built heuristics.
- **EvoHyp** [87] framework focuses on hyper-heuristics, created from evolutionary algorithms and their components. Here, the authors enable framework users to construct both selection and generation HHs for both construction and perturbation LLHs types.

2.2.6 Conclusion on Heuristic Solvers

To conclude our review on heuristic approaches for solving optimization problems, we shortly refresh each heuristic level.

On the basis remain simple heuristics with all their domain-specific knowledge usage and particular tricks for solving problems. Usually, they are created to tackle a concrete problem instance, applying simple algorithmic approach. The simplicity of development and usually fast runtime result in a medium results quality.

On the next level inhabit meta-heuristics. They could be compared with more sophisticated solutions hunters which could not only charge directly but also take a step back when stuck in a dead end. This additional skill enables them to survive in new and complex environments (optimization

[1] Cross Domain Heuristic Search Challenge website: asap.cs.nott.ac.uk/external/chesc2011/

problems). However, some adaptations to understand a concrete problem and parameter tuning for better performance still should be performed.

Among with MHs exist hybrid-heuristics. There is nothing special here, they just took some survival abilities from several meta-heuristics hoping to outperform and still requiring adaptation and tuning. In some cases this hybridization provides an advantage, but as the time shows, they did not force MHs out. Thus, we can conclude that the provided balance between development effort and exposed results quality not always assure users to use them.

Finally, those who lead the others, hyper-heuristics are on the upper generality level. Operating by the other heuristics, HHs analyze how good the former are and make a use of this knowledge by solving a concrete problem using those best suited heuristics. Imposing such great abilities, hyper-heuristics tackle not only the concrete optimization problem but an entire class of problems.

2.3 Setting Algorithm Parameters

Most of the existing learning algorithms expose some parameter set, needed to be assigned before using this algorithm. Modifying these parameters, one could change the system behavior and a possible result quality.

When we are talking about the problem of settings the best parameters, following terms should be outlined explicitly:

1. **Target System (TS)** is a subject whose parameters are undergoing changes. Simply, it could be a heuristic, machine learning algorithm or other system.

2. **Parameter** is one of setting hooks, exposed by TS. It should be described in terms of its type and possible values.

3. **Configuration** is a unique combination of parameter values, required to run TS.

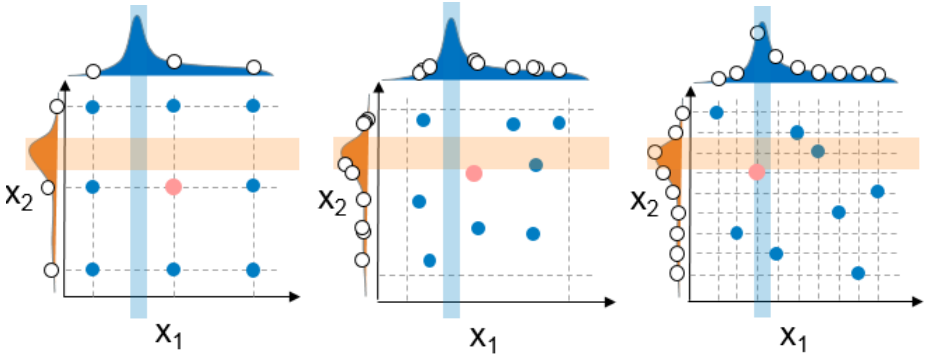
4. **Search Space** is a set of all possible Configurations for defined parameters.

In this thesis we use notions of *parameter* and *hyper-parameter* interchangeably, since the approaches discussed in this section are generally applicable also in machine learning cases. As an example, consider a neuron network. Hyper-parameters in this case specify a structure (number of hidden layers, units, etc.) and define a learning process (rate, regularization parameters values, etc.). Changing them dramatically affect the network's performance and results.

A frequently tackled optimization problem is a **parameter settings problem** (PSP): the process of searching hyper-parameter values that optimize some characteristic of TS. When talking about NN example, PSP could be defined as task of network's accuracy maximization with a given dataset, resulting in a single-objective PSP (SO-PSP). Optimizing a number of TS characteristics simultaneously, such as training time and prediction accuracy, one transforms the SO-PSP into a multi-objective PSP (MO-PSP).

The same applies to heuristics. A proper assignment of hyper-parameters has a great impact on the exploration-exploitation balance and as a result, on an overall algorithm performance.

Up until now there were formalized many approaches for solving task of settings hyper-parameters. One of the simplest ways is just trusting your (or someones else) intuition and using the parameters that seem more or less logical for a particular system and a problem instance. People quickly abandoned this error-prone technique in favor of automatic approaches. It was also motivated by an increasing computational capacities, which gave an opportunity to evaluate more configurations in less time. These automatic methods could be split into two technique families: off-line *parameter tuning* and on-line *parameter control*.



a) Grid. **b)** Random. **c)** Learning.

Figure 2.6 Automated Parameter Tuning Approaches.

2.3.1 Parameter Tuning

Roughly speaking, the off-line approach is a process of traversing the search space of hyper-parameters and evaluating TS with these parameters on some set of toy problems. At the end of this process, the best found HPs are reported and later used to solve a new, unforeseen problem instance.

In this part of thesis we outline existing automated approaches for parameter tuning, illustrating them in Figure 2.6[1]. In this example, the TS exposes two parameters: X_1 and X_2. Each figure shows dependencies between X_1 (horizontal axis) and X_2 (vertical axis) values and the subject of optimization along those axes (here the maximization case is depicted). The best configurations, found by each approach are highlighted in pink.

Grid search parameter tuning. It is a rather simple approach for searching parameters. Here the original search problem is relaxed and later solved by a brute-force algorithm. The set of all possible configurations (parameter sets) for relaxed problem is derived by specifying a finite number of possible values for each hyper-parameter under consideration. After evaluating all configurations on TS, the best found solution is reported. As you can see, this approach could skip promising parts of search space (Figure 2.6a). Moreover, the required time to probe all possible combinations is increasing by means of a factorial complexity.

Random search parameter tuning. This methodology relies on a random (often uniform) sampling of hyper-parameters and their evaluation on each iteration. At the first sight, it might look unreliable to chaotically traverse the search space. But empirical studies show that with a growing number of evaluations this technique starts to outperform grid search [9]. As an example, let us draw your attention to the best configurations (highlighted in pink) found by grid (Figure 2.6a) and random search (Figure 2.6b) techniques. The best randomly sampled configuration is definitely better than the one found by the grid search.

Heuristic search parameter tuning. By their nature, heuristic algorithms may be applied to tackle the most black-box search problems. Since the parameter tuning is one concrete type of search problem, it is also often tackled by a high-order heuristic approaches (meta-, hybrid-, hyper-heuristics) ??????. The advantage of heuristics usage lays in their ability to learn during optimization process and use an obtained memory to guide the search more efficiently.

[1] Original graphics are taken from [65]

Model-Based Search Parameter Tuning. In the most cases, the dependencies between tuned parameter values and optimization objective do exist and can be utilized for hyper-parameter tuning. By predicting which parameter combinations lead to better results, model-based tuning could make precise guesses. As it showed in Figure 2.6c, after accumulating enough information the learning algorithm starts making more precise guesses, which in contrast to previously described model-free approaches are desirable and more robust.

Naturally, this optimization problem could be tackled by almost every discussed in this thesis approach. However, taking into account the fact that (1) TS is often a *black-box* we eliminate exact and approximate solvers, (2) the evaluation cost is high therefore, it is not desirable to apply the described above heuristics directly, since they require to perform a high number of TS evaluations to find a good configuration. With this idea in mind, researchers started to (1) create optimization algorithms that traverse the search space of configurations more efficiently and (2) build models that could imitate the dependencies between parameters and objective values, a so-called *surrogate models*. The former direction is nothing else but an enhancement to already existing optimization techniques, which allows accumulating and utilization of more information, obtained during an optimization. The later approach is crucial for problems where TS is expensive to evaluate. It often used as an enhancement in optimization algorithm, enabling one to simulate evaluation of real system instead of expensive direct evaluations. Still, it is a frequently used approach to combine the previously reviewed search space traversal techniques such as evolutionary algorithms, simulated annealing, tabu search with the surrogate models for optimization.

2.3.2 Systems for Model-Based Parameter Tuning

Parameter tuning is an obligatory task when the maximum system performance is a must-have requirement and should be performed at the design time. Novel tuning approaches are usually built in form of frameworks with exposed hooks for attaching the TS.

Since the target system evaluations here supposed to be costly, parameter tuning frameworks are trying to utilize every single bit of information from evaluations and a creation of surrogate models and using efficient optimization approaches is obligatory.

In this section we review some existing open-source parameter tuning systems from following perspectives:

- **Conditional parameters support** is a provided to user and handled by tuning system ability to describe conditional dependencies between hyper-parameters. As an example, imagine *crossover type* parameter of genetic algorithm that takes only some specific values: partially mapped crossover (PMX), Cycle Crossover (CX), etc. Binding a concrete crossover type, one will be required to provide parameters for this concrete crossover type, but to eliminate the respective parameters for other crossover types. This type of dependency could be described in form of a parent-child relationship, however other types of dependencies also exist.

- **Parameter types support** is one of the basic tuning systems usability requirements. More concretely, TS parameters could be not only a numerical (integer or fractional) but also a categorical in form of strings, boolean values, etc. Considering categorical data types, they could be either *nominal*, which depict only possible atomic values, or *ordinal*, which imply also value comparison but no distance notion. For instance, let us refer to genetic algorithm parameters. *Population size* is of numerical integer type in range $[1..\infty)$, *mutation probability* is of numerical fractional type in range $[0..1]$, *crossover type* is of categorical nominal type with $\{\texttt{PMX}, \texttt{CX}\}$ choices. Note, we could also view the population size as a set of finite values $\{10, 100, 1000\}$ therefore, turning it into the categorical ordinal parameter.

- **Extensibility** is crucial when someone would like to try a new promising and intriguing learning algorithm for guiding a search, that was not available in the parameter tuning system yet. Practically, one may need not only new learning algorithm but some other features like a non-trivial stopping criterion, tools for handling stochastic behaviors, or different strategies for random sampling (which are used to replace a model-based prediction, while the tuning system is learning).
- **Parallel evaluations** required for utilizing available computational resources that could scale horizontally, providing simultaneous evaluation of multiple configurations. This often could speeds-up the learning process.

Among reviewed systems we could distinguish those, which were created directly to face the parameter tuning problem and the others that are more generic optimizers but still applicable in parameter tuning cases. A concrete optimizer will be usable for searching the parameters, if it exposes several features. Firstly, it must consider an optimization function evaluation to be expensive and tackle this problem explicitly. For instance, using surrogate models or the other TS approximations. Next, a potential tuner should be able to tackle dependencies and conditions among parameters.

SMAC

Sequential Model-based Algorithm Configuration (SMAC [1]) [53] is a system for parameters tuning, developed by the AutoML research group (here we review the 3^{rd} version of SMAC).

In their research, scientists generalized the process of parameter tuning under the *Sequential Model-Based Optimization* (SMBO) term as the iterations between (1) fitting models and (2) using them to choose the next configurations for evaluation. This term naturally formalizes most of the existing (except MBMO, but we do not tackle multi-objectiveness in this thesis) parameter tuning approaches and may be used as a distinguishing characteristic of optimization algorithms, since they naturally could be applied not only to the parameter tuning problems.

SMACv3 is an extension introducing the learning models and sampling mechanisms to previously existing random on-line aggressive racing (ROAR) algorithm. The authors showed that a machine learning in general and regression models in particular (playing the role of surrogates) are applicable not only for parameter tuning but also for optimizing any expensive black-box function.

The development of this system was motivated to tackle the existing limitations of all published SMBO approaches, namely, expanding the applicability not only to numerical but also to categorical parameters and optimizing the target algorithm performance not only on a single but on number of problem instances (benchmark set), reducing the variance influence.

A routine in SMAC could be viewed as an iterated application of three steps: (1) building a learning model, (2) using it for making choices which configurations to investigate next and (3) actual evaluation of TS.

The evaluation (3) here is carried out by the original ROAR mechanism, where the running of each new candidate solution continues until enough data (from benchmark set of problem instances) is obtained to either replace the current solution or reject the candidate. On contrary to model-less ROAR, SMAC at step (1) builds the regression random forest surrogate — an instance of machine learning algorithm [19]. The usage of the regression decision trees is motivated by the fact that they fit well to categorical data and complex dependencies in general. Later, at step (2) an iterative local search (ILS) heuristic is applied in combination with *expected improvement* (EI) evaluation (part of the Bayesian optimization process) [98]. The EI is a measurement of possible solution quality improvement obtained by an underlying configuration therefore, higher EI means the candidate is better. ILS starts

[1] SMACv3 GitHub repository `https://github.com/automl/SMAC3/`

at the best previously found configuration and traverses its neighborhood distinguishing between configurations using EI and regression model built at step (1). SMAC compares configurations by means of the objective value and considers only minimization case. EI is large for those configurations, which have a low predicted cost and for those, with high uncertainty in results therefore, providing the exploration-exploitation balance [57].

IRACE

IRACE [1] is a hyper-parameter tuning package [74] as the implementation of *iterated racing algorithm* [16].

The underlying methodology is somewhat similar to the one implemented in SMACv3 and comprise three main steps: (1) sampling new configurations using a prior knowledge, (2) empirically finding the best ones among the sampled using the racing algorithm and (3) updating the prior knowledge to bias future samples towards better configurations. The prior knowledge here is represented as a probability distribution of values for each parameter independently (truncated normal and discrete distributions for numeric and categorical parameter types respectively). During step (3), the probability distributions are build using the best found in step (2) configurations, increasing a promising values sampling chance.

Iterated racing step (2) here is a process of running the target system using sampled configuration on a set of problem instances. After solving each instance, the statistically worse-performing configurations are rejected and racing proceeds with remaining ones. This process continues until it reaches the predefined number of survivals, or after solving a required amount of problem instances (in this case all remaining configurations are considered to be good).

IRACE supports various data types, such as numerical or categorical and the possibility of conditions description as well. While the problem of data types is resolved by the usage of different underlying distribution types, the conditional relationships are handled by the dependency graphs. During step (1), non-conditional parameters are sampled firstly and only afterwards, if the respective conditions are satisfied, the dependent parameters are sampled.

HpBandSter

A distributed Hyperband implementation on Steroids (HpBandSter [2]) is the realization of BOHB algorithm [37] in the software framework. While SMAC outperforms and partially reuses the decisions made in ParamILS [54], BOHB (Bayesian optimization combined with Hyperband) is the parameter tuning tool that outperforms SMAC and was created by the same AutoML research group.

As it stated in the framework name, the SMBO routines are carried out with mainly two algorithms: learning and configurations sampling is performed by the Bayesian optimization (BO) technique *tree parzen estimator* (TPE), while evaluation of sampled configurations and their comparison are carried out by the Hyperband (HB) algorithm.

The TPE usage instead of naïve Gaussian processes-based BO and expected improvement evaluation was motivated by a better dimensional scaling abilities and an internal support of both numerical and categorical parameter types. However, some minor transformations are still required. Unlike vanilla BO, where the optimization is done by modeling the result distributions given the configuration parameters, TPE builds two distributions of parameter values. It splits the configurations into two sets distinguishing their 'goodness'. During the sampling, it proposes those parameter values, which have a high probability to be in the 'good' distribution and simultaneously low probability to be in the 'bad' one. For more detailed explanation we refer to TPE description given in [10].

[1] IRACE GitHub repository `https://github.com/MLopez-Ibanez/irace`

[2] HpBandSter GitHub repository: `https://github.com/automl/HpBandSter`

A central part of BOHB, namely the Hyperband algorithm, is a promising bandit-based strategy for hyper-parameter optimization [72], in which the *budget* for entire parameter tuning session is defined beforehand and divided between iterations. The role of budget could play any setting that controls the accuracy of a configuration evaluation by TS, where estimation with the maximal budget provides the most precise configuration evaluation, while the minimal budget results in the least accurate approximation of configuration evaluation result. The running examples of budget could be a number of iterations in iterative algorithm, a number of epochs to train the neuron network, or a number of problem instances from benchmark set to evaluate. As a result, the requirements in TS arise to expose and support the budget usage as expected in BOHB.

At each iteration, in original version HB samples a number of configurations uniformly at random. The authors introduced an *intensification* mechanism according to which, a number of per-iteration sampled configurations decreases for the later iterations, while the amount of budget given for iteration remains the same. As an outcome, first iterations of HB are full of coarse-grain evaluated configurations, while the later iterations produce a higher number of more precise measurements. Each iteration of HB is split to the number of *successful halving* (SH) procedure executions, which at each execution drop poorly performing configurations (usually $2/3$). As one may expect, since the number of measured configurations in subsequent iterations decreases, the amount of SHs execution drops too therefore, the remaining configurations are evaluated more precisely.

The binding of Hyperband and Bayesian TPE is made in several places. Firstly, the learning models are updated each time, when new results are available for every budget value. Next, at each HB iteration instead of random sampling, the TPE model is used to pick next configurations. Note that BOHB uses only the surrogate, which was built on the configurations evaluated with the largest budget. This decision results in more precise surrogate models and therefore, better predictions in the later stages of parameter tuning process.

BRISEv2

The great part of software potential lays not only in its ability to tackle a problem at hand but also on the general usability and adaptivity to unforeseen tasks. Here we review a 2^{nd} version of BRISE[1] [88], since the very early BRISE versions (major version 1 [89]) were more monolithic and hard to apply for parameter tuning problem at hand.

While designing this system, authors were focused not solely on learning mechanisms for parameter prediction but on the overall system modularity as well. Being a software product line (SPL), BRISE was designed as a set of interacting components (nodes), each acting according to its own specific role. The system could be viewed from two perspectives. One is a birds-eye view on all available nodes with their roles and the other is a fine-grained description of *main-node* concretely.

Before reviewing each perspective, it is worth to justify the central terms used in the system. Note, that some of them are similar to the defined above, but they were explicitly implemented in form of interacting entities.

- **Experiment** encapsulates the information about a concrete run of BRISE. For instance, within a parameter tuning session it carries such information as BRISE experiment description (a specification of parameter tuning procedure in JSON format), all evaluated during session configurations with their results.

- **Configuration** is a combination of input parameter values for TS. It could be run several times to obtain a statistical data therefore, contain number of *Tasks*. Naturally, the configurations are

[1] BRISEv2 GitHub repository: `https://github.com/dpukhkaiev/BRISE2/releases/tag/v2.3.0`

comparable in terms of the results averaged over performed runs.

- **Task** is a single evaluation of TS under provided configuration for specified in description problem scenario.
- **Parameter** is a meta-description of concrete configuration part and a building block for search space. It defines a set or range of possible values.
- **Search space** comprises all parameters and their dependencies and could verify the validity of configuration.

From a birds-eye view perspective, BRISE consists of *main-node* as the system backbone, several *worker-nodes* as target algorithm runners under provided configurations, *event-service* to distribute tasks between available worker-nodes, *front-end-node* to control and report optimization process on a web-page, and non-obligatory *benchmark-node* that could be handy for executing and analyzing a number of experiments.

The main-node is a combination of objects, which interact in terms of queue callbacks. Therefore, when a new configuration is evaluated, the new model is built and used for the next configuration prediction. The intent of introducing aforementioned terms is to use them as a core of the framework, while such components as *prediction models*, *termination criteria*, *repetition management* or *outliers detection* are exposed to client for the variability reasons. Naturally, the developers also created a set of available out-of-the-box implementations for each variability component.

To use BRISE for parameter tuning, one should (1) construct an experiment and search space descriptions in JSON format and (2) add the respective target system evaluation logic in *workers*. All the rest will be carried out by the system.

2.3.3 Parameter Control

Generally speaking, the biggest disadvantage of parameter tuning approaches is defined by the fact that they usually require many TS runs to evaluate its performance with different configurations. On the opposite site, the parameter control approaches are solving this issue but the drawback lays in their universality.

The advantageous characteristic of any system is its ability to adapt at runtime. It could happen so, that an algorithm with tuned parameters performs well in the early beginning of a problem solving process but in later stages it struggles. The other algorithm configuration may result in an opposite behavior. This could be caused by various reasons and it is often hard to tell, which of them the algorithm is facing at the moment.

In contrast to the parameter tuning approaches, where optimal parameters are *firstly* searched and only afterwards are used to solve the OP, the parameter control is an approach of searching the parameters, *while* solving the OP. It also could be expressed as a system reaction to the changes in a solving process. Sometimes, it is named as an *on-line* parameter tuning. The drawback of this approach lays in a lack of generality, since often a parameter control technique is embedded into an algorithm therefore, is algorithm-dependent.

The only one broad classification facet we were able to distinguish is the *type of control mechanism*, where the *deterministic* and *adaptive* strategies exist. The first type suggests changing the parameters in a predefined schedule, while the second type assigns the parameter values upon received feedback. To the best of our knowledge, the adaptive approaches are mostly dependent on the concrete algorithm instance. Therefore, it is hard to present a generic classification of parameter control approaches for all algorithms however, this could be done for each particular algorithm family.

We provide an insight of the parameter control reviewing the examples of proposed strategies for some meta-heuristics. For the more comprehensive review of the recently published strategies we encourage the reader to examine the source paper, used here [52].

Parameter Control in Simulated Annealing

The most frequently controlled parameters in simulated annealing algorithms are the *cooling rate* (the velocity of temperature decrease) and the *acceptance criteria* (decision, whether to accept a proposed solution, or not).

The control in cooling rate parameter is motivated as follows: if the temperature decreases too rapidly, the optimization process may settle in the local optima, but too low cooling rate is a computationally expensive, since SA requires more TS evaluations to converge. Among deterministic approaches, researchers mainly distinguish linear, exponential, proportional, logarithmic and geometrical cooling schedules. In contrast to deterministic approaches, in [59] the authors proposed an adaptive strategy to change the cooling rate, based on the statistical information, evaluated on each optimization step. Concretely, if the statistical analysis, named in research as a *heat capacity*, shows that the system is unlikely to be trapped in the local optima, the cooling rate is increased. On contrary, it is decreased if the possibility of being trapped is high.

In [45] the authors propose an adaptation of another hyper-parameter: acceptance criteria. The utilized mechanism is based on thermodynamics fundamentals, such as *entropy* and *kinetic energy*. The authors suggest replacing the standard acceptance criteria (based on the current temperature and the solution quality) with the one based on the solutions entropy change evaluation.

Many researches were made to investigate, which is the best among deterministic and adaptive strategies [5, 28, 55, 75, 104]. In many cases, the authors conclude the adaptive methods provide more robust and promising results.

Parameter Control in Evolutionary Algorithms

While searching the parameter control examples in heuristics, one will find dozens of proposed methodologies for the evolutionary algorithms. It is arising from the fact that an idea of changing the algorithm parameters dynamically came from in EAs [61]. The motivation for such number of performed studies lays in the strong dependence of an algorithm performance on parameter values.

The deterministic and adaptive mechanisms in EAs are extended by the 3^{rd} type — the *self-adaptive* approach. It implies an encoding of parameter values in the solution genomes therefore, allowing them to co-evolve with the solutions at runtime [30]. All the proposed strategies could be split into two families: one includes the algorithms proposing to adapt a concrete parameter solely and the other, which includes the approaches to control a group of parameters. In EAs the commonly implicated hyper-parameters are the *population size*, the *selection strategies* and the *variation aspects* (namely the crossover and mutation operators). In case of interest, we propose the reader to analyze recently conducted reviews and researches dedicated to the parameter control in evolutionary algorithms [1, 30, 61, 99].

There is one rather intriguing parameter control approach proposed for the evolutionary algorithms. In [60] the authors introduce a reinforcement learning (RL) parameter controller, which goal is to select the EA parameters on-line, evaluating a set of simple observables. They include: a genotype diversity, a phenotype diversity, a fitness standard deviation, a fitness improvement and a stagnation counter. The RL in this work is executed following MAPE-K methodology [20]. On each iteration the observables are **m**onitored, their values are **a**nalyzed to build a parameter control **p**lan, which is **e**xecuted in the next iteration. The letter **k** denotes the central part of thus methodology — the knowledge. This set

of proposed observables could be split onto two logical groups. One is the algorithm specific with the genotype/phenotype diversities and the fitness standard deviation, while the other is algorithm-independent and includes the fitness improvement and the stagnation counter. We believe the proposed RL approach could be applied to other algorithms, with the only requirement in exposing the observable knowledge. The proposed in [60] parameter control methodology may one of the first to generalize a parameter control techniques and later in Section 3.1 we use it as a part of our approach.

2.3.4 Conclusion on Parameter Setting

At this point, we finalize our review of the parameter settings problems with a three conclusions:

1. The parameter tuning area is investigated widely and nowadays the research settles in form of combining different learning models to implement the SMBO algorithms in framework-like tuning systems.

2. The parameter control is actively driven by two motivations. Firstly, the runtime changes in solving process are unpredictable therefore, the control is believed to better fit them, comparing to the statically defined by tuning techniques parameter. Secondly, the resources spent for off-line parameter settings are high, but they are paid-off by a high quality of algorithm configuration therefore, from this perspective, the control approaches are trying to reach the off-line settings quality spending less resources. Unfortunately, nowadays the on-line approach is not generic to be commonly applicable.

3. The decision on concrete technique is use-case specific and is driven by the amount of available resources and the required setting quality as well.

2.4 Combined Algorithm Selection and Hyper-Parameter Tuning Problem

The goals of automatic machine learning are quite similar to persecuted by hyper-heuristics. They both operate on search space of algorithms (or their building blocks) which later are combined, with an objective to find the best performing one, and used to solve the problem at hand.

In this section we review one particular representative of automatic machine learning systems. Based on ML framework Scikit-learn [86], Auto-sklearn system [40] operates over number of classifiers, data and features preprocessing methods **including their hyper-parameters** to construct, for given dataset, the best performing (in terms of classification accuracy) machine learning pipeline. This problem was formalized as *combined algorithm selection and hyper-parameter tuning problem* (CASH) and presented previously in Auto-WEKA [105] system. Intuitively it could be rephrased as follows: "For given optimization problem, find the best performing algorithm and its hyper-parameters, among available, and solve the problem".

Note, how Auto-sklearn is in some sort similar to HHs, which use LLHs for traversing a search space and solving the OP. CASH problem seems to us as union of problems solved by HHs and parameter tuning approaches. We also found that the architecture search problems [36] (related to neural networks) are nothing else but particular case of CASH.

Turning back to Auto-sklearn, the crucial decisions made here is the combination of off-line and on-line learning, resulted into state-of-art performance of Auto-sklearn in classification tasks.

During the off-line phase, for each of available dataset, published by OpenML community [41], the search of best performing machine-learning pipeline was done using BO technique implemented in

discussed previously SMAC [53] framework. After that, the *meta-learning* executed to conduct the meta-features for each dataset. The entropy of results, data skewness, number of features and their classes is a sparse set of meta-features used to characterize a dataset (overall number is 38).

The resulting combination of dataset, machine learning pipeline and meta-features were stored and later used to seed the on-line phase of pipeline search. The information from meta-learning phase is used as follows: for a given new dataset, system derives the meta-features and selects some portion of created during off-line phase pipelines, that are the nearest in terms of meta-feature space. Then these pipelines are evaluated on a new dataset to seed the BO in SMAC, which results in ability to evaluate, in principle, well-performing configurations at the beginning of tuning process.

During the on-line phase, the other crucial improvement was introduced. Usually, while searching the best-performing pipeline, a lot of effort spent to built, train and evaluate intermediate ones. After each evaluation, only the results and pipeline description are stored but the pipeline itself is discarded. In Auto-sklearn, however, the idea lays in preserving previously instantiated and trained pipelines, obtained while solving the CASH problem. Later they are used to form an ensemble of models and tackle final problem together. This mean, that the results of this architecture search is a set of models with different hyper-parameters and preprocessing techniques, rather than one model. This ensemble starts from the worst performing ones (obtained at the search beginning) and ending by the best suited for dataset at hand. Naturally, their influence on final results are weighted.

However, the potential of off-line phase is derived entirely from the existence of such dataset repository and depends on availability of homogeneous datasets. The proposed on-line methodology, which mimics the regression trees, is more universal and could be reused widely.

In general, the empirical investigation of proposed approach universality would be rather intriguing, since the only cases of Auto-sklearn application we found are the classification tasks but not a regression problems [13, 40].

The field of automated machine learning is one of trending research directions, that is why there exist dozens open-source systems, such as *Auto-Weka* [105], *Hyperopt-Sklearn* [66], *Auto-Sklearn* [40], *TPOT* [83], *Auto-Keras* [56], etc. Among open-source, there are many commercial auto-ml systems, such as *RappidMiner.com*, *DataRobot.com*, Microsoft's *Azure Machine Learning*, Google's *Cloud AutoML*, and Amazon's *Machine Learning on AWS*.

2.5 Conclusion on Background and Related Work Analysis

In this chapter we have presented the review of optimization problems, their concrete instances and existing solver types. Ruffly speaking, there exist several levels of generality in heuristic solvers: simple heuristics, meta-heuristics and hyper-heuristics.

The applicability of each algorithm is problem-dependent and derived from exploration-exploitation balance and strength, revealed in particular case. It is hard to guess beforehand, which algorithm will outperform the others in an unforeseen use-case. With respect to this, hyper-heuristics seems to be the most perspective and universal solvers, since they do not tackle the problem directly, but rather select and apply the best suited among controlled algorithms.

From the other perspective, the solver performance is also dependent on values of its parameters, often called the *hyper-parameters*. It turns out, that the parameter setting is also an optimization problem. There exist several ways to solve it: (1) set the values manually, basing on own experience and intuition, (2) utilize the tuning systems with find the best values automatically and later use those found parameters, or (3) exploit the parameter control mechanisms, which are often embedded into solvers themselves. Among all strategies, parameter control seems like the golden middle, since tuning requires lots of expensive algorithm runs to produce a good parameter settings, while manually choosing

hyper-parameters is an error-prone process, that requires experienced guidance.

The outcome of no free lunch theorem can not be ignored, according to which no single algorithm can tolerate broad range of problems equally outperforming other solvers. That is why we can not set aside hyper-heuristics, which are designed to select the best suited, in particular case, problem solving algorithm.

The research in automatic machine learning made step further and are tended to combine both algorithm selection and parameter tuning problems into single CASH problem, formalized in [105]. The search space in CASH problem is formed of algorithm variants and their respective hyper-parameters. However, one solver can not use the parameters of another, thus the resulting search space happen to be 'sparse'. In general, the structure of CASH problem is almost the same as regular parameter tuning case. That is why the commonly used solver for CASH problem are systems for parameter tuning: SMAC in Auto-sklearn and Auto-Weka, Hyperopt in Hyperopt-Sklearn and so forth. Not many surrogate models are able to handle such search spaces: random forest machine learning model and Bayesian optimization approaches with exotic kernel density estimators [70]. Even fewer optimizers are able to perform well in such sparse spaces. The other drawback, is that the CASH problem definition is limited to searching the algorithm and its parameters in the off-line manner.

A scope of the thesis is defined at this point.

We believe, that the results of both problems, namely algorithm selection and parameter settings have a reverse dependency on problem at hand. That is why, a search for the best tool (solver) and its setting (parameters) should be performed in on-line manner, in other words, while solving the optimization problem.

As CASH merge algorithm selection and parameter tuning techniques to get the outstanding performance, here we found a merge of on-line algorithm selection and parameter control problems an intriguing and worth-to-try idea. Thus, in this thesis we are trying to achieve the best of both worlds. The resulting approach should be able to solve an optimization problem, applying the best suited low-level-heuristic and setting its parameters in the runtime. With this idea in mind, we investigate a possibility of turning existing parameter tuning system into on-line selection hyper-heuristic with parameter control in low level heuristics.

3 Concept Description

While there exist no universal approach to control the algorithms parameters (Section 2.3.4), our conclusion on the literature analysis was the absence of existing approaches to combine both on-line techniques for the algorithm selection and the parameter settings (Section 2.5).

In this Chapter we propose the methodology to resolve this problem, excluding the implementation details.

In Section 3.1, we introduce the generic parameter control technique and expand it with the use-case of algorithm selection. As concluded in Section 2.5, the main weakness of the reviewed approaches to tackle CASH problems lays in the inability of learning mechanisms to fit and predict in sparse search spaces. The same issue arises in case of on-line algorithm selection and parameter settings, and we resolve it on two levels: firstly in the search space structure and secondly in the prediction process. In Section 3.2 we present the joint search space of both algorithm selection and parameter control problems. We outline the functional requirements for such space. Next, we describe the related prediction process in Section 3.3. While decoupling the learning models from the search space structure, we provide the certain level of flexibility in the usage of different learning models. Finally, in Section 3.4 we direct our attention to the low level heuristics (LLH) — a working horses of the desired hyper-heuristic. We highlight the requirements to LLH that are crucial in our case.

3.1 Combined Parameter Control and Algorithm Selection Problem

The base idea of the parameter control approaches lays in the solver behavior adaptation as the response to changes in the solving process (Section 2.3.3). As we mentioned during the heuristics review (Section 2.2), the algorithm performance is highly dependent on the provided exploration-exploitation balance, which in turn, depends on (1) the algorithm itself and (2) its configuration. The task of parameter control is to find the later, which provide the best performance.

In our work, we solve the parameter control problem utilizing a similar to proposed in [60] reinforcement learning (RL) approach for evolutionary algorithms. The underlying idea of RL could be described as a process of performing actions in some environment in order to maximize the reward, obtained after each performed action. To apply this technique onto the parameter control problem, we must define what are those *actions* and how to estimate the *reward*. Thus, for making the parameter control applicable to broad range of algorithms, we analyze not the solver state itself but the optimization process (in [60], the authors use both algorithm-dependent and generic metrics). To realize the MAPE-K control loop, we must interrupt the solver, analyze the intermediate results, learn the current trend among parameters, configure the solver with the most promising parameter values and continue solving. The number of MAPE-K loop iterations i define the granularity of learning, where one should balance between *time to control* (TTC) the parameters vs *time to solve* (TTS) the problem. Naturally, the limitation of proposed approach is the use-cases, where $TTS >> TTC$.

Yevhenii: Should I highlight the limitation(s) here or in conclusion and refer from here?

To evaluate the gained in iteration i reward, instead of using straight solution quality value, we calculate the quality improvement, obtained with the provided configuration C_i. When the search process converges towards the global optimum, the improvement value tends to decrease, since the

amount of significantly better solutions drops. Using the improvement values directly or could confuse the learning models and therefore, cause the prediction quality to struggle. To resolve this issue, the relative improvement (RI) of solution quality is calculated using Equation (3.1), where S_{i-1} and S_i are the solution qualities before and after i^{th}) iteration respectively.

The evaluated $C_i \rightarrow RI$ pairs in previous iterations are then used to predict the configuration for next iteration C_{i+1}. At this point, we made two decisions in the sampling process: (1) hide the search space shape and (2) use the surrogate models for finding configurations that lead to the highest reward.

$$RI = \frac{S_{i-1} - S_i}{S_{i-1}} \tag{3.1}$$

After sampling the C_{i+1} configuration, we set it as the solver parameters. To proceed with the solving process, we seed the solver with the solutions from $i - 1$ iteration as well.

When it comes to the algorithm selection problem (discussed in Section 2.2.5), we treat the solver type itself as the subject of parameter control and use the proposed RL approach to estimate the best performing algorithm. However, when we add the solver type as a parameter, the resulting search space become sparse and requires special treatment. Two commonly used approaches for tacking this problem exist. The first requires special type of learning models, while the second suggests the problem transformation in a way of excluding the undesired characteristics.

During the review of model-based parameter tuning approaches (Section 2.3.1), we made a conclusion that all reviewed systems follow strictly the first idea. For instance, as the surrogate models, BOHB [37] and BRISE [88] use the Bayesian probability density models. Those surrogates could naturally fit to the described search space shape, but the proposed approaches are not able to make the predictions effectively, since the most of predicted configurations will violate the dependencies. As the illustration, imagine after i^{th} iteration, the surrogate models learn about two superior parameters: one indicates a well-performing heuristic type (the Genetic Algorithm), the other — an effective configuration for another algorithm type (an exponential cooling rate for the Simulated Annealing). In this case, the reviewed systems sampling methods will tend to predict the invalid configurations with those two parameter values.

In this thesis we follow the second approach namely, the problem transformation in order to sample the valid configurations only. The following Section depicts a required preparation step, made in the search space, while the later is dedicated to the prediction process.

3.2 Search Space Structure

When the time comes to selecting not only the solver parameters but also the solver itself, the united search space no longer could be presented as 'flat' set of parameters since it tends to appearance vast amount of invalid parameter combinations. Let us estimate the number of all possible configurations vs the amount of meaningful ones. Suppose, we have the N_s number solver types, each exposing the $N_{s,p}$ number of hyper-parameters with the $N_{s,p,v}$ number possible values. The aggregated quantity of configurations N_c in the disjoint search spaces is calculated as the number of possible combinations using Equation (3.2).

$$N_c = N_s \cdot \prod_{1}^{N_{s,p}} N_{s,p,v} \tag{3.2}$$

However, if we decide to tune (or rather to control) the solver type itself, the resulting quantity of possible configurations is calculated using Equation (3.3).

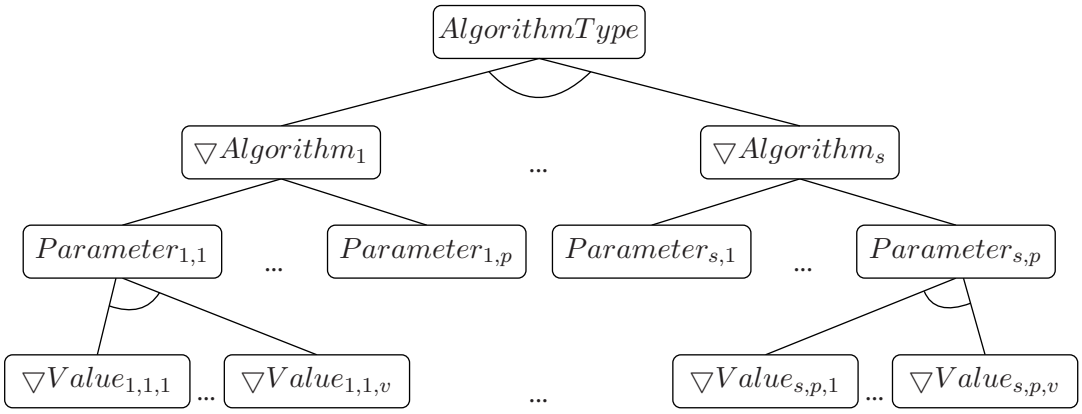


Figure 3.1 Search space representation.

$$N_c = \prod_{1}^{N_s} \prod_{1}^{N_{s,p}} N_{s,p,v} \tag{3.3}$$

For the better intuition, lets try some numbers. By setting all $N_s = N_{s,p} = N_{s,p,v} = 3$ (the rather small example), the amount of configurations estimated separately for each solver equals to $N_c = 81$ (Equation (3.2)). However, if we join the solver parameter spaces, Equation (3.3) shows the significant growth in the search space size: $N_c = 19683$. Note, the number of *really unique* configurations remains the same thus, in the joint space it is only $\approx 0.4\%$. By setting the $N_s = N_{s,p} = N_{s,p,v} = 4$, this number drops to $\approx 9 \cdot 10^{-8}\%$. It could decrease even further if the dependencies among hyper-parameters exist. In such case, the predictive abilities of models may straggle.

To overcome this, we utilize similar to the utilized in IRACE [74] framework idea: *explicitly indicate the dependencies as a parent-child relationship among the search space entities p, firstly predict the parent parameter, afterwards — the children.* This give us an opportunity to threat the algorithm type as the regular categorical parameter, makes the search space structure uniform and simplifies the prediction process.

This decision sets the following search space *structural requirements*:

S.R.1 The **parent-child relationship** must describe the dependencies between different parameter types.

S.R.2 The **uniform parameter types** simplifies the structure and hides the domain-specific intent of each parameter thus, algorithm type and its hyper-parameters are treated in the same way.

S.R.3 The **value-specific dependencies** describe a concrete parent value(s), when the child should be exposed. For instance, the parameter *algorithm type* has a number of possible values, each of them requires own set of hyper-parameters, which should remain hidden for the other solver types.

Figure 3.1 shows an example of such search space with s algorithm types, each having p parameters with v possible values. The entities with triangles $\bigtriangledown$, namely the concrete values of parameters, form the joint-points to which the other parameters could be linked.

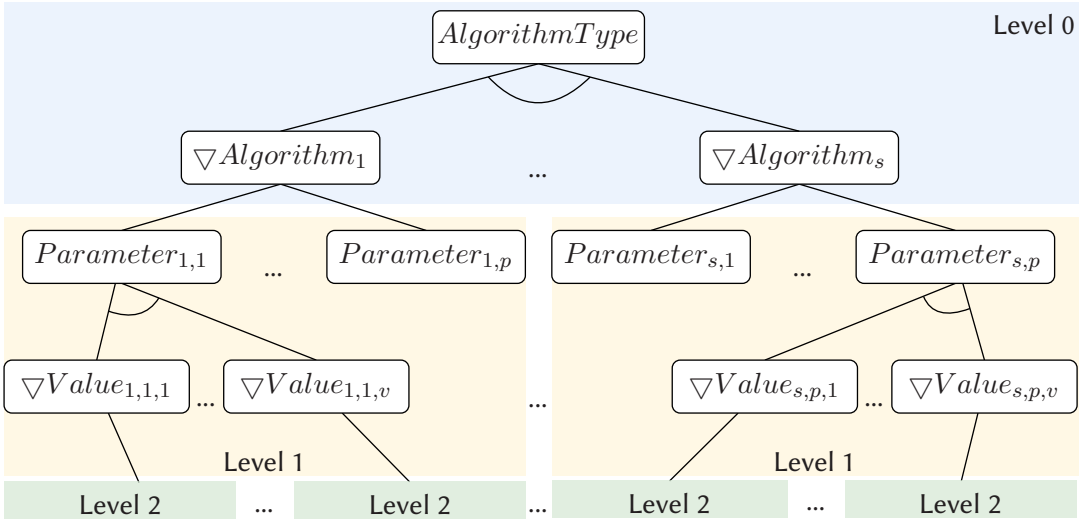


Figure 3.2 Level-wise prediction process.

3.3 Parameter Prediction Process

After formalizing the search space structural requirements, let us switch to the prediction process and define the *functional requirements* for both search space and prediction process, which should be fulfilled to decouple the learning models from the complex search space shape.

The idea of this decoupling lays in resolving the value-specific dependencies among the parameters in a step-wise prediction approach. To do so, we firstly predict the parent value, which in case of the hyper-heuristic is a low-level heuristic type (Level 0 in Figure 3.2). Afterwards, the search space must expose the parameters of this solver only, ignoring the others (Level 1 in Figure 3.2). The dependencies among exposed parameters, are then handled in the same way (Level 2 and further in Figure 3.2).

The prediction on each level is performed in three main steps: (1) filtering the required for this level information, (2) building the surrogate model and (3) finding the best performing parameters on this level.

While building the surrogates and making the predictions, we ignore the information from levels above and below with the motivation to simplify the overall process and hide the search space structure. Also, when we predict on the parent level, it will not change on the descendant levels thus, we do not need to operate useless static information. While the backward ignorance is clear, the forward data omission puts a restriction on the surrogate models. Cutting off the parameter values from the deeper levels, we may get the data points with the same current level parameters values (also called as *features* in machine learning) but different results (*labels*). Thus, only those surrogate models should be used on such level(s), which will not be confused by the multi-valued dependencies in data (when the same input result in different outputs). During the implementation description in Section 4.3.3 we clarify, which models are the better choice in such cases and implement one of the promising.

Certainly, during the problem solving, the quality trends among parameter values may change. For instance, at later stages the domination of one solver could be declined in comparison to other. Or, the previously best-performing parameter values are not as good and should be replaced by the other. These changes may be caused by the various reasons, which we are not tackling. Instead, the old trends should be left out by some forgetting mechanism.

At this point, let us summarize the functional requirements.

- **In the search space we need:**

S.F.R.1 The **data filtering mechanism**, which will be used to find out only those feature-label pairs, which can be utilized to learn the dependencies on current level.

S.F.R.2 The **sampling propagation mechanism**, which will be used to randomly sample the parameter values for the next level taking into account currently available parameter values, which is required to expose the parameters after predicting on current level.

S.F.R.3 The **parameter description mechanism**, which will provide the information about a type and possible values for the given parameters. This knowledge will later be used by the models for making the parameters values prediction.

S.F.R.4 The **configuration validation mechanism**, which will find out, whether the parameter ranges are not violated by the selected values (flat validation), and whether for all selected values the dependent (exposed) parameters are selected properly as well (deep validation).

- **In the prediction models:**

P.F.R.1 The **model encapsulation mechanism**, which should aggregate and hide the level-wise approach of the search space traversal and the feature ignorance as well. On contrary, it should rely on underlying models for making the prediction.

P.F.R.2 The **model unification mechanism**, which is required for the system variability in terms of the learning and sampling algorithms.

P.F.R.3 The **information forgetting mechanism**, which is required to follow only the recent trends among the parameter values dependencies.

3.4 Low Level Heuristics

As we discussed during the hyper-heuristics review in Section 2.2.5, they are built of two main components — the high level heuristic (HLH) and the low level heuristic (LLH). Note the used *solver type* term in this Chapter is nothing else but the LLH in hyper-heuristic. The previous two sections were dedicated to the search space and prediction models description, which form the logical components of the HLH. No hyper-heuristic could work without LLH therefore, in this section we discuss the requirements for the low-level heuristics.

The proposed idea of the MAPE-K reinforcement learning application, implies the usage of anytime algorithms (see classification of solvers in Section 2.1.2). They may be implemented in various frameworks or even programming languages, the only requirement is to expose a common interface.

Firstly, we want these algorithms to continue their solving process from the previously found solution but not to start the process from scratch. Before the start, they should accept the predicted by HLH hyper-parameters, and the previously reached solution(s) (possibly, by the other solver).

Secondly, after the algorithm execution, the solution quality should be estimated and reported to the HLH to proceed with the RL.

Both actions should be performed in the implementation independent way therefore, following a predefined shared interface, described above. We discuss it in Section 4.4, dedicated to LLH implementation.

3.5 Conclusion of concept

When the requirements, specified for the search space and the prediction process are fulfilled, it provides a certain level of overall system flexibility in the following use-cases:

1. The **parameter tuning** case is possible, if one builds a search space of the single LLH, its hyper-parameters, and disables the solution transfer between the iterations.
2. The **parameter control** case is possible, if one builds a search space of the single LLH, its hyper-parameters, and enables the solution transfer between the iterations.
3. The **off-line selection hyper-heuristic** is possible, if one builds a search space of the multiple LLHs, and disables the solution transfer between iterations. In this case, the LLHs will be used with the static hyper-parameters.
4. The **on-line selection hyper-heuristic** is possible, if one builds a search space of the multiple LLHs, and enables the solution transfer between iterations. In this case, the LLHs will be used with the static hyper-parameters as well but seed with the solutions.
5. The **on-line selection hyper-heuristic with parameter control** is possible by building the search space of multiple LLHs, their hyper-parameters and enabling the solution transfer between iterations.

Note, that the off-line cases estimate the solution quality directly, while the on-line cases use the relative solution quality improvement.

It is worth to mention that the proposed structure of search space representation is similar to the *Feature Model*, used to describe the Software Product Lines (SPL) [97]. In Figure 3.1 and Figure 3.2 we used the notions from SPL feature models to denote an *alternative* parameter values. The process of configuration construction within a search space can be referred as the *Staged Configuration* in SPL.

4 Implementation Details

In this Chapter we dive into the development description of listed in Chapter 3 requirements.

The best practice in software engineering is to minimize an implementation effort reusing the already existing and well-tested code. With this idea in mind we select one of reviewed in Section 2.3.1 open-source parameter tuning frameworks as the code basis for desired hyper-heuristic. We are also planning to reuse the existing low level heuristics (LLH) implementation from other frameworks. While the LLH-basis may be used almost out-of-box, the HLH-basis requires changes to be applicable in our case.

In Section 4.1 we analyze the parameter tuning frameworks from a perspective of required effort to implement listed in Section 3.3 HLH characteristics. In Section 4.1.2 we conclude the analysis selecting the most suited HLH code basis. Afterwards, we split the HLH implementation description into two logical parts, as we have done during the concept description. In Section 4.2 we discuss the search space development, while Section 4.3 is dedicated to the prediction process. Finally, in Section 4.4 we perform a similar code basis selection for LLH, present a set of reused meta-heuristics, their adaptations and the process of importing them in our system.

4.1 Hyper-Heuristics Code Base Selection

Before starting with the framework analysis, let us firstly outline the important characteristics from the implementation perspective.

The first two crucial criteria are the framework *variability* and *extensibility*. The desired HLH should be easily variable in terms of changing such functionality as the learning models or the termination criteria to use a new one. We are also planning to use a possibly different models for the LLH selection and parameters control therefore, the code basis should be variable in terms of different model usage for each prediction level (see Figure 3.2).

The next is support for the *on-line optimization*. It is a bit complex system characteristic, which we are willing to distinguish. As it turns out, many parameter tuning systems require full evaluation of target system, while others are not, but they run evaluations isolated. In case of the desired hyper-heuristic, we treat the configuration evaluation as a trial to improve the problem solution using a particular LLH and tuning its parameters. It implies an important system ability to tackle the optimization problem (OP) using reinforcement learning approaches (Section 3.1).

The final characteristic is the *conditional parameters* support. Since, it is a complex feature laying not only in the search space but also in the prediction process, we pay a close attention to both of them.

4.1.1 Parameter Tuning Frameworks Analysis

SMACv3 We begin our review with the implementation of Sequential Model-based Algorithm Configuration framework, distributed under the BSD-3 license.

The idea of SMACv3 lays in an enhancement of ROAR mechanism with the model-based sampling algorithm.

The ROAR mechanism is a derivative from the FocusedILS algorithm (solver in the parameter tuning framework ParamILS [54]) where each evaluation of a new candidate solution on problem instance

performed sequentially and in isolation. Since the ROAR evaluation strategy is also used in SMACv3, we expect it to require much effort to enable system solving the problem on-line.

As we mention in Section 2.3.2, the underlying surrogate model in SMACv3 is selected statically among random forest or Gaussian process kernel density estimator. Both models could naturally fit complex dependencies among parameters. To sample next configuration, the *one-exchange* neighborhood of the best found so far configuration is traversed using the surrogate model and the expected improvement estimations. An ability of both surrogates to fit a sparse search space is promising, and the usage of expected improvement guarantees to converge the search process to the global optimum given enough time. However, the major drawback in this system is a lack of abilities to include the conditional dependencies between parameters into the sampling process. In fact, the used here search space representation framework ConfigSpace [73] is able to specify the dependencies among hyper-parameters. But, to the best of our knowledge, the one-exchange neighborhood used as sample mechanism in SMACv3, is unaware of the dependencies therefore, violates them during the parameters sampling resulting in illegal combinations. Those cases are controlled and rejected by the ConfigSpace, but we believe that in case of sparse search spaces it could lead to ineffective sampling and struggling in system predictive abilities.

Unfortunately, we did not find any officially published empirical studies of such cases and can only make guesses based on our own intuition, but SMACv3 developers advises for operation in such cases [1] may serve as an evidence to our assumptions correctness. One of the possible solutions here could be the usage of a conditional-aware one-exchange neighborhood definition for sampling process.

IRACE This framework implements the iterated racing algorithm to evaluate a set of configurations during the parameter tuning session (Section 2.3.2). The software is distributed under the GNU General Public License with an open source code.

The framework uses a *Friedman test* [27], or as an alternative *paired t-test* for statistical analysis of racing in parallel configuration. As the surrogate models, IRACE uses the probability distributions of those parameter values, which shown to be good during the racing step. The prediction process is defined as the step-wise sampling in previously derived distributions. It elegantly handles the conditions among parameters and illuminates the possibility of invalid configuration appearance. However, we found it to be static in terms of variability and extensibility on the learning mechanisms.

In terms of parallel evaluations, the algorithm utilizes all available resources at the beginning of each racing step, but as the process continues fewer evaluations are executed simultaneously therefore, the available resources are idling and not utilized completely at all steps of IRACE execution.

For the on-line problem solving, let us discuss the racing algorithm. As we described in Section 2.3.2, this step is executed with (1) a set of TS configurations sampled for evaluation and (2) a benchmark set of optimization problems. Afterwards, the TSs are starting to solve the problem set under each configuration, while the racing algorithm terminates worst-performing configurations. In case of hyper-heuristic, it is possible to use a single problem instance and the divided into parts TS running time, which at each pause synchronize the current solutions to proceed with the best found. Doing so, it will be possible to adapt the system for on-line problem solving however, the granularity of parameter control will be reduced. The reason for such reduction is the amount of information obtained after each race: only the best configurations are reported, leaving the performance evidences of the others behind. We believe, this information may be used to create a more precise surrogate models (for instance, TPE).

HpBandSter As we discussed in Section 2.3.2, HpBandSter is an implementation of BOHB algorithm, which turns to be a hybridization of Hyperband and Bayesian Optimization approaches.

[1] Visit SMACv3 repository `https://github.com/automl/SMAC3/issues/403`

A role of Hyperband in this duet is the configuration evaluation and comparison, while the Bayesian Tree Parzen Estimator (TPE) suggests which configuration to evaluate next. The idea behind this combination lays in elimination of weak sides in each algorithm with strengths of the other. For instance, in original Hyperband the configuration sampling is made uniformly at random, which results in a slow converge of an optimization process. As for the BO TPE, a drawback here lays in a configuration evaluation. Naïve Bayesian optimization approaches do not take into account the early evidences about the TS performance. Thus, even when the proposed configuration results in a poor starting and intermediate TS performance (which may be an evidence of a weak final performance), BO still continues TS execution. These facts motivated authors to merge those two algorithms to create one for parameter tuning with strong anytime (HB) and final (BO) performance. The resulting hybrid effectively uses available computational resources in parallel (HB) in combination with scalable and robust learning mechanisms (BO).

Let us discuss the process of handling the conditions between hyper-parameters. SMACv3 and HpBandSter as well use ConfigSpace framework for search space representation. As we discussed in SMACv3 description above, ConfigSpace naturally allows to encode the dependencies and conditions among parameters. The TPE learning models are also able to fit somehow these dependencies by means of the *impuration* mechanism[70]. Shortly saying, when fitting the surrogate models, the disabled parameters (or their values) are replaced with their default values. Later, while building the surrogate models those default values are ignored therefore, the probability densities still represent a proper parameter values distributions. However, consider an appearance of two configurations families: C_1 and C_2, such that some parameter P_i is forbidden in C_1 but is required in C_2. On contrary, the other parameter P_j is required in C_1 but forbidden in C_2. If these configuration families are turn to be superior, the resulting probability densities will be biased towards P_i and P_j values. As a consequence, the proposed prediction mechanism will sample non-default parameter values for both P_i and P_j, which results in the configurations with violated parameter dependencies. The sparser the search space, the more harming an effect will be in a prediction performance. One possible treatment here is to change the sampling process introducing an intermediate layer, which will perform a parameter prediction in level-wise approach suggested in Section 3.3.

BRISEv2 BRISEv2 is a software product line (SPL), created with an aim at solving the expensive optimization problems in general and for the parameter tuning in particular (Section 2.3.2).

The advantage of BRISEv2 over other systems comes from its *main-node* modular design. It is a set of cooperating core entities (Experiment, Search Space and Configuration) with other non-core entities, exposed to user for variability. The prediction models, termination criteria, outliers detectors, repetition strategies, etc. are representatives of these non-core and variable components. A number of implementations are provided out-of-the-box for all variable entities, but we focus our attention only to the implemented sampling process. The reason of such a greedy review is that the underlying search space representation is carried out by the same ConfigSpace, while the provided surrogate models are ridge regression model with polynomial features and Bayesian Tree Parzen Estimator (TPE). We are not going to repeat ourselves reviewing the ConfigSpace + TPE combination, but we have to put a few words about the ridge regression.

Ridge is the machine learning linear regression model with regularization [50]. Being a linear model, its abilities to fit a sparse search spaces is poor therefore, the machine learning community are suggested to treat such cases with *conditional linear regression models* [24]. The underlying idea is to split the search space into sub-search spaces and build a separate regression model there. But, to the best of our knowledge, this approach is not built-in into the used in BRISEv2 ridge regression model.

As for the on-line problem solving support, the routine of an optimization process, implemented in

BRISEv2 is nothing else than a reinforcement learning. After each new obtained evidence (configuration), a new surrogate model is built to react on the learning process by predicting the next configuration. This makes the on-line parameter tuning approach, presented in Section 3.1 embedding into BRISEv2 much easier.

4.1.2 Conclusion on Code Base

The most among reviewed parameter tuning systems share the same SMBO approach for problem solving. They utilize a rather similar techniques for building the surrogate models and making the predictions however, the different system architectures are implemented.

To sum um our review, we use a term *quality* to aggregate both (1) the provided out-of-the-box desired characteristic support and (2) the required effort to adapt it, if necessary. For the visual representation, we collect the reviewed characteristic qualities in each software framework into Table 4.1.

The quality estimates are quantized into three ordinal values:

1. **Poor** quality means the weak characteristic support and much effort required to improve it.
2. **Average** quality means the weak characteristic support and requires less amount of effort to provide it.
3. **Good** quality means a good out-of-the-box characteristic support, and requires minor or no changes at all.

Table 4.1 Code basis candidate systems analysis.

Characteristic	SMACv3	IRACE	HpBandSter	BRISEv2
Variability & extensibility	Average	Poor	Average	Good
On-line optimization	Average	Average	Average	Average
Conditional parameters	Poor	Good	Poor	Poor

Among the reviewed software systems, the majority were created as an implementation of some concrete algorithm (or a combination of algorithms), which results in the reduction of the system flexibility. Every reviewed framework requires much adaptation effort and preparation steps, which we will discuss in upcoming sections. Between such features as a proper support of conditional parameters vs variability and extensibility, the former plays a settle role in our case. Thus, from our perspective, BRISEv2 is the most promising candidate for the hyper-heuristic with parameter control creation.

4.2 Search Space

Previously, in Section 3.2 we presented a set of structural requirements for the search space representation: parent-child relationship should be presented explicitly, supporting different parameter types in a values-specific approach. To support the prediction process, in Section 3.3 we listed the functional requirements in a form of mechanisms: data filtering, sampling propagation, parameter description and configuration verification.

In this section we (1) analyze the available ConfigSpace framework, how it fits to our requirements and (2) decide whether to use it or to set it aside to perform our own search space representation implementation.

4.2.1 Base Version Description

From the structural point of view, in ConfigSpace[1] the parameter coupling is made implying parent-child relationship, which fit into our requirements. The set of parameter types suite the most of use-cases and the value-specific dependencies are supported as well. Thus, the structural requirements S.R.1, S.R.2 and S.R.3 are perfectly met.

When it comes to the functional requirements, ConfigSpace samples random configurations in a completion approach. In other words, there is no step-wise configuration construction, but only the final and valid ones are produced. To the best of our knowledge, there is no straightforward way to expose the underlying parent-child dependencies among parameters to distinguish a non-dependent, which is required for the prediction models, except of carrying them on a side and evaluating manually. As a consequence, the data filtering mechanism should be implemented on a side and the sampling propagation as well. The framework exposes an ability to validate a fully created configuration but not a set of parameters (flat validation). It worth mo mention that the configuration in this case turns to be a proprietary class. As for the parameter description, the amount of exposed knowledge is satisfying. Here we conclude that the functional requirements, except S.F.R.3, are not met.

As the conclusion, we decided to set aside the 3^{rd} party ConfigSpace framework. The reason for dong so is the amount of adaptation effort and the usage of foreign dependencies, more concretely: (1) it requires much adaptation effort and implies its usage as a core entity in BRISEv2, (2) it obligates us to replace the other core entity — Configuration, (3) it forces us to use a third proprietary entity — Hyperparameter.

4.2.2 Search Space Implementation

From the structural requirements we know that the parameters in search space should be treated uniformly. The desired feature tree shape is perfectly handled by the *composite* design pattern. With this idea in mind, we construct the search space as a composite *Hyperparameter* object with four possible hyper-parameter types: integer and float as numerical, nominal and ordinal as categorical. This fulfilling S.R.2, specified in Section 3.2. An implemented class diagram could be found in Appendix.

Yevhenii: add class diagram in appendix

In the code snippets provided through the explanation we highlight signatures of implemented methods, which fulfills our requirements specified in Chapter 3.

Search space construction. The S.R.3 implementation is performed by adding a construction method *add_child_hyperparameter* in the Hyperparameter class (Listing 4.1). It should be called on a parent object, specifying the activation value(s) (*activation_categories* argument) of parent hyper-parameter which are exposing the child.

```
1 class Hyperparameter:
2   ...
3   def add_child_hyperparameter(self, other: Hyperparameter, activation_categories: Iterable[
      CATEGORY]) -> Hyperparameter:
4     pass
5   ...
```

Listing 4.1 S.R.1 implementation.

[1] ConfigSpace GitHub repository: `https://github.com/automl/ConfigSpace`

Note, currently we support a composite construct only by means of categorical parameter types therefore, requiring a list of activation categories. We postpone a composition on numerical ranges enhancement for the future work, while our current needs do not include such cases.

Search space role in prediction. Imagine a number of configurations were already evaluated. For making the prediction in tiered approach, the parameter values on a current level should be selected before moving to the nest one. For it, we firstly filter data, which fits to this level by means of S.F.R.1. The filter accepts already chosen parameter values and iterates over the available configurations. At each iteration it finds out, whether the already chosen parameter values (*base_values*) form a sub-feature-tree of the parameter values under comparison (*target_values*). It is implemented in form of hyper-parameter instance method *are_siblings*, presented in Listing 4.2. As the outcome, this method decides, whether should this particular configuration be included into the dataset or not. For instance, if the selected LLH type in *base_values* is not the same as one in *target_values*, the result will be negative.

```
class Hyperparameter:
  ...
    def are_siblings(self, base_values: MutableMapping, target_values: MutableMapping) -> bool:
      pass
  ...
```

Listing 4.2 S.F.R.1 implementation.

After data filtering, the time comes to find out, which parameter values we must predict. For doing so, the search space, must expose the current level parameters by means of S.F.R.2 implementation in *generate* method in Listing 4.3. Since we always interact with a search space root object, the call to *generate* is executed recursively. If a callee finds itself in *values* argument (which depicts the current $parameter\ name \rightarrow random\ parameter\ value$ mapping), it redirects a call to all *activated* children. If it does not, it adds itself a to the *values* and terminates the recursion.

```
class Hyperparameter:
  ...
    def generate(self, values: MutableMapping) -> None:
      pass
  ...
```

Listing 4.3 S.F.R.2 implementation.

A randomly sampled for the current level values are then used for getting a description of current level and using it to (1) cut-off the data from levels above and below (simply selecting the required key-value pairs from the parameter mapping), (2) build the surrogate models and (3) make the prediction.

To build the surrogate models we require an available data (parameters) description. Thus, the S.F.R.3 implementation is performed in method *describe* presented in Listing 4.4. This is once again a recursive call, which terminates when the parameter object can not find the activated children or himself in the provided *values*.

The resulting description contains a mapping from parameter name to its type and the range of possible values: either a set of categories for categorical, or lower and upper boundaries for numerical types.

```
class Hyperparameter:
  ...
    def describe(self, values: MutableMapping) -> Description:
      pass
  ...
```

Listing 4.4 S.F.R.3 implementation.

This description is then used by the prediction models for building surrogates and making the predictions, which will replace a randomly sampled parameter values obtained after *generate* method call.

The described above process is controlled by S.F.R.4, implemented as method *validate*, presented in Listing 4.5. The control occurs in two places. Firstly, before starting a new loop of $filter \rightarrow propagate \rightarrow describe \rightarrow predict$, we check whether the construction process is finished (deep validation), meaning all parameter values were chosen, and we have a valid configuration. And secondly, after making the prediction by models (flat validation). In the later case, if the conditions are violated, the predicted values are discarded and sampled randomly. Since the sampling process implemented in hyper-parameters guarantees to provide valid parameter values, after maximally N mentioned above loops, we derive a new and valid configuration, where N is a maximal depth in the defined search space.

```
class Hyperparameter:
  ...
  def validate(self, values: MutableMapping, recursive: bool) -> bool: pass
  ...
```

Listing 4.5 S.F.R.4 implementation.

4.3 Prediction Process

The next step is an investigation and planning of prediction logic adaptation. In Section 4.1.1 we learned that BRISEv2 provides two learning models: Bayesian tree parzen estimator (TPE) and ridge linear regression. Both models could be used as surrogates within a tiered parameter values sampling however, this process should be generalized.

P.F.R.1 implies the addition of entity, which will encapsulate the prediction process, described in Section 4.2.2. We also make this entity responsible for the forgetting strategy therefore, fulfilling P.F.R.3. Both requirements are not yet fulfilled in BRISEv2, so we must implement them from scratch.

As for P.F.R.2, the current implementation of BRISEv2 already provides some level of model unification using a required interface, which, however, is too coarse-grained and implies binding of three logical steps: data preprocessing, surrogate models creation and optimization with surrogates to predict a next configuration.

The following parts of prediction logic description are dedicated to (1) P.F.R.1 + P.F.R.3 implementation in form of *Predictor* entity and P.F.R.2 in form of decoupled data preprocessing mechanism and the prediction models. Note, the current implementation does not solve the problem of binding surrogate models creation and optimization over them. We postpone the solution for future work. Instead, we implement a simple random search over the surrogate models since as mentioned in Section 2.3.1, given enough evaluations, the random search results becomes comparable to model-based algorithms. We are allowed to do so, since the evaluations over surrogate models are cheap.

4.3.1 Predictor Entity

In addition to previously listed logic during the search space description, a role of predictor is also to decouple learning models from (1) feature tree shaped search space and (2) other core entities such as Configuration. Besides the static search space, the input to predictor is the available in a moment data (evaluated configurations), while the desired output is a configuration. Listing 4.6 provides a pseudo-code of predictor implementation.

To implement the information forgetting mechanism, we refer the idea similar to mentioned in [38] *sliding window*. According to it, the predictor should use specified in a settings number of the latest

configurations as information for surrogate models creation. We modify this logic, allowing user to specify not only a static number, but also a percentage of the latest configurations (line 3), which fulfills the P.F.R.3. Naturally, more exotic approaches may arise such as the statistical analysis to estimate a required configuration number or the other type of meta-learning, but we leave it out of this thesis scope.

The next question is a decoupling of prediction models from the search space structure by means off fulfilling P.F.R.1. As we discussed in Section 4.2.2, to predict a parameter values on each level, the models should be built on only related to this level information. For doing so, after filtering the data (line 6), predictor propagates the prediction from previous level to a current one (line 7), removes the parameters values from other levels and derive a description for the obtained parameters (line 9). Independently, we instantiate a specified in the predictor settings surrogate model for this level and fit it with the related information (lines 11-13). Afterwards, we make a prediction and check if the search space boundaries are not violated (lines 16-17). If either model can not properly fit the data, or the prediction is invalid, we keep the parameter values sampled randomly (lines 18-20).

```
1  class Predictor:
2    def predict(measured_configurations):
3      # P.F.R.3
4      level_configurations = trim_in_window(measured_configurations)
5      prediction = Mapping()
6
7      # Deep validation
8      while not search_space.validate(prediction, recursive=True):
9
10       # Filtering the data
11       level_configurations = filter(search_space.are_siblings(prediction, x), level_configurations)
12
13       # Propagate the prediction
14       randomly_generated = search_space.generate(prediction)
15
16       # Derive the level description
17       full_description = search_space.describe(randomly_generated)
18       level_description = trim_previous_levels(description, prediction)
19
20       # Cut-off data and build models
21       data = trim_accodring_to_description(level_configurations, level_description)
22       model = get_current_level_model()
23       model.build(data, level_description)
24
25       # Level prediction and flat validation
26       if model.is_built():
27         level_prediction = model.predict()
28         if not search_space.validate(level_prediction, recursive=False):
29           level_prediction = randomly_generated
30       else:
31         level_prediction = randomly_generated
32
33     return Configuration(prediction)
```

Listing 4.6 P.F.R.1 + P.F.R.3 implementation pseudo-code.

For the sake of simplicity we omit some minor implementation details and provide the description of (1) data preprocessing and (2) available surrogate models in the Sections below.

4.3.2 Data Preprocessing

The data preprocessing concepts may be split into two complementary parts: an obligatory data encoding and optional data transformation. The first is required to make the underlying model compatible with the provided data. Imagine the parameters values to be a simple strings. Having a surrogate model, which is constructed as the probability densities of parameter values (TPE), one should first derive a numeric data encoding those string parameter values. The second concept is applied on a data, which is already suitable. This is usually done to improve an available surrogate model accuracy by reducing the bias, variance or both. An example of encoding could be a simple indexing of all possible string values. It is performed as a replacement of strings by their indexes during the data preprocessing. On a contrary, for the transformation one may try to add the polynomial features to already available data with an aim to disclose more complex dependencies.

The decision of which encoding to use is often defined by the learning model as a choice among several applicable variants. On contrary, the decision on data transformation is carried out by user and depends on the concrete use-case and experience.

In all reviewed parameter tuning systems, the data preprocessing is implemented only by means of an obligatory for underlying learning models encoding and omitting the possible data transformation. In most cases, it is implemented as a simple label enumeration and not encapsulated at all (as an example, check ConfigSpace's Configuration method *get_array* [1]). Being the most straightforward approach, this encoding may introduce a non-existing patterns in categorical data. For instance, having 3 possible LLH types: genetic algorithm, simulated annealing and evolution strategy, the label encoding will encode such parameter values to numbers 0, 1 and 2 respectively. When such data is passed to the surrogate for learning, some models may interpret it as the GA is closer to SA than to ES within a search space. To prevent this, the other preprocessing should be used for instance, binary encoding.

In any case, the intent of this discussion is to provide the reader with an insight of data preprocessing importance, but the discussion of possible cases and their influence are out of this thesis scope. Here instead we decided to gain a certain level of flexibility by providing a uniformed wrapper for the preprocessing routines implemented in Scikit-learn machine learning framework [86].

We omit the details of wrapper implementation since it is a single object decorator, instantiated with the provided scikit-learn preprocessing unit. The wrapper is executed each time before the actual surrogate performs learning and after making the prediction to inverse the transformation.

To make models and preprocessing units interfaces compatible we store the level information in form of DataFrames — tabular data representation in Pandas framework [2]. In Listing 4.6 line 21 denotes a step of configuration objects transformation to DataFrame, keeping only the current level features.

4.3.3 Prediction Models

As a derivative from predictor implementation, the underlying prediction models should expose a unified interface and behavior. Due to per-level prediction process implementation, the surrogate models are acting on a search space levels without forbidding dependencies. This enables us to use in addition to previously discussed surrogates a vast range of other learning algorithms, for instance, linear regression models. In fact, the previously used in BRISEv2 ridge regression with polynomial features is nothing else then a combination of data preprocessing step from the Section above with the ridge regression model from Scikit-learn framework. Later in this Section, we discuss an implementation of unified wrapper for Scikit-learn linear models.

[1] ConfigSpace documentation `https://automl.github.io/ConfigSpace/master/API-Doc.html`

[2] Pandas Github repository `https://github.com/pandas-dev/pandas`

As a step further, we also add the implementation of multi-armed bandit (MAB) selection strategy proposed in [4]. This is motivated by a great reported performance of the selective hyper-heuristics built on MAB as HLH. It is worth to mention that MAB is applicable only to categorical parameters types.

We also decouple the previously available in BRISEv2 Bayesian TPE from the data preprocessing logic however, no other major changes except refactoring are required. Thus, we do not find a reason for the detained presentation of TPE implementation here.

Scikit-learn Linear Model Wrapper

Scikit-learn is one among the most popular open-source machine learning frameworks. As a consequence of flexible architecture ($H <= T$ framework design patter), Scikit-learn often plays a central role in other products providing implementations for numerous building blocks for machine learning pipelines. These advantages in combination with a comprehensive documentation result into a large and active framework community [1].

All available in Scikit-learn linear regressors are implementing the same interface and usage routines. For instance, before making a prediction, the regression model should be trained on a preprocessed data, providing separate sets of *features* and *labels*. Afterwards, one may use model to make a prediction for unforeseen features and the surrogate will produce a corresponding label according to the learned dependencies. This implies that to find the best parameter combination, one should still solve the optimization problem but with the reduced evaluation cost.

To reuse the available in framework surrogate models we create the wrapper as an object decorator, implementing the required in *Predictor Model* interface. The pseudo-code of this wrapper is presented in Listing 4.7.

During the model creation, we firstly instantiate features and labels preprocessors, and transform the input data (lines 4-6). The underlying process of model building includes also a verification step, which is performed by means of cross-validation: splitting the set of data to k disjoint folds, training k model each time excluding one fold for accuracy verification (line 9). If the potential model accuracy is less than predefined threshold — the model is considered to be not accurate enough therefore, we reject it (line 15), forcing the predictor to use the random parameter values. However, if the model is able to perform well, we train it on an entire dataset and store for further usage (lines 12-13).

Later, for making the prediction (if the model was built successfully) we firstly sample features values from this level uniformly at random (line 20). Afterwards, we transform them using the same preprocessing steps, as we applied during the train creation (line 21). Then, using the regression model, we make a prediction for sampled features and transform those predictions back into original labels (lines 23-24). Finally, we select the best feature by means of predicted labels, transform it and return to *Predictor* (lines 26-27).

[1] Scikit-learn GitHub repository `https://github.com/scikit-learn/scikit-learn`

```
class SklearnModelWrapper(Model):
  def build_model(features, labels, features_description):
    # Data preprocessing
    features_preprocessors, labels_preprocessors = build_preprocessors()
    transformed_features = features_preprocessors.transform(features)
    transformed_labels = labels_preprocessors.transform(labels)

    # Model accuracy validation
    score = cross_validation(model, transformed_features, transformed_labels)
    if score > threshold:
      # Training on all available data
      model.fit(transformed_features, transformed_labels)
      model_is_built = True
    else:
      model_is_built = False
    return model_is_built

   def predict():
    # Solving a reduced optimization problem with help of surrogates
    features = random_sample(features_description)
    features_transformed = features_preprocessors.transform(features)

    labels_predicted_transfored = model.predict(features_transformed)
    labels_predicted = labels_preprocessors.inverse_transform(labels_predicted_transfored)

    prediction_transformed = select_by_labels(features_transformed, labels_predicted)
    prediction = features_preprocessors.inverse_transform(features_transformed_chosen)

    return prediction
```

Listing 4.7 Scikit-learn linear model wrapper pseudo-code.

Multi-Armed Bandit

Originally, the multi-armed bandit (MAB) problem was introduced in [92] and defined as follows: for a given set of choices c_i with unknown stochastic reward values r_i, which are distributed normally with variance v_i, the goal is to maximize the accumulated reward, selecting several times among available choices c_i. The problem obtained its name as an analogy to one-hand slot machines in casino and tackles the well-known exploration vs exploitation dilemma.

In most of the times, MAB is solved by RL approaches, which analyze the already available evidences before performing each next step. In [4] the authors proposed the Upper Confidence Bound algorithm as an intuitive solution: in iteration k, among available choices select one with a maximal UCB value. The UCB for each category is calculated according to Equation (4.1), where first component Q is a quality of category under evaluation and represents the exploitation portion of UCB. The second component estimates the exploration portion balancing the number of time each category was selected. The multiplier C is a balancing coefficient.

$$UCB = Q + C \cdot \sqrt{\frac{2 \log \sum_1^i n_k^i}{n_k}} \tag{4.1}$$

In this work we implement a proposed in [71] Fitness-Rate-Average based MAB (FRAMAB) with two reasons: (1) it is an intuitive and robust approach, (2) according to the benchmarks in [38] it outperforms the other MAB algorithms. In FRAMAB, n_k^i denotes the overall number of categories, while n_k is a

number of times the category under evaluation was selected. The quality estimation Q in FRAMAB is the average improvement, obtained by the underlying category.

As for the balancing coefficient C, the authors in [38] were evaluating a range of values between $10^{-4}...10^{-1}$. The dominance of C values for various problem types were different, therefore we expose it to user for configuration.

In addition to the statically defined C value, we propose a mechanism for C estimation by means of standard deviation among improvement. The motivation for this is as follows: if there exists an uncertainty in category domination the deviation will be high and therefore, encouraging the exploration portion of UCB values. We do not provide a pseudo-code for this model implementation since it straightly repeats the provided above algorithm description.

4.4 Low Level Heuristics

When our HLH is ready to solve an OP, the time comes to provide the tools for solving. A role of LLH in our hyper-heuristic (HH) may play everything from a single heuristic algorithm to the meta-heuristic (MH) or even other HH. As we discussed in Section 2.2.2, nowadays the MH research is referred as the framework growth time. Therefore, we are able not only to reuse a single heuristic but to instantiate a set of underlying heuristics among available in framework. Thus, in this Section we present a review of several toolboxes (meta-heuristic frameworks) with an intent to select the best suited one, implement a facade for framework usage and use the available algorithms as LLHs in our hyper-heuristic.

Before diving into description of available frameworks we briefly outline the LLHs characteristics with respect to which we analyze each framework:

1. **Set of meta-heuristics**, which we will be able to use as LLHs in our HH.
2. **Exposed hyper-parameters**, which are required for LLH tuning. We point it out explicitly, since it happens so that the parameters of an algorithm are exposed not fully.
3. **Set of supported optimization problems**, which will define the applicability of our HH. The wider this set, the more use-cases developed HH is able to tackle. Among them we also separately distinguish the required in our case TSP.
4. **Warm-startup**, which is required to continue the problem solving from previously reached solution. The underlying LLH should not only report finally found solution(s) but also to accept them as the starting points.
5. **Termination criteria**, which is reviewed to control the intermediate results of optimization process by HH. In our system we use the wall-clock time or number of target system evaluations termination to stop the LLH and report the results.

4.4.1 Low Level Heuristics Code Base Selection

We distinguish the following frameworks as the LLH code base: Solid [1], mlrose [2], pyTSP [3], LocalSolver [4], jMetalPy [5] and jMetal [6].

[1] Solid GitHub repository `https://github.com/100/Solid`
[2] mlrose GitHub repository `https://github.com/gkhayes/mlrose`
[3] pyTSP GitHub repository `https://github.com/afourmy/pyTSP`
[4] LocalSolver website `https://localsolver.com`
[5] jMetalPy GitHub repository `https://github.com/jMetal/jMetalPy`
[6] jMetal GitHub repository `https://github.com/jMetal/jMetal`

Solid. A framework for a gradient-free optimization. It comprises wide range of MH skeletons with exposed hyper-parameters: genetic algorithm, evolution algorithm, simulated annealing, particle swarm optimization, tabu search, harmony search and stochastic hill climbing. The support of warm-startup is not provided and requires changes in each algorithm as a consequence of the base class absence. As for the termination criteria, algorithms in this framework support the termination after maximal number of TS evaluations and after reaching the desired quality but not a time based. Once again, to add a new termination criteria, one should modify the core of all algorithms. The framework does not provide the problem instances, nor domain-dependent parts of algorithms therefore, to use this framework one will need to carry out not only a domain-specific framework adaptation but also the problem description.

mlrose. A framework with implementation of various well-known stochastic optimization algorithms such as: naïve and randomized hill climbing, simulated annealing, genetic and mutual-information-maximizing input clustering (MIMIC) algorithms. Each listed solver is implemented as a separate function with exposed set of hyper-parameters. It is possible to control an initial state, which is handy in our case. As for the implemented OPs, the framework comprises a large set of different types: one max, flip-flop, four and six peaks, continuous peaks, knapsack, traveling salesman, n-queens and max-k color optimization problems. The proposed termination criteria are represented only by one criterion controlling the number of TS evaluations. As in the previous framework, here the algorithms are not sharing the same code basis therefore, it may require much effort for their adaptation in general and to introduce a new termination criteria in particular.

pyTSP. A system, specially designed to tackle a traveling salesman optimization problem. Together with visualization techniques, it also provides a wide bunch of different algorithms. Here they are divided into four groups: (1) construction heuristics with the nearest neighbor, the nearest insertion, the farthest insertion and the cheapest insertion algorithms, (2) a linear programming algorithm, (3) simple heuristics among which a pairwise exchange, also known as 2-opt, a node insertion and an edge insertion, and finally (4) a meta-heuristics, represented by the genetic algorithm. As one may expect, the only supported problem type here is the TSP, moreover the representation of problem does not follow a broadly used in research community manner. The other drawbacks of this framework is a partial hard-coding of hyper-parameters and an absence of exposed termination criteria. Also, the construction heuristics by their nature do not expose the possibility to feed them with the initial solutions. However, in some other algorithms the functionality to specify the initial solution is present.

LocalSolver. A commercial optimization tool with free academic license. It is implemented in C++, but the API is exposed to such programming languages as Python, C++, Java and C#. The software implements a local search programming paradigm [7, 8] therefore, the algorithm itself and its parameters are not exposed. It is required from the user to provide a solver-specific problem description. Thanks to a detailed documentation, the desired TSP example could be found among the number of other problem instances. Two possible termination criteria are exposed: a wall-clock time and a number of solver iterations. Also, the framework supports a possibility to set an initial solution for the solver therefore, it looks like a good candidate for the LLH. However, our trial to use the tool showed up that the provided academic license could not be easily used within BRISEv2 containerized architecture. A possible work-around is to deploy a license server on a host machine and force workers to register themselves, but we found this to be an expensive implementation task from the perspective of required effort.

jMetalPy. An open-source meta-heuristic framework for multi- and single-objective optimizations. Among the provided single-objective algorithms one will find genetic algorithm, evolution strategy, local search (hill climber) and simulated annealing. Even if the list of proposed heuristics is not the largest in comparison to other reviewed frameworks, every implemented algorithm exposes its hyper-parameters for tuning. We also found the code well-structured therefore, in case of required changes they could be made with less effort. A functionality to warm-up the solving process by already obtained solutions is available out-of-the-box. The various termination criteria are ready as well, among which based on the wall-clock time and the number of TS evaluations. The list of supported single-objective optimization problems consists of knapsack, traveling salesman and four synthetic problems: one max, sphere, Rastrigin and subset sum.

jMetal. A meta-heuristic framework implemented in Java is an alternative to previously reviewed Python-based jMetalPy. This framework also provides meta-heuristics for multi- and single-objective OP. For later, jMetal developers implemented following algorithms: naïve and covariance matrix adaptation evolution strategies (CMA-ES), genetic, particle swarm (PSO), differential evolution and coral reef optimization algorithms. It is worth to mention that among this list, not all algorithms are universally applicable to wide range of OPs. For instance, CMA-ES, PSO and differential evolution can be applied only to OPs with continuous numeric input such as synthetic mathematical problems. In contrast to implemented in Python brother, jMetal supports only one termination criterion based on number of TS evaluations, and do not support algorithm warm-startup at all. Also, the drawback here is an absence of a proper termination criteria abstraction therefore, the adaptation should affect a framework core.

To sum up our discussion, we aggregate the described characteristics in Table 4.2, which is similar to Table 4.1, presented during the HLH code basis selection. Once again, the characteristics qualities are scored into three ordinal values: poor, average and good with respect to provided functionality and required effort for adaptation.

Table 4.2 Meta-heuristic frameworks characteristics.

Characteristic	Solid	mlrose	pyTSP	LocalSolver	jMetalPy	jMetal
Set of heuristics	Poor	Poor	Good	N/A	Average	Good
Exposed hyper-parameters	Good	Good	Poor	Poor	Good	Good
Provided OPs	Poor	Good	Poor	Good	Average	Average
Warm-startup support	Poor	Good	Poor	Good	Good	Average
Termination criteria	Average	Poor	Poor	Good	Good	Average

Our ultimate goal is not to reach the best performance in provided solution, but to investigate, whether a proposed concept to solve the problem of algorithm selection and generic parameter control simultaneously using the reinforcement learning is able to outperform the baseline performance measures. Thus, while selecting LLH, the quality of provided heuristics and their hyper-parameters are playing a crucial role. There is no perfect candidate among the reviewed systems since every system requires some sort of adaptation. For our experiments we decided to use three LLH: two MHs from Python-based jMetalPy (simulated annealing and evolution strategy) and one from Java-based jMetal (evolution strategy).

4.4.2 Scope of Low Level Heuristics Adaptation

The selected frameworks propose many algorithm implementations. Since the same people are leading the development process, the overall architecture of both frameworks is somehow similar. Nevertheless,

the proposed features are slightly different. For instance, jMetal does not provide time-based termination, nor warming-up the solver by initial solutions. Therefore, we split the adaptation of frameworks on two parts, one is dedicated to jMetalPy and in the other we discuss jMetal.

jMetalPy. During the analysis above, we found out that the provided features are greatly fit our requirements. Even if the lists of implemented MHs and supported OPs are not that wide, we could simply reuse the provided out-of-box implementations. To do so, we implement a framework wrapper (see Listing 4.8), which according to the received task creates a desired instance of the optimization problem type and instantiate the MH solver with provided hyper-parameters (line 3). Later, this wrapper will be called to start a solver execution and report the results in a framework-independent way (line 6). To prevent an expensive problem instances loading during within one run, we cache it in memory (lines 10-11). Also, we cache an expensive I/O introspection calls, which are used to find framework components: algorithms, termination criteria or different algorithm operators such as mutation, selection, crossover, etc. (lines 13-17).

```
1  class JMetalPyWrapper(ILLHWrapper):
2
3    def construct(hyperparameters: Mapping, scenario: Mapping, warm_startup_info: Mapping) -> None:
4      # Constructing meta-heuristics initialization arguments, attach initial solutions
5      pass
6    def run_and_report() -> Mapping:
7      pass
8
9    # Helper methods
10   @lru_cache(maxsize=10, typed=True)
11   def _get_problem(problem_name: str, init_params: HashableMapping) -> Problem:
12     pass
13   @lru_cache(maxsize=10, typed=True)
14   def _get_algorithm_class(mh_name: str) -> jmetal.core.algorithm.Algorithm.__class__:
15     pass
16   @lru_cache(maxsize=10, typed=True)
17   def _get_class_from_module(name: str, module: object) -> Type:
18     pass
```

Listing 4.8 jMetalPy framework wrapper pseudo-code.

While experimenting with the framework, we found several implementation flaws in listed algorithms. The fixes for these bugs were submitted as contributions [1] [2] to implemented open-source framework.

jMetal. On contrary to jMetalPy, this framework is implemented in Java therefore, we can not perform the software instantiating in s straightforward way. Note, BRISEv2 workers are based on Python. There are several libraries which allow to execute a Java code within Python: JPype [3], Py4J [4] or PyJNIus [5]. The usage of one among listed modules enables us to build the same framework wrapper, as we did in previous case. Since currently we are planning to use only one meta-heuristics, the implementation of such wrapper will be unreasonable. Thus, to use a provided in jMetal ES, we pack it into an executable file with exposed parameters and call it from worker script, providing a hyper-parameters settings and warming-up solutions.

[1] jMetalPy PR 1: https://github.com/jMetal/jMetalPy/pull/67
[2] jMetalPy PR 2: https://github.com/jMetal/jMetalPy/pull/80
[3] JPype GitHub repository: https://github.com/jpype-project/jpype/
[4] Py4J GitHub repository: https://github.com/bartdag/py4j
[5] PyJNIus GitHub repository: https://github.com/kivy/pyjnius

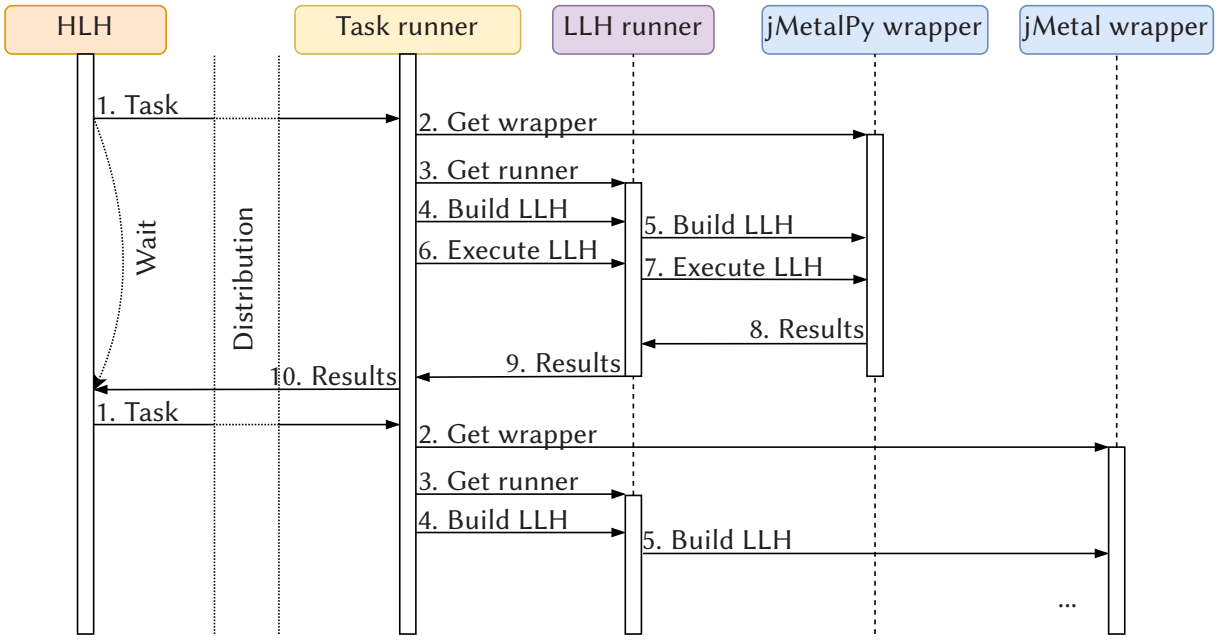


Figure 4.1 The low-level heuristic execution process.

4.4.3 Low Level Heuristic Runner

When the MH wrappers are ready, we use them as different execution strategies of low level heuristic with unified *ILLHWrapper* interface. To operate these wrappers we implement a separate entity — LLH runner, which forwards the construction and execution commands to the wrapper, tracks the state, make general information logging and pass the results after execution. This enables us to easily scale workers horizontally since they are homogeneous and state-less (not taking into account the caching mechanisms). The resulting process of LLH execution from the worker perspective is represented as a sequence diagram in Figure 4.1.

4.5 Conclusion

The performed implementation of proposed in Chapter 3 concept was done reusing the existing frameworks. The hyper-heuristic is mostly based on the modular BRISEv2 framework for parameter tuning. We utilize BRISEv2 prediction models in form of reinforcement learning as a HLH, while several homogeneous workers are carrying out the optimization process using LLH. For the LLH implementation we reuse the existing meta-heuristic frameworks jMetalPy and jMetal. Despite the selected code basis, the proposed approach could be implemented in the majority of parameter tuning systems following SMBO approach however, requiring the adaptation according to our review in Section 4.1.

In this particular case, we were forced to set aside the previously used search space representation and add our own to handle the configuration sparseness issue. The intermediate entity *predictor* was added to decouple the search space shape from the learning-prediction process. It allows us to extend the previously available models with the several others: fitness-rate-average based multi-armed bandits for categorical parameter selection and linear regressors from Scikit-learn framework as surrogates models. We also decoupled data preprocessing step and reused the respective tools from Scikit-learn framework.

We believe the proposed implementation will serve well not only as a hyper-heuristic, but also as old

good parameter tuning framework.

5 Evaluation

The concepts, proposed in Chapter 3, implemented in Chapter 4 of search space representation, prediction process and based on this generalized parameter control approach, selection hyper-heuristic and the hyper-heuristic with parameter control should be broadly evaluated. The experiments may be performed in number of investigation directions, starting from the developed reinforcement learning performance with respect to system configuration and ending with the scalability to different problem sizes.

The structure of this Chapter is as follows. We start the evaluation process with a review of TSP benchmark set in Section 5.1. The following benchmarks could be divided to two main parts. The first is dedicated to the evaluation of developed concept in comparison to the base line and is presented in Section 5.4, while in the second we investigate an influence of the proposed hyper-heuristic with parameter control settings on its performance (Section 5.5). Finally, in Section 5.6 we conclude a discussion of the obtained results.

5.1 Optimization Problem

Through this thesis we are tackling a vehicle routing problem — the traveling salesman OP, which explanation could be found in Section 2.1.1. Nevertheless, as a reminder we repeat its short definition here. We also include the other details, related to the benchmarks.

"Given a set of cities and the distances among them, find the shortest path, which visits all cities". It is a combinatorial OP with a number $n = N!$ of possible solutions. For the benchmarks we use several instances of symmetric TSP (distances $x_i \rightarrow x_j$ and $x_j \rightarrow x_i$ are equal) from a publicly available and broadly used benchmark set TSPLIB95 [1]. The advantage of choosing this benchmark set lays in a broad compatibility of solvers and frameworks with the proposed standardized problem instance description (including used jMetal and jMetalPy). The TSP in this case is defined as a set of city coordinates therefore. Thus, before starting to solve a problem, the distance matrix should be built, calculating Euclidean distances between cities. For more detailed explanation of TSPLIB95 problem instance files refer to [90].

For our benchmarks we select four problem instances from a simpler case harder they are: *kroA100*, *pr439*, *rat783* and *pla7397* of sizes 100, 439, 783 and 7397 cities respectively. The optimal tours for each of these problem instances were previously obtained by exact solvers and reported in aforementioned library. While presenting our evaluation results, we refer to them therefore, we present the optimal solution results in Table 5.1.

Table 5.1 TSP instances optimal tour length.

TSP instance	Optimal tour length
kroA100	21282
pr439	107217
rat783	8806
pla7397	23260728

[1] TSPLIB95 website: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

Note, that the goal of this thesis and the evaluation in particular is not to beat the exact solvers in any case, but to investigate the applicability of proposed generic parameter control concept.

5.2 Environment Setup

To run our experiments we use an enhanced by our approach BRISEv2 and deploy it in Docker containers on a single host machine with following characteristics:

- **Hardware:** Fujitsu ESPRIMO P958 computer with 64GB 2667MHz RAM (16GB * 4 pcs), Intel Core i7-8700 CPU @ 3.2 GHz (6 cores * 2 threads) and Samsung 1TB SSD.
- **Software:** GNU/Linux Fedora 29 host OS and installed docker version 1.13.1.

We deploy 6 homogeneous BRISEv2 workers with LLHs on the same host machine to carry the problem solving process. We run each experiment 9 times for 15 minutes to obtain the statistical data.

5.3 Meta-heuristics Tuning

As we conclude in Section 2.3.4, the goal of parameter control is to reach at least the quality of parameter tuning approaches. Therefore, before running the major set of evaluation experiments, we have to perform a parameter tuning for the underlying LLHs.

5.3.1 Parameter Tuning System Configuration.

As a tuning system, we used our the implemented concept but in the tuner mode. As we described in Section 3.5, to enable the parameter tuning mode, we built a search space based on the singe LLH with its parameters. In our particular case it were three search spaces for each underlying meta-heuristic respectively. We also disabled the solution transfer between each configuration, forcing LLH to use the OP each time from scratch.

For each LLH we run the tuning for 8 hours on 10 deployed worker nodes and three minutes for task evaluation. The underlying prediction mechanism was configured to use TPE with 100% window size. We also disabled the repetition strategy (*repeater* entity), leaving each configuration evaluated only once (with one task). We do so since our preliminary experiments have shown that the variance among evaluations is negligible. As one may expect, since the repetition strategy was disabled, outliers detection was turned off as well.

5.3.2 Target Optimization Problem and Search Space of Parameters.

The role of target optimization problem was played by one of evaluated TSP instances: *rat783*. We selected this instance because, it is a middle size problem, comparing all the used through evaluation OPs.

jMetalPy evolution strategy. This meta-heuristic is implemented in a framework as naïve evolution strategy however, we found an important recombination mechanism missing therefore, the heuristic is performing mostly by means of the mutation operations. As a configuration, this ES implementation requires providing several hyper-parameters. Integer μ (*mu*), which denotes the number of parents in the population, while integer λ (*lambda*) defines the number of offspring. We tune both parameters in ranges $[1..1000]$. Boolean *elitist* defines the selection strategy, which true value enables elitist selection

$(\mu+\lambda)$, while false disables the elitist selection (μ, λ) (more detains in Section 2.2.2). Also, the framework proposes two possible *mutation types* for combinatorial OPs: permutation swap and scramble mutation, which we use for tuning. The respective mutation probability is tuned in range $[0..1]$.

jMetalPy simulated annealing. In this meta-heuristic authors defined the solution neighborhood by means of the same mutation operators, mentioned above. Thus, we use them and the same mutation probability range for tuning the SA. Unfortunately, the authors did not provide other but exponential cooling schedule and did not expose parameters temperature or alpha. This is the reason of such tiny parameter space for this MH.

jMetal evolution strategy. The set of exposed hyper-parameters is almost the same, as we described for the Python-based MH implementation. The only difference that the mutation is represented only by one type, therefore we exclude it from the parameter space but leaving the mutation probability. All the other parameter ranges are the same as for the defined above ES.

5.3.3 Parameter tuning results.

The process of parameter tuning is depicted in Figure 5.1. During the session each MH was probed with at least $1.5k$ configurations.

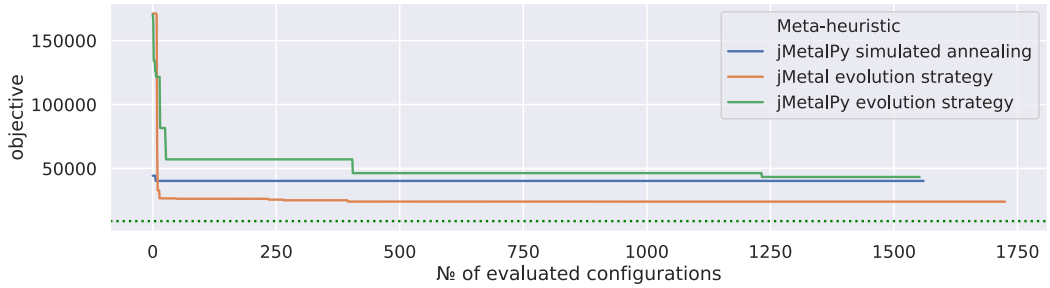


Figure 5.1 The low level heuristics parameter tuning process.

In the figures below we propose a visual analysis of the parameter tuning results. For each meta-heuristic we separately present the numerical and categorical parameters.

The numeric hyper-parameters are showed as scattered points of parameter value (*x-axis*) and the respective objective function result (*y-axis*), obtained for configuration with this parameter value. Although such an isolated approach to analyze data in some cases may be error-prone, still it enough to get a birds-eye view on the existing dependencies. To represent trends among numeric parameter values we draw the regression line (4^{th} degree) in green. At the top and to the right of the graph presented also the axis value densities. Thus, the density on a right side shows which objective values and how often were obtained, changing the underlying parameter, while the density on the top shows which parameter values were selected more often.

As for the categorical parameters, we plot their values as violin plots. It is a combination of box plot with the addition of a kernel density plot on each side. Since in our case, all categorical parameters of underlying algorithms have only two values, each violin plot shows which results of an objective function and how often were obtained. Using colors we depict different value of underlying parameter, while the shape of violin shows an expected result value and its probability. Inside the figure we also

draw three dashed lines. A middle line with long dashes is a median, while lower and upper lines with short dashes show first and third quartiles respectively.

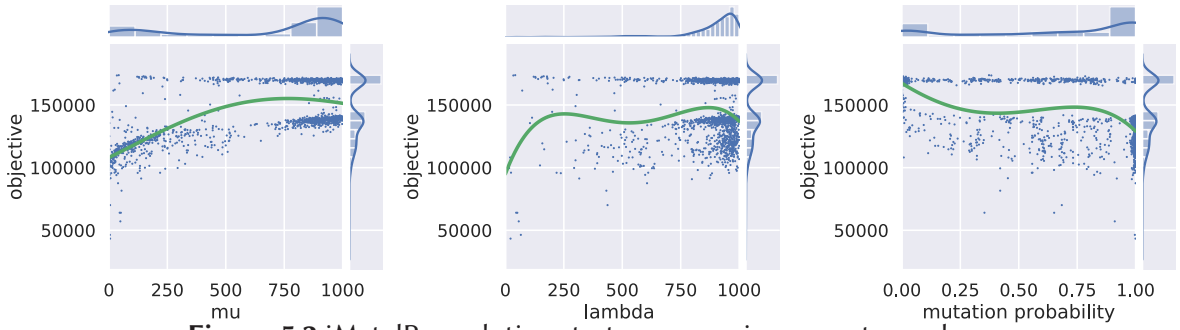


Figure 5.2 jMetalPy evolution strategy numeric parameters values.

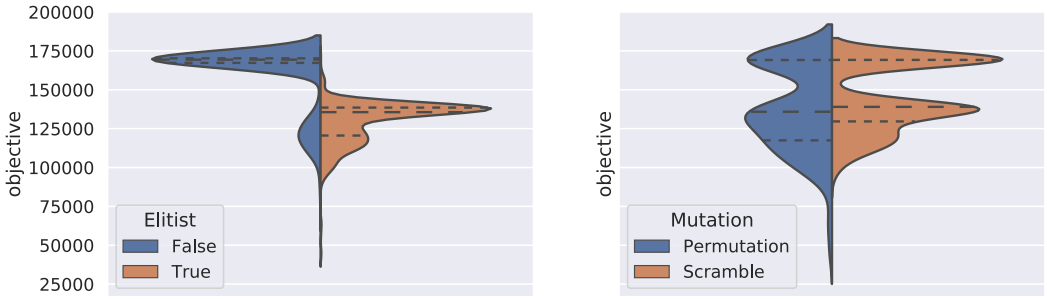
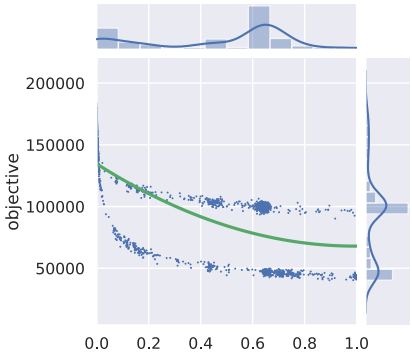


Figure 5.3 jMetalPy evolution strategy categorical parameters values.

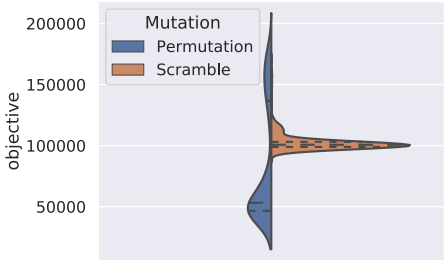
jMetalPy evolution strategy parameters. Looking on the Figure 5.2, one will see an explicit dependency between the number of parents (Figure 5.2 parameter *mu*) and the objective function: less amount of parents are tended to produce the better results. However, the dependency is such clearly observable for the number of offspring (Figure 5.2 parameter *lambda*). We may see, that a high number of offspring does not tend to provide good results, but the number of performed estimations for low *lambda* is not enough to be strongly ensured that this value is better. Yet, even with small amount of observations we may make a guess that low *lambda* is a good parameter choice. With respect to the mutation probability, it may be observed, that the higher mutation rates tend to produce a better results.

As for the categorical parameters, one may see a strong bias towards bad results when using non-elitist algorithm version (Figure 5.3 parameter *elitist*). When concerning the mutation type, the dominance is not an obvious, but permutation version of mutation is slightly outperforms scramble type (Figure 5.3 parameter *mutation*).

jMetalPy simulated annealing parameters. This heuristic were tuned by means of only two parameters: categorical mutation type, which results are presented in Figure 5.4b and numerical mutation probability with graphs in Figure 5.4a. One may see a strong dominance of permutation mutation type, while scramble produce an average but stable results. The mutation probability trends are also clear: higher parameter values produce better results. Indeed, the dependency on mutation probability is obvious, since the underlying algorithm is performing the search space traversal by means of solution mutation. The two lines of results, that could be viewed on the Figure 5.4a are correlated with the mutation type: lower corresponds to usage of permutation, while upper to scramble mutation.



a) Mutation probability.



b) Mutation type.

Figure 5.4 jMetalPy simulated annealing parameters.

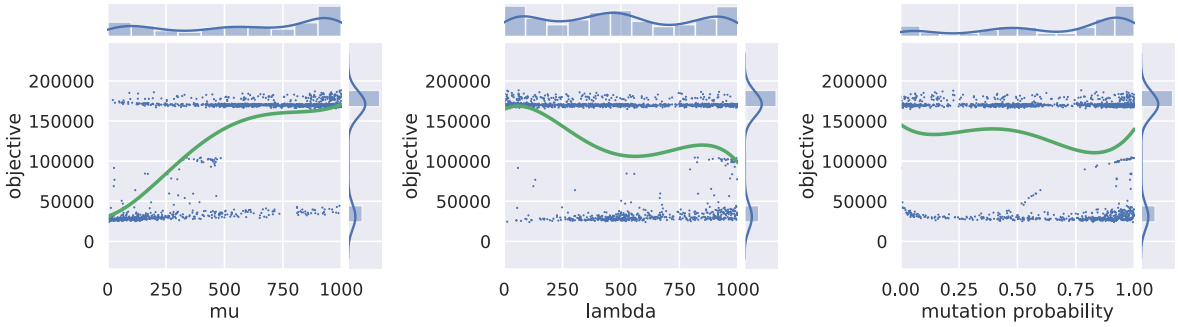


Figure 5.5 jMetal evolution strategy numeric parameters values.

jMetal evolution strategy parameters. The final heuristic under investigation is the Java-based implementation of viewed above ES. Even if at the first glance the regression lines are not looking the same, the overall trends are similar: lower values of *mu* parameter result in better objective, while the mutation probability should be kept high. On contrary to Python-based ES, here the middle-range values of parameter *lambda* produce the best results. It may be explained by the fact of performance straggling in Python-based version: with large offspring number, the computational effort, required to accomplish the iteration increases, while Java-based version could handle it. A dominance of elitist version of algorithm is non-obvious, but this could be seen from a distribution first quartile.

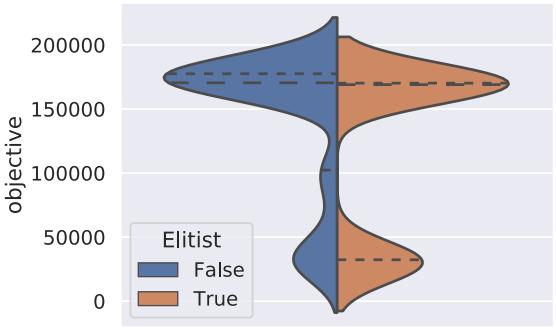


Figure 5.6 jMetal ES elitist parameter.

We collected the best performing configurations of each meta-heuristic and presented them in Table 5.2. We also highlight here the default parameter values, which were selected with motivation of being in the middle of the values ranges.

Table 5.2 Static hyper-parameters of low-level meta-heuristics.

Hyper-parameter	Default value	Tuned value	Estimated range
jMetalPy evolution strategy			
μ	500	5	[1..1000]
λ	500	22	[1..1000]
elitist	False	True	True, False
mutation type	Permutation	Permutation	Permutation, Scramble
mutation probability	0.5	0.99	[0..1]
jMetalPy simulated annealing			
mutation type	Permutation	Permutation	Permutation, Scramble
mutation probability	0.5	0.89	[0..1]
jMetal evolution strategy			
μ	500	5	[1..1000]
λ	500	605	[1..1000]
elitist	False	True	True, False
mutation probability	0.5	0.99	[0..1]

5.4 Concept Evaluation

5.4.1 Evaluation Plan

To evaluate the performance of developed approach we firstly need to define the base line. In most cases it is the single meta-heuristics, which are solving the OP using static hyper-parameters. However, to evaluate the parameter control feature we must make a closer look on the performance of each separate heuristic with static and dynamic hyper-parameters. For selection hyper-heuristic analysis we compare the performances of all underlying MHs running separately and together within a hyper-heuristic. Note, in this case the hyper-parameters are statically defined. And last, but not least, to evaluate a selection hyper-heuristic with enabled parameter control we compare it to separately running underlying meta-heuristics with parameter control and to selection hyper-heuristic.

In order to organize the evaluation plan, we distinguish two stages. At the first stage the LLH selection occurs, while at the second one we chose hyper-parameters for the selected LLH. At each stage we may use different prediction approaches, which description could be found in Section 4.3.3. To select the LLH, apart from random and static selection we also use FRAMAB (see Section 4.3.3) and Bayesian ridge regression model implementation from Scikit-learn framework (see Section 4.3.3). Note, for the Bayesian ridge regression model we use a default parameters, which could be found in the framework documentation [1]. To select the hyper-parameters for LLHs, apart from static default and tuned variants we also use random selection, available in BRISEv2 TPE and the mentioned above Bayesian ridge. The set of used techniques is presented in the Table 5.3.

Using this table, we now could pick a prediction technique to form a desired system configuration. For instance, mentioned above baseline could be encoded into configurations starting from *4.1.1* for the evolution strategy from jMetalPy framework, running with default hyper-parameters and ending with *4.3.2* for evolution strategy from jMetal framework, running with tuned beforehand parameters.

Our benchmark plan for the concept evaluation looks as a set of following experiment groups:

- **Meta-heuristics (MH).** The baseline. We evaluate each used meta-heuristic separately with

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html`

Table 5.3 Prediction techniques used for the concept evaluation.

LLH selection	LLH parameters selection
1. Random	**1.** Default
2. Multi-armed bandit	**2.** Tuned beforehand
3. Bayesian ridge regression	**3.** Random
4.1. Static jMetalPy.ES	**4.** Tree Parzen Estimator (TPE)
4.2. Static jMetalPy.SA	**5.** Bayesian ridge regression (BRR)
4.3. Static jMetal.ES	

default and tuned hyper-parameters: *4.1.1* and *4.1.2* for jMetalPy evolution strategy; *4.2.1* and *4.2.2* for jMetalPy simulated annealing; *4.3.1* and *4.3.2* for jMetal evolution strategy respectively.

- **Meta-heuristics with parameter control (MH-PC).** The set of experiments dedicated to verify an impact of the generic parameter control on meta-heuristics performance. A selected set of experiments looks as follows: *4.1.3, 4.2.3, 4.3.3* to investigate the influence of random parameter allocation; *4.1.4, 4.2.4, 4.3.4* to check TPE-based parameter control and *4.1.5, 4.2.5, 4.3.5* to probe Bayesian-ridge-based parameter control.

- **Selection hyper-heuristic with static parameters (HH-SP).** These benchmarks are dedicated to an investigation of the implemented on-line selection HH performance. It implies the LLHs usage with static parameters therefore, we evaluate HH-SP performance with default and tuned beforehand LLH parameters. Experiment codes are following: *2.1, 2.2* for FRAMAB-based HH-SP and *3.1, 3.2* for Bayesian-ridge-based HH-SP.

- **Selection hyper-Heuristic with parameter control in LLH (HH-PC).** This is a final set of benchmarks for concept evaluation. By this we evaluate an influence of simultaneous on-line LLH selection and parameter control on system performance. The respective experiment set is following: *1.3, 2.4, 2.5, 3.4, 3.5.*

The aggregated concept benchmark plan is presented in Table 5.4.

Table 5.4 Concept benchmark plan.

Experiment group	Related codes
MH	4.1.1., 4.2.1, 4.3.1
	4.1.2, 4.2.2, 4.3.2
MH-PC	4.1.3, 4.2.3, 4.3.3
	4.1.4, 4.2.4, 4.3.4
	4.1.5, 4.2.5, 4.3.5
HH-SP	1.1, 1.2
	2.1, 2.2
	3.1, 3.2
HH-PC	1.3
	2.4, 2.5
	3.4, 3.5

5.4.2 Concept Evaluation Results

Baseline Evaluation

As we discussed previously, our results comparison should be done against the defined baseline. Therefore, this section is dedicated to review of the meta-heuristics performance out-of-the-box on different problem sizes and parameter settings. For visibility reasons we plot the intermediate and the final performance evidences for each problem instance separately, since they naturally imply different result ranges.

Since we are tackling a set of TSP instances, which were previously solved by other exact solvers, we also present an optimal solution, available for each instance as a green dashed line.

kroA100 TSP instance. The TSP for 100 cities is a relatively small problem instance. Therefore, all underlying MHs reach a local optimum after first few minutes of the run and stuck there till the end, making a relatively small moves (Figure 5.7). Note, the bold line is a statistical mean of all 9 experiment runs, while a shadow around it is a confidence interval. One may observe how the parameter tuning affects different MHs: in some the difference is dramatic (ES), while others are almost not affected (Section 5.4.2).

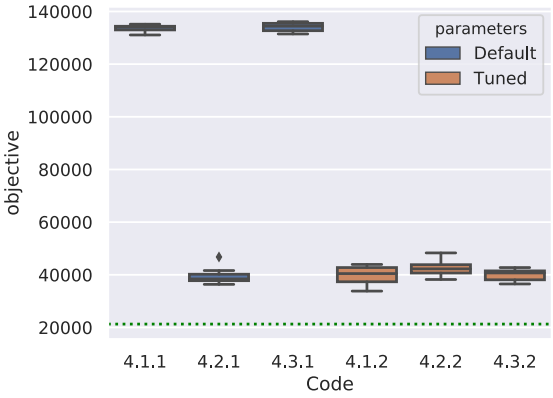


Figure 5.8 Final results of meta-heuristics with static parameters on kroA100.

An observation of a worse SA results with tuned parameters, in contrast to default values is explained by the fact that for algorithm tuning we used different problem instance (rat783). It only confirms a motivation of the parameter control approaches: tuning is not problem-instance-universal technique.

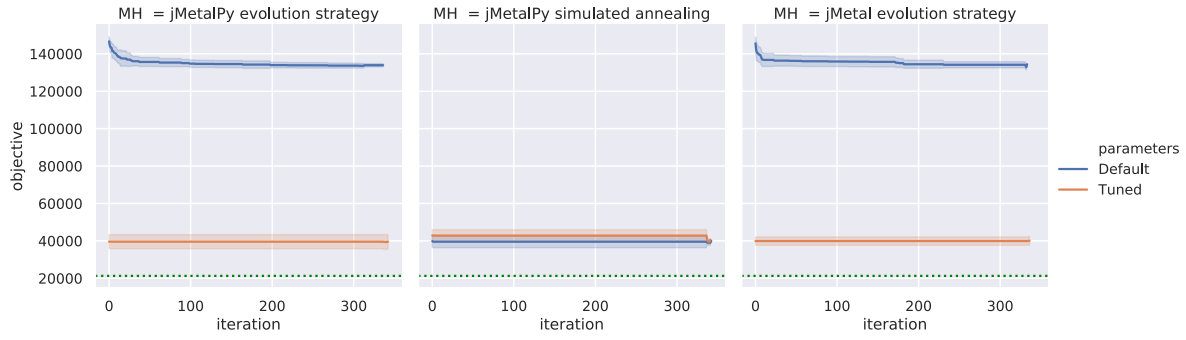


Figure 5.7 Intermediate results of meta-heuristics with static parameters on kroA100.

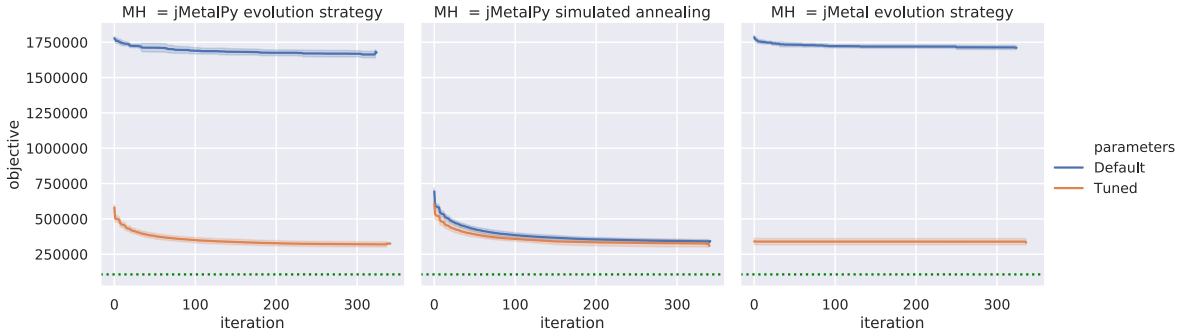


Figure 5.9 Intermediate results of meta-heuristics with static parameters on pr439.

pr439 TSP instance. The next problem instance comprises a 4 hundred cities. Since the number of possible solutions increased, the required time for MHs to settle in a local optima is increased as well. We may see a similar to the previous experiment set trends: produced by SA results are almost not affected by the parameter tuning, however with tuned parameters the performance is slightly better. From the other side, Python version of tuned ES requires more time to converge in local optima, while Java-based reaches it after a couple of first iterations (Figure 5.9).

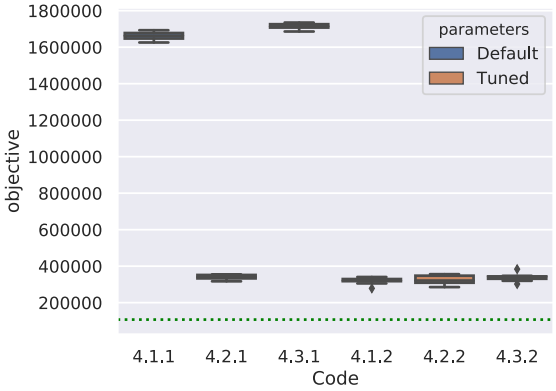


Figure 5.10 Final results of meta-heuristics with static parameters on pr439.

The final result quality of underlying MHs is the same as on previously reviewed problem instance: MHs with tuned parameters (an SA with default) produce solutions of a similar quality (Section 5.4.2).

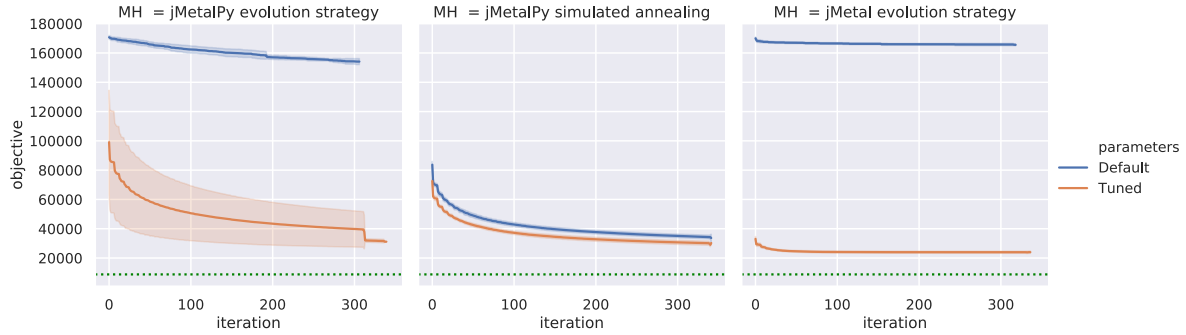


Figure 5.11 Intermediate results of meta-heuristics with static parameters on rat783.

rat783 TSP instance. This is an average size problem among reviewed in this thesis. The behavior of solvers in this case changes slightly. For instance, ES from jMetal framework with default parameters is unable to solve a problem however, while using an optimized hyper-parameters it quickly reaches a local optima (after 50 iterations) with the best produced results (Figure 5.11). Analyzing performance evidences of the other heuristics (Python-based), we may conclude that they did not reach a local optima in a given 15 minutes therefore, their final results are slightly worse (Section 5.4.2). Taking into account previous problem instances we may guess that given enough time they will reach the results of jMetal ES. Note the decreased stability of jMetalPy ES reflected by a large confidence interval.

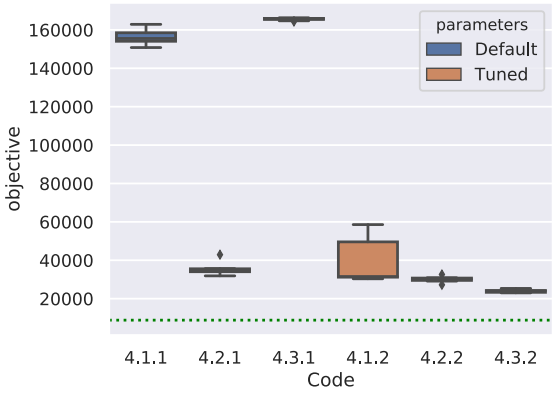


Figure 5.12 Final results of meta-heuristics with static parameters on rat783.

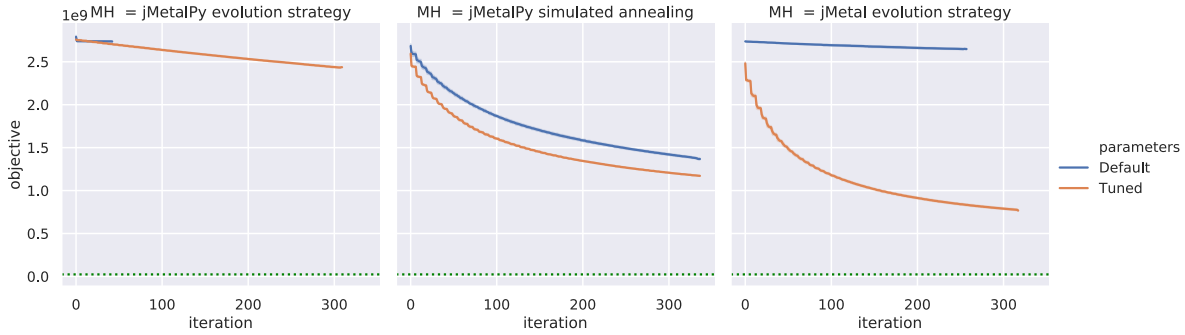


Figure 5.13 Intermediate results of meta-heuristics with static parameters on pla7397.

pla7397 TSP instance. The largest investigated here TSP instance for $7.4k$ cities is however, referred as a middle-size OP in used TSPLIB95. Here the performance evidences changed the most therefore, we discuss each MH separately.

Python-based version of ES provide the worst results with both default and tuned parameters. Note the amount of performed iterations by this MH in a given 15 minutes — less than 100. It is affected by a several reasons. Firstly, the amount of time required to perform an internal iteration increased dramatically. Thus, even with specified 15 seconds for one task run, actually it requires much more (up to 1 minute) to accomplish a task. We have a several guesses, what may cause such a behavior. Firstly, it is a general Python performance issues, in most of the times caused by usage of global interpreter lock (GIL), which explanation is out of this thesis scope. Secondly, it may be caused by the algorithms code basis implementation in jMetalPy framework. To implement a generic termination criteria (and some other features) the authors utilized a push-observer design pattern [6] according to which the underlying algorithm (ES in our case) triggers its observers after each iteration. Therefore, stopping criteria may be evaluated only after finishing an iteration, which in case running ES with TSP for $7.4k$ cities, may take a while. We observed the ES algorithm termination after the first internal iteration, which causes a poor solution quality improvement. Naturally, there is also an overhead for the results sending through the network, but as Java-based ES with default parameter shows, this causes decrease only in 100 configurations. We also eliminate the possible issue in a required time for problem loading (building distance matrix), since the worker node does it only once and stores its cached version. In any case, this issue requires deeper investigation, that we postpone to the future work. Running jMetalPy ES with tuned parameters fixes the issue with task delays and therefore, results in higher number of external iterations, but the solution improvement are still weak (see left picture on the Figure 5.13).

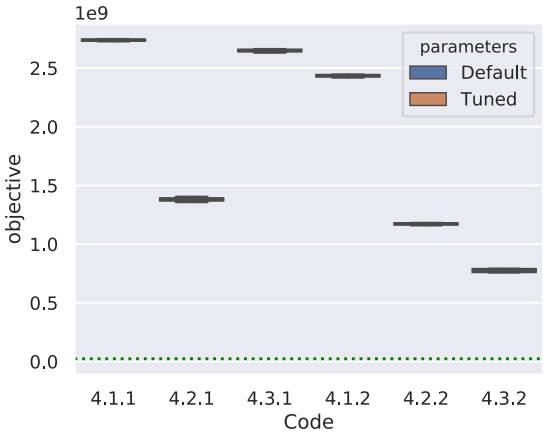


Figure 5.14 Final results of meta-heuristics with static parameters on pla7397.

As in the previous case, the jMetalPy-based simulated annealing produce a good quality improvement, least depending on the hyper-parameter values. A resulting progress curve, presented in a central picture of Figure 5.13 shows that SA requires more time to converge that was provided and is still far

from its potential local optima.

The final evidences obtained from the jMetalPy ES show it dominance in intermediate and final results (see Figure 5.13 and Section 5.4.2 respectively). In this example we observe a dramatic impact of the solver parameter tuning. Using default configuration, ES is struggling in making improvements. Our guess here is the same as for the Python-based version — the number of internal iterations is extremely low a good search space traversal, but they are finishing much faster.

Discussion. The observed results of meta-heuristics execution confirm the algorithm parameter setting problem importance, discussed in Section 2.3. An effect of proper parameter selection is different among algorithms. In our case, the performance of two out of three solvers are highly dependent on the hyper-parameter settings. Thus, an application of our generic parameter control approach to these algorithms is rather intriguing and may partially reveal the overall methodology benefits.

From the other side, we also observe the only one algorithm domination among the others with static parameters. See how all MHs were solving each TSP instance with default parameters. In each case SA outperforms two other ES. The usage of all three MHs in a selection hyper-heuristic with static (default) hyper-parameters will reveal the implemented selection HH applicability. In this case we expect to observe the results close to provided by pure SA. The other case — the application of MHs to the biggest TSP instance with tuned hyper-parameters. Here Java-based ES is the best. Thus, we expect to observe such a behavior of selection hyper-heuristic. Also, it should be rather interesting to see the impact of Python ES struggling with default parameters on HH.

Generic Parameter Control

As we discussed in Section 2.3.3, the goal of parameter tuning lays in adaptive changing of underlying algorithm parameters with to optimize some performance measurement. In our case, we apply the proposed in Section 3.5 methodology to set the parameters of anytime algorithms in a runtime. Here is a brief reminder: at each RL step HLH is analyzing the performance of solver with previous configurations to choose the parameters values, which hopefully lead to the higher solution quality improvements. Afterwards, we run the solver with sampled parameters for a predefined time (15 seconds) to get new evidences.

Here we compare the performance of algorithms with statically defined default and tuned hyper-parameters to dynamically changing parameter values by means of RL control. As previously, we perform the comparison for each problem instance separately.

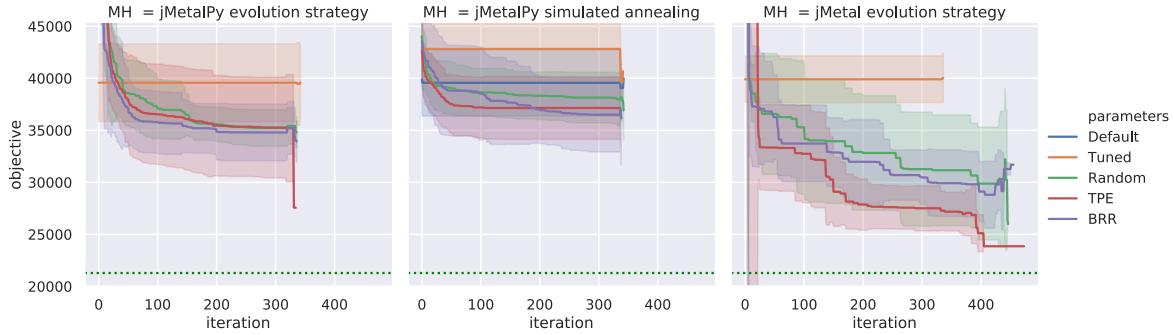


Figure 5.15 Intermediate results of meta-heuristics with parameter control on kroA100.

kroA100 TSP instance. Comparing to the baseline, parameter control in a small problem instance was able to reach and even outperform the results of MHs on static parameters after first 50 iterations (Figure 5.15). We may observe, that even randomly changing the parameters of heuristics (however, this random sample still emits valid configurations) in a runtime results in better solutions comparing with static parameters. It is caused by the changes in a neighborhood definition (mutation type) and traversal process (mutation probability). In most cases, given enough time the learning-based parameter assigning outperforms random allocation (Figure 5.16).

Note the amount of configurations (iterations) performed by jMetal ES, which could be seen in Figure 5.15. According to our plan, a given time for MH run is 15 minutes, 15 seconds for running one configuration on 6 available workers. Thus, in the most optimistic case, the number of iterations should be $\frac{15 \cdot 60 \cdot 6}{15} = 360$ but, we observe even more than 400. After an investigation, we came to conclusion that it is caused by an implementation flaw an insight of which is following. jMetal MHs provide only iteration-number-based termination criterion, which is not encapsulated how it is done in jMetalPy. For our needs we added also a time-based but did not remove the previously existing. For the iteration counter used a regular integer number, which we set up to its maximal value when using a time-based criterion. Given a specific 'light' algorithm configuration (low μ, λ and mutation probability), with this OP MH is able to reach the maximal number of iteration in less than 15 seconds therefore, terminating early and triggering a new parameter control iteration. Certainly, it is our implementation bug, fix of which we postpone to the future work.

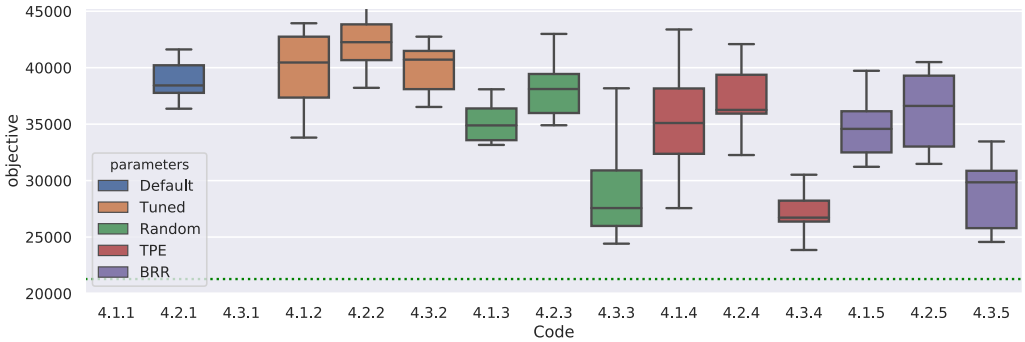


Figure 5.16 Final results of meta-heuristics with parameter control on kroA100.

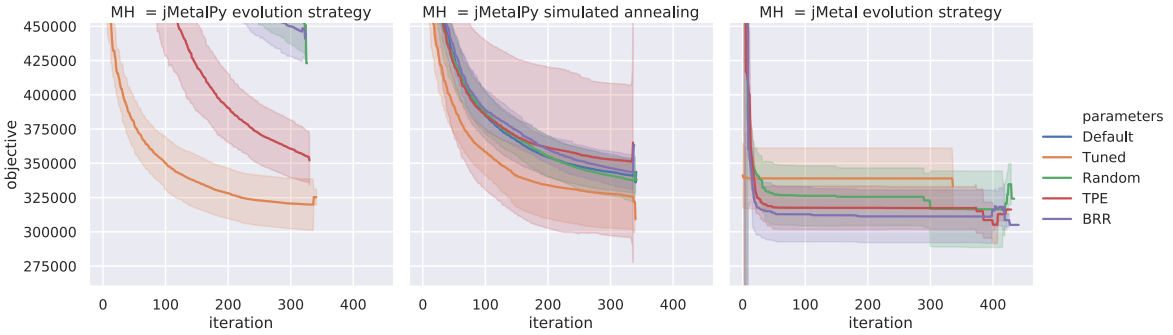


Figure 5.17 Intermediate results of meta-heuristics with parameter control on pr439.

pr439 TSP instance. When parameter control is applied to a larger problem, it is starting to require more time for finding a better quality configurations for jMetalPy ES. In particular, only the TPE-based approach was able to reach tuned parameters results quality. The intermediate performance was reduced, since model-based tuning required more knowledge to find a good-performing settings. However, looking on the progress curves in Figure 5.17, RL TPE-based control technique will probably be able to reach and outperform tuned beforehand parameters.

The case with jMetalPy SA shows (1) reduction in the final results qualities with controlled parameters (Figure 5.18) and moreover (2) unstable behavior of MH with TPE-based control (Figure 5.17). As we concluded during the baseline results analysis, the parameter settings in this MH does not dramatically affect its performance therefore, the results of random-based parameter allocation are similar to model-based approaches.

On contrary, applying generic parameter control to jMetal ES MH leads to a better final results, comparing with static tuned parameters(Figure 5.18). Note, as for previous problem instance, even a random-based MH parameters allocation outperforms statically defined but, the required time to converge and a result floating increased as well.

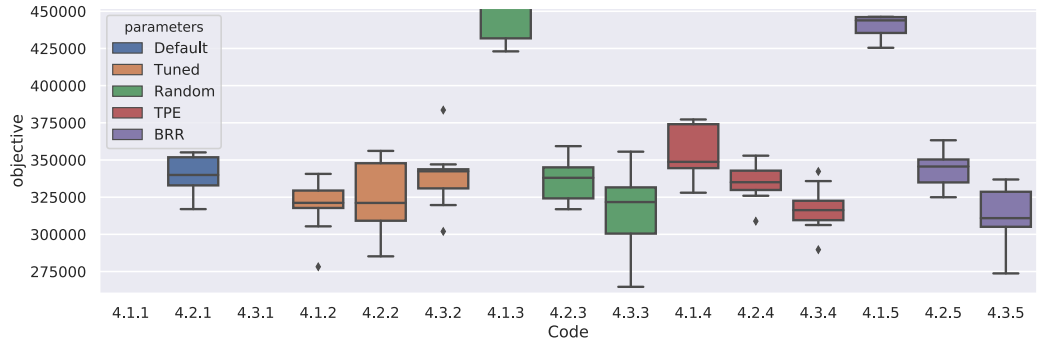


Figure 5.18 Final results of meta-heuristics with parameter control on pr439.

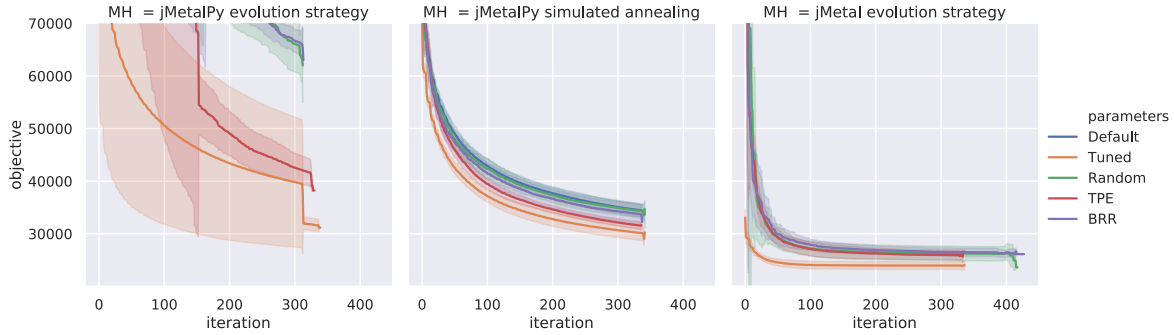


Figure 5.19 Intermediate results of meta-heuristics with parameter control on rat783.

rat783 TSP instance. This is a problem instance, which was used to set the meta-heuristic parameters by means of off-line tuning. In this case, the parameter control in jMetalPy SA and jMetal ES did not manage to outperform the tuned parameters but only nearly reached their quality. All approaches, including the random selection resulted in a similar intermediate performance (Figure 5.19).

For jMetalPy ES MH, only TPE-based parameter control produced good but rather unstable final performance, comparing to tuned hyper-parameters. The other (random- and BRR-based) approaches did not manage to select the parameters, which would perform well enough and therefore were left out of presented quality ranges in Figure 5.20.

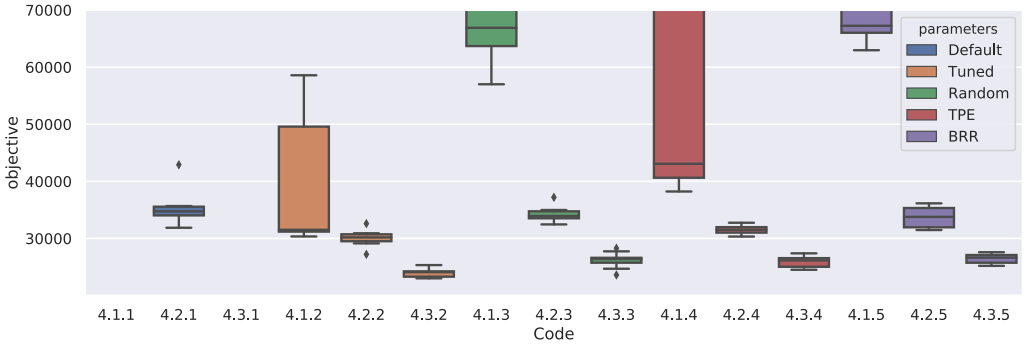


Figure 5.20 Final results of meta-heuristics with parameter control on rat783.

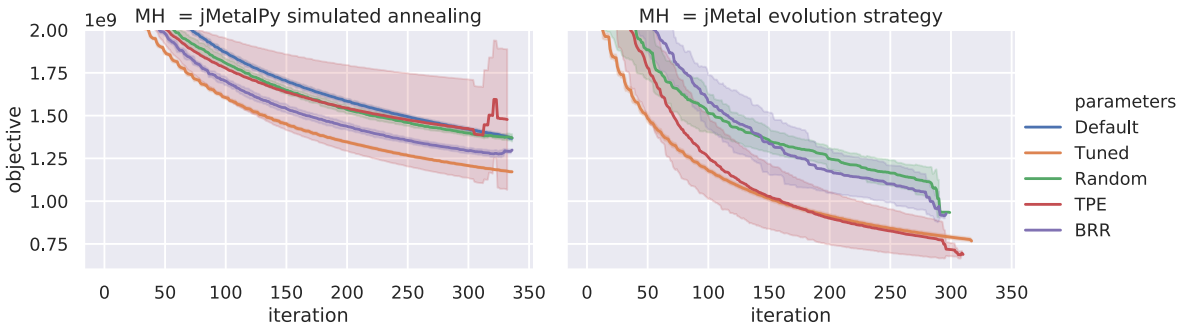


Figure 5.21 Intermediate results of meta-heuristics with parameter control on pla7397.

pla7397 TSP instance. For the final problem instance we omit the parameter control results of jMetalPy ES since it did not manage to perform even slightest improvement in comparison to the default parameter values, presented in a baseline description. It is caused by a fact that in early stages our approach acts as a random search, since not enough evidences were obtained to build models for prediction. Thus, is case of jMetalPy ES, the MH was running with badly performing configurations and as we explained in Section 5.4.2, did not manage to perform enough iterations to improve solution quality and was left out of this instance discussion. To resolve this issue, the time-based task termination methodology should be properly implemented in MH wrappers, but we postpone it for the future work.

As in previous cases, the least parameter-settings-sensitive jMetalPy SA shows an ability to perform well with any approach of the parameter settings (Figure 5.21). However, neither among them outperform tuned beforehand algorithm configuration in final results quality (Figure 5.22).

As for jMetal ES, model-based approaches are outperforming the randomized parameter values allocation. Moreover, TPE model outperforms even the results of algorithm with tuned beforehand hyper-parameters.

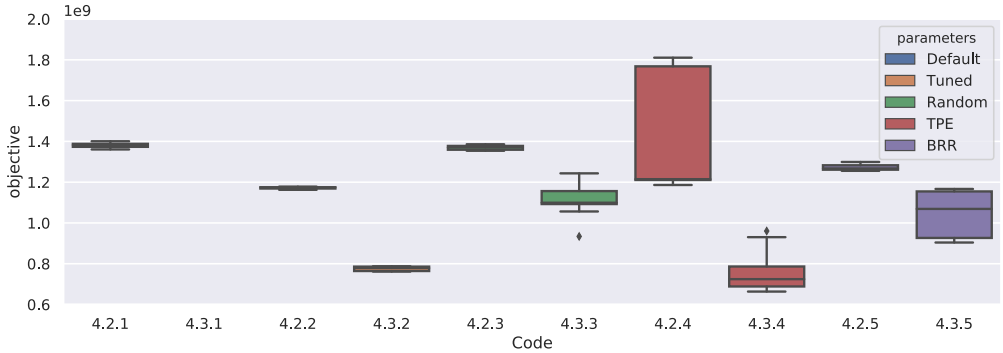


Figure 5.22 Final results of meta-heuristics with parameter control on pla7397.

Discussion. In general, the review of meta-heuristic performance on different problem instances showed that the proposed generic parameter control approach is applicable. It is capable to improve a final algorithm performance in comparison to statically selected default parameters. Moreover, in some cases (all MHs with kroA100, jMetal ES with pr439 and pla7397) the algorithm showed better results than with tuned parameters.

Taking into account the results with random parameter allocation we conclude that learning mechanisms should and must be improved further by means of different surrogate models usage or proper optimization over surrogates. Leaving the improvement steps to future work we conclude that the proposed generic parameter control concept is able to produce better results while solving an unforeseen problem on-line.

Selection Hyper-Heuristic with Static LLH Parameters

The second mode of developed approach and at the same time a main goal of the thesis is a process of dynamic algorithm selection. It is implemented in form of described in Section 3.5 reinforcement learning-based on-line selection hyper-heuristic. In this part of evaluation we combine three available meta-heuristics with static parameter (default and tuned) into a single selection hyper-heuristic. Three approaches to select the LLH were investigated, in combination to two possible parameters settings it results in 6 combinations for each problem instance.

We present the process of problem solving in two forms. Firstly, we present the process of problem solving, distinguishing selected at each iteration LLH and the results, which it gave. For doing it, we selected only the first repetition (out of 9 available), since presenting all repetitions will not be possible to understand. Nevertheless, we present the final results of all runs in form of box-plots, comparing them to the underlying LLH performance. LLHs solely and combined into the selection hyper-heuristics are configured to use a static hyper-parameter values (HH-SP). On the left side we present final performance with the default parameter values, while on the right site — the tuned beforehand values.

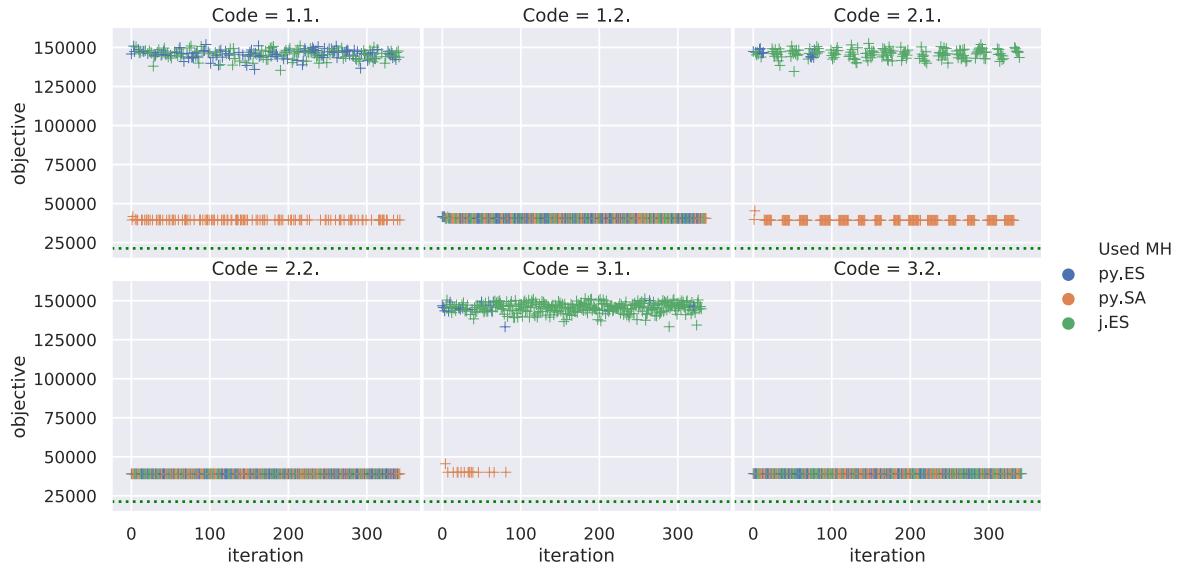


Figure 5.23 Intermediate performance of on-line selection hyper-heuristic with static hyper-parameters on kroA100 (single experiment).

kroA100 TSP instance. The intermediate results with random LLH allocation show us, as expected, unbiased uniform selection of LLH (codes 1.1, 1.2). We observe the same behavior with all HH-SP approaches, while they utilize LLHs with tuned parameter values (codes 1.2, 2.2, 3.2), which is motivated by an equal performance of all LLHs on this TSP instance. Thus, the point of interest here are LLHs with default parameter values (codes 2.1, 3.1 in Figure 5.23). When FRAMAB is used as HLH, a preference in j.ES and py.SA appears. We observe a repetitive pattern in LLH allocation, when FRAMAB reaches a local optimum (code 2.1). It is caused by a deterministic essence of the algorithm. Being in a local optimum, FRAMAB's exploration mechanism fully guides a selection. Due to the usage of a similar time-based LLH termination, all workers are starting the next round in bunches. Thus, when a new round starts, FRAMAB operates on static information and allocates all next configurations with the same LLH. From the other perspective, when the process reaches a point, where the advantage changes towards another LLH, FRAMAB behaves inertly. One may argue this will cause a struggle in a performance, which is rather a logical conclusion. It requires a further investigation, which we are forced to postpone for the future work. In case of BRR usage, the bias dominates in j.ES however. According to presented in Figure 5.24 final results statistics for this problem instance, given at least one dominating LLH (default parameters values case), even a random LLH selection could utilize to obtain a good final results.

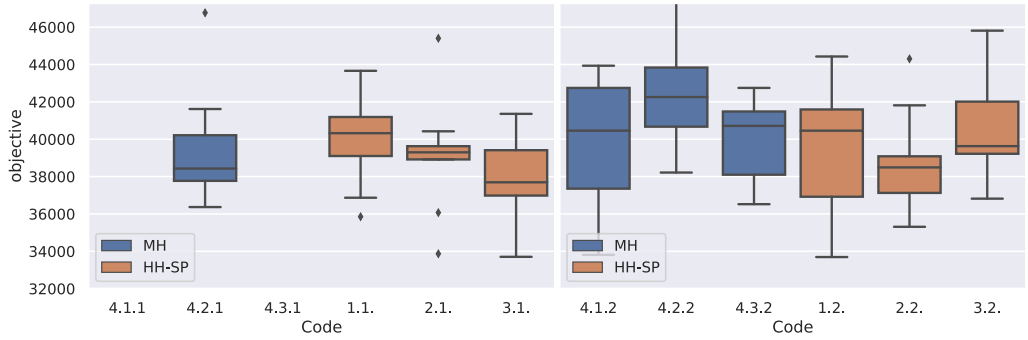


Figure 5.24 Final results of on-line selection hyper-heuristic with static hyper-parameters on kroA100 (statistic of 9 runs).

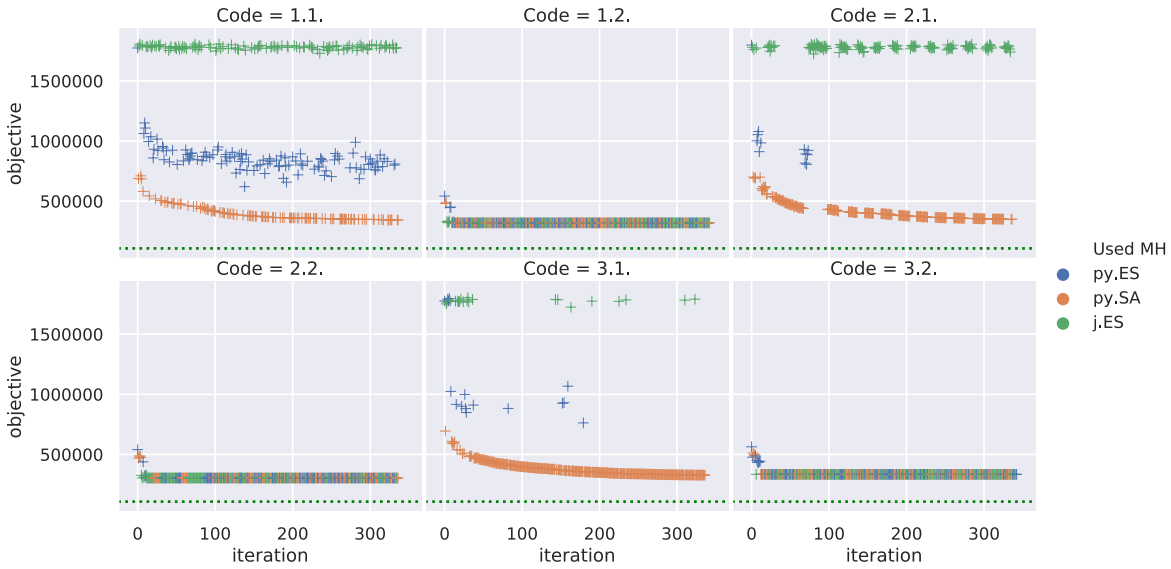


Figure 5.25 Intermediate performance of on-line selection hyper-heuristic with static hyper-parameters on pr439 (single experiment).

pr439 TSP instance. Evaluating the 439 cities instance solving process, we observe a similar behavior while using tuned hyper-parameters therefore, we do not draw our attention here: every LLH reach a local optimum in a couple of first iterations and no dominant LLH may be distinguished.

On contrary, when using the default parameter values a bias towards py.SA occurs, since it showed the best results improvement during the baseline evaluation. After reaching a local optima, FRAMAB-based HH-SP (code 2.1) frequently allocates two heuristics: py.SA and j.ES, while BRR-based (code 3.1) continues to allocate mostly py.SA (Figure 5.25). Comparing to the previous problem instance, BRR dramatically changed its behavior to utilize in the most cases the best performing py.SA. Thus, we can conclude the BRR HLH provides more exploitation capabilities, when compared to FRAMAB HLH.

The final result quality of all HH-SP on this problem instance are roughly similar to provided by the best performing py.SA with default parameters. Once again, all tuned LLHs separately provide a similar final result quality therefore, their combination in HH-SP leads to the same final quality (Figure 5.26).

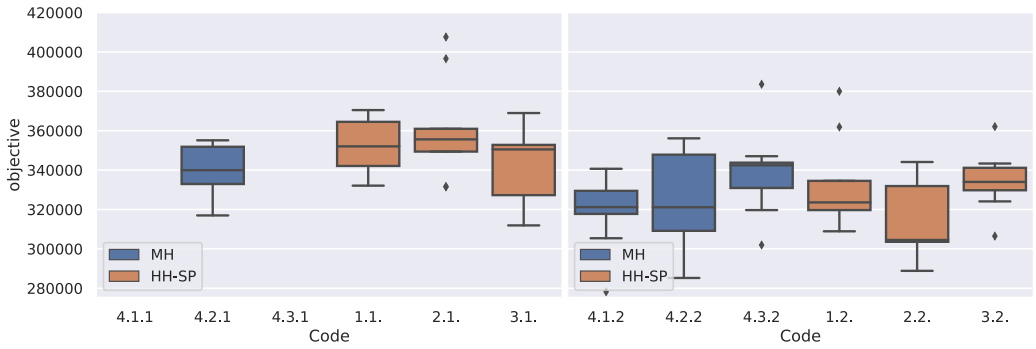


Figure 5.26 Final results of on-line selection hyper-heuristic with static hyper-parameters on pr439 (statistic of 9 runs).

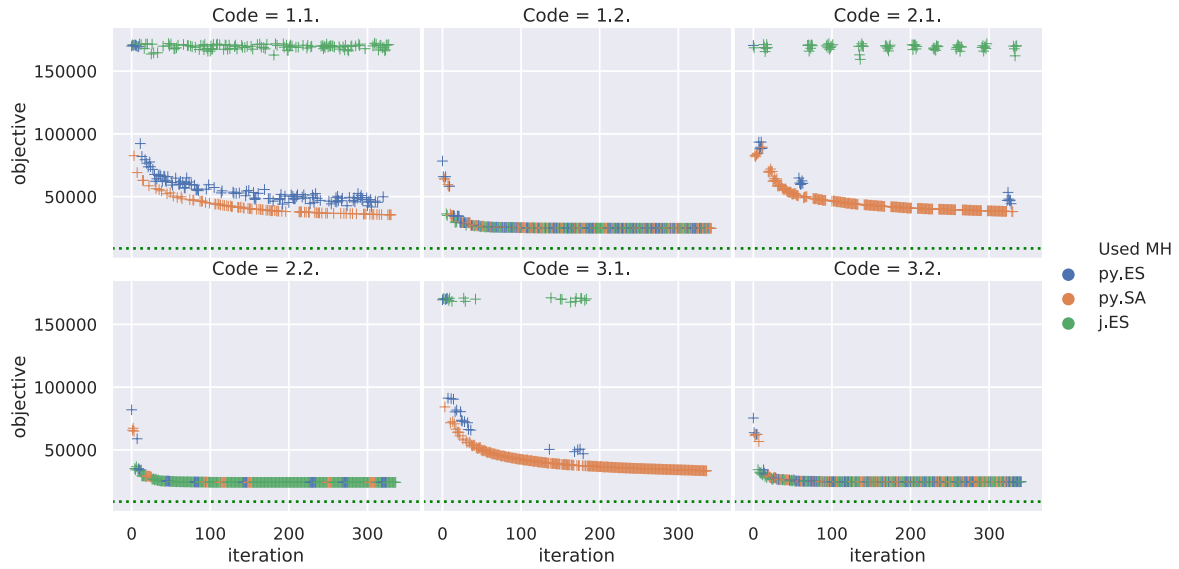


Figure 5.27 Intermediate performance of on-line selection hyper-heuristic with static hyper-parameters on rat783 (single experiment).

rat783 TSP instance. With the 783 cities instance heuristics are starting to perform in a slightly different manner even with the tuned parameters. As you remember from baseline evaluation, here the best performing was j.ES, afterwards appeared py.SA and lastly — py.ES. As a consequence, using FRAMAB HLH, HH-SP frequently utilizes j.ES (see code 2.2 in Figure 5.27). On contrary, BRR HLH utilizes all LLHs almost evenly. We may conclude that BRR was 'confused' by performance evidences from other heuristics, since the process quickly converged into a local optima.

In the case with default parameter values, both learning models most frequently predicted the best performing py.SA, which is an expected behavior.

The final result quality of all model-based HP-SP are at least as good, as the solution quality provided by the best underlying LLH for both default and tuned parameter values (Figure 5.28). Even in a case with the default parameters, a random allocation of one good performing LLH among three available (codes 1.1 and 1.2) results in a good final solution quality however, may require more time to converge (see intermediate and final performances for code 1.1).

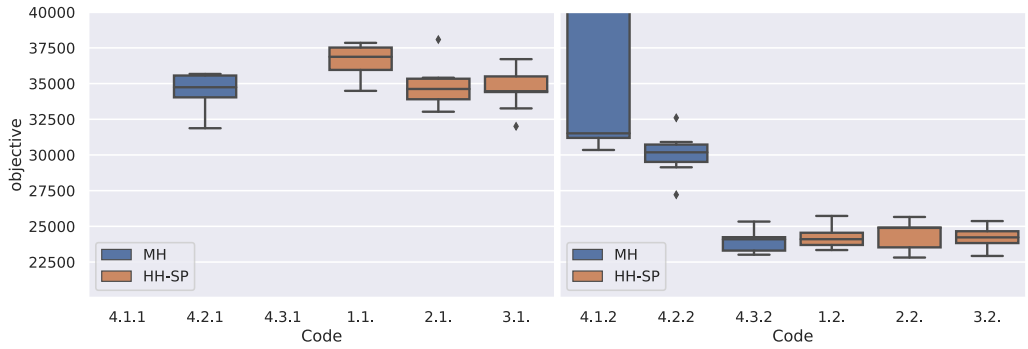


Figure 5.28 Final results of on-line selection hyper-heuristic with static hyper-parameters on rat783 (statistic of 9 runs).

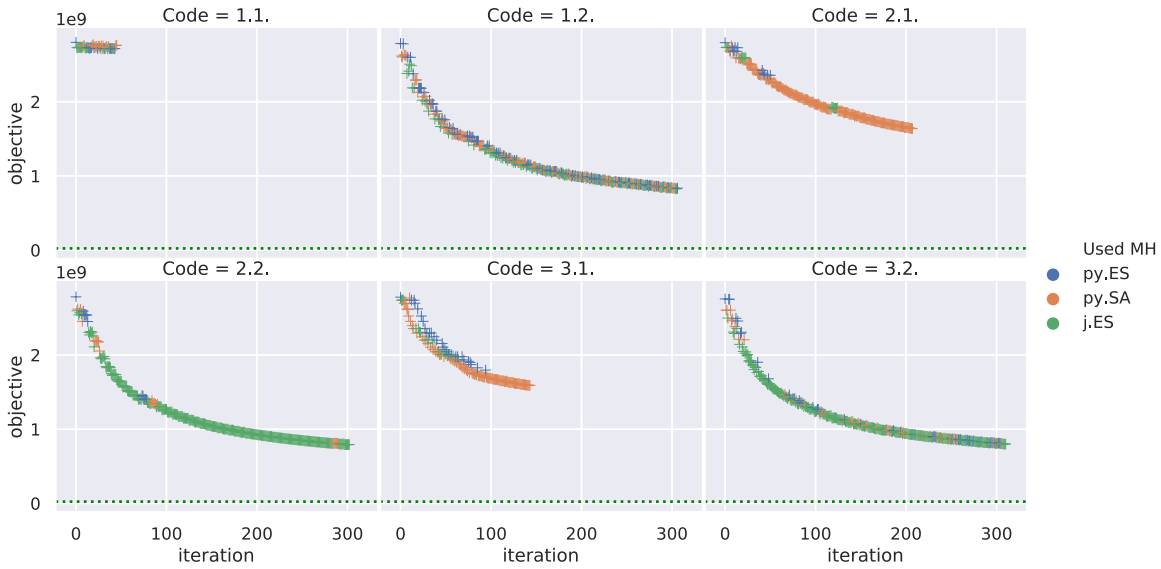


Figure 5.29 Intermediate performance of on-line selection hyper-heuristic with static hyper-parameters on pla7397 (single experiment).

pla7397 TSP instance. Finally, the largest TSP instance tackled in this thesis. Our observations different MH heuristics on this problem were as following: py.ES with default parameters had the worst performance, while the tuned algorithm version was able to outperform only default parameters of j.ES. On contrary, j.ES with tuned parameters produced the best results, outperforming py.SA. The Python version of simulated annealing with both parameter settings produced average results, which were better than default version of j.ES and both py.ES. All these performance evidences are presented in Figure 5.13. Therefore, here we expected to observe a high frequency of usage: (1) py.SA for default parameter case, and (2) j.ES for tuned parameters. Naturally, it holds only for learning-based HH-SP.

The displayed chronic of LLHs allocation in Figure 5.29 completely match our expectations. In case of usage LLHs with default parameters, the most frequent choice of FRAMAB HLH was py.SA (code 2.1). We observe the same behavior of BRR HLH, but this model was also frequently selecting other LLHs such as py.ES(code 3.1). The results are happening to be unexpected, when comparing random allocation with BRR-based (codes 1.1 and 3.1): the frequent usage of py.ES in 1.1 'killed' the optimization process, while the same frequent MH calls in 3.1 were performing much better. Referring to the final results, presented in Figure 5.30 we observe a high diverse in final result qualities when py.ES was allocated frequently (codes 1.1. and 3.1.). This is an unexpected behavior, especially for BRR HLH (code 3.1) that

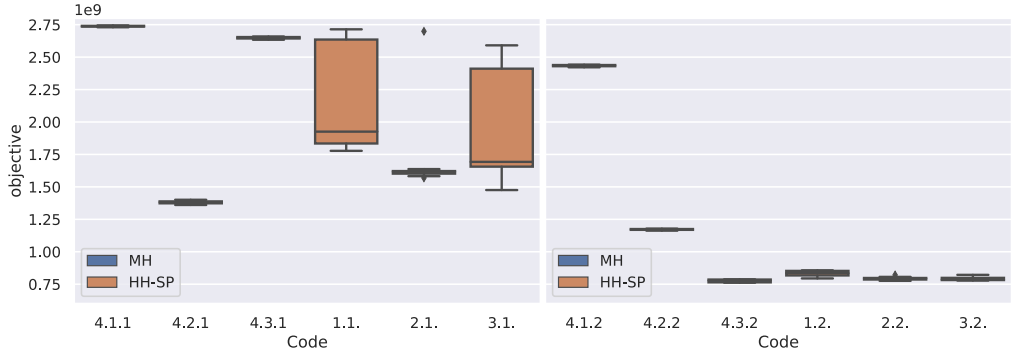


Figure 5.30 Final results of on-line selection hyper-heuristic with static hyper-parameters on pla7397 (statistic of 9 runs).

should be thoroughly examined, but due to the time limit we are forced to postpone the investigation for a future work.

As for the case with tuned LLHs, all selection approaches produced comparable to the best available LLHs final results. However, due to frequent usage of j.ES by FRAMAB and BRR, their final result quality slightly outperform random-based selection.

Discussion. According to our observations of the developed selection hyper-heuristic performance we conclude that a proposed concept implementation performs as expected. Two utilized LLH selection approaches are performing differently when reach a local optimum. We claim the FRAMAB is a more perspective HLH, since after reaching local optima it starts to balance between previously seeing good performing LLH. In a contrary, BRR continues to select only the best performing heuristic. In case when the advantage of one LLH changes to another, BRR may require more time to learn this. Nevertheless, it is only our guesses, which need the evaluation proves.

Generally speaking, observed issues require not only a thorough investigation (pla7397 code 1.1, 3.1), but also a generic approach to handle a potential flaws in LLH performance that may cause struggling of overall HH-SP execution. The implemented system should be evaluated by means of HLH configuration influence on the performance. Also, a further investigation of adding several new LLHs should be evaluated by means of required computation effort to find good LLHs vs pure performance of these LLHs.

Selection Hyper-Heuristic with Parameter Control

The final evaluation is dedicated to performance analysis of the suggested approach of merging selection hyper-heuristic with generic parameter control in LLHs. A minimal goal of both (1) HH-SP and (2) MH-PC merge is to obtain the best algorithm (1) with tuned hyper-parameters (2) performance. In this evaluation set we follow a similar to previously used approach of intermediate results review distinguishing allocated LLH types at each iteration and comparing the final results quality with a baseline. Once again, for the intermediate results we outline only a single repetition, while the final results are compared by means of all 9 repetitions. As specified in the evaluation plan (Section 5.4.1), for the LLH selection we use three approaches: random, FRAMAB and BRR sampling, while for setting the LLH parameters we use the same random and BRR sampling, but apply TPE instead of FRAMAB.

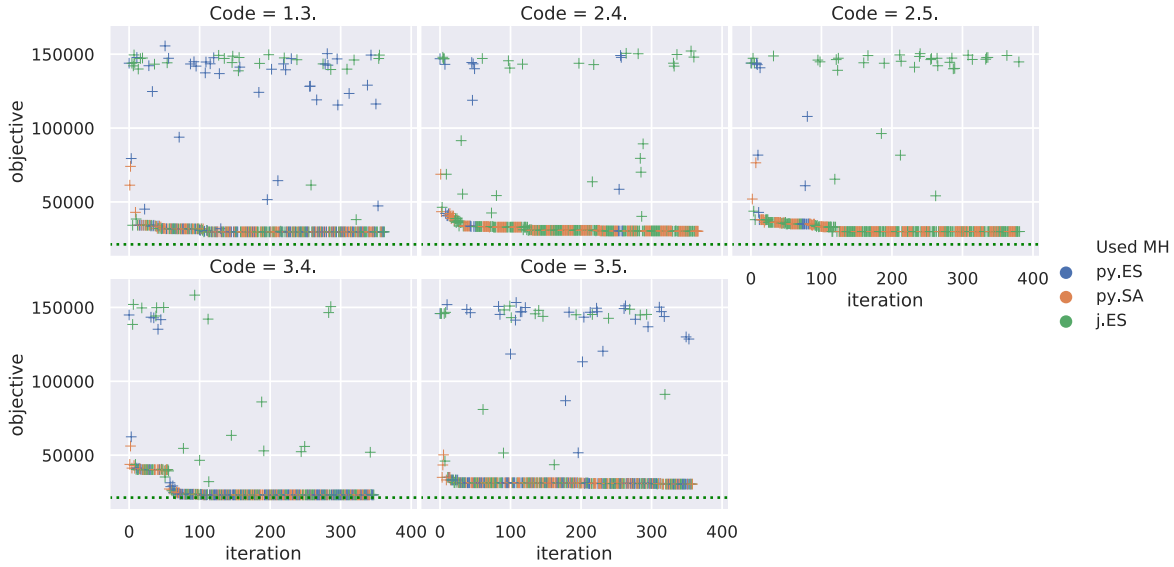


Figure 5.31 Intermediate performance of HH-PC on kroA100 (single experiment).

kroA100 TSP instance. During the solving process a similar to HH-SP patterns of algorithm allocation may be observed for both FRAMAB-based (codes 2.4, 2.5) and BRR (codes 3.4, 3.5) HLHs. However, in this case the intermediate results are differing, since the parameter control has started to search for good LLHs settings (Figure 5.31). For instance, consider the appeared 'noise' in the results of both evolution strategies heuristics. It is caused by a boolean parameter *elitist*, which defines the selection strategy. If the values is `false`, the outcome of heuristic run may be a dramatic decrease in solution quality (a more detailed description could be found in Section 2.2.2). We may observe a case, when py.ES using a `elitist=true` parameter setting made an improvement in solution quality, see Figure 5.31, code 3.4, around 50n iteration. TPE model (used to tune parameters in code 3.4) learned it and did not use an `elitist=false` parameter combination for this MH later in this experiment: only a j.ES-based noise appearing afterwards. Simultaneously, BRR model (used to select LLH in code 3.4) learned the improvement done by py.ES and selected it more frequently.

Since kroA100 is a relatively small TSP instance, the final performance of all approaches, including random LLH and parameter selection approach (code 1.3) are roughly the same (Figure 5.32). Nevertheless, all of them outperform a baseline, even with the tuned parameters (4.{1,2,3}.1 for default and 4.{1,2,3}.2 for tuned py.ES, py.SA, j.ES respectively in Figure 5.32).

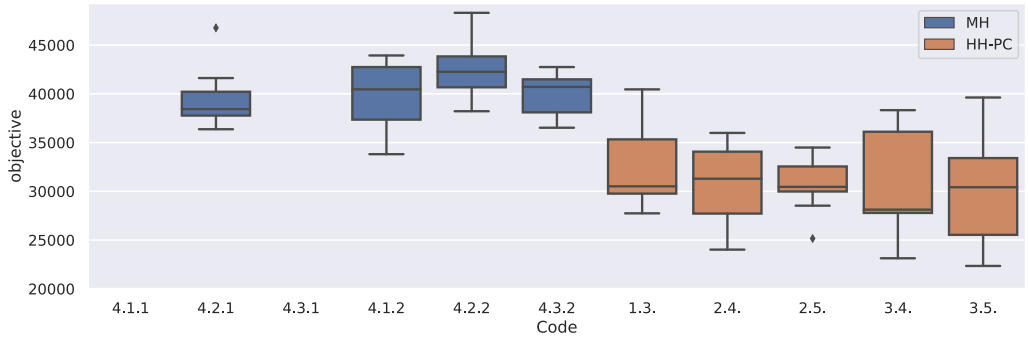


Figure 5.32 Final results of HH-PC compared with MH on kroA100 (statistic of 9 runs).

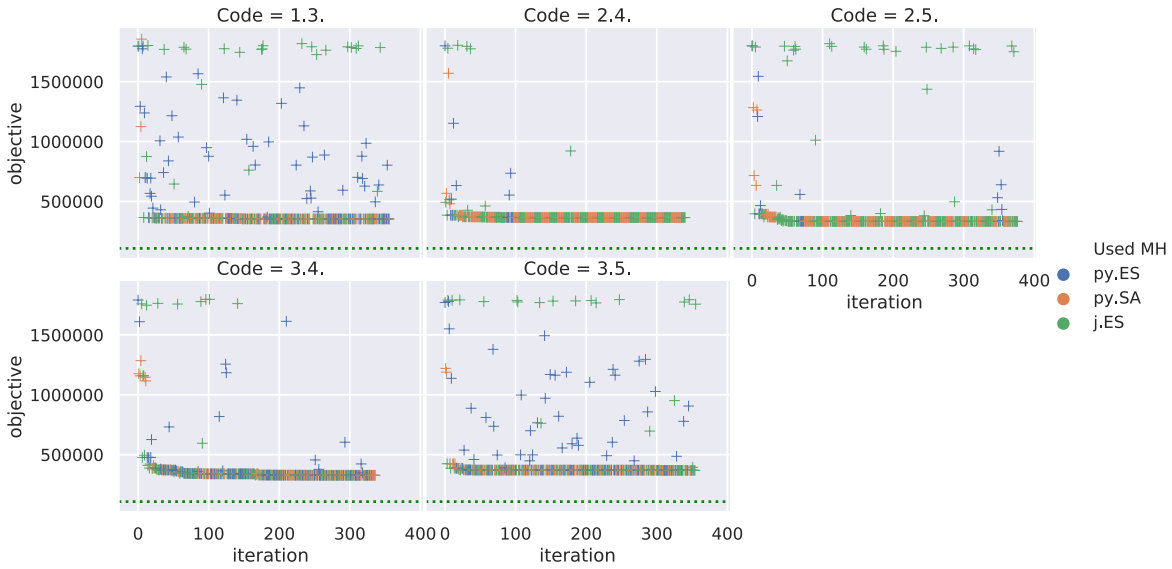


Figure 5.33 Intermediate performance of HH-PC on pr439 (single experiment).

pr439 TSP instance. With this TSP instance the learning process quickly converged in a local optima without a making an improvement moves. Because of this, the LLH allocation schedule, performed by FRAMAB (codes 2.4 and 2.5) is similar to one observed in HH-SP: two solvers were repeatedly executed, one is j.ES and the another is py.SA (Figure 5.33). In a contrary, BRR LLH selector did not manage to find an outperforming solver, therefore, was executing them uniformly (codes 3.4, 3.5), similarly to a pure random approach (code 1.3). Talking about the same distinguishable *elitist* parameter, TPE-based parameter control algorithm in both cases (codes 2.4, 3.4) found out that it rarely provide a good solution quality, therefore was setting it to `elitist=false` less frequently.

Comparing the final results quality of baseline with implemented HH-PC on this problem instance, we observe a similar to tuned MHs (codes 4.{1,2,3}.2) performance in all cases (Figure 5.34). Once again, the purely randomized selection of both LLHs and their parameters gave a good performance (code 1.3), which may be explained by a random allocation of py.SA, which is less sensitive to parameter settings. During these calls, the algorithm managed to produce the most valuable improvements to the solution quality (see the beginning of solving process in Figure 5.33 code 1.3).

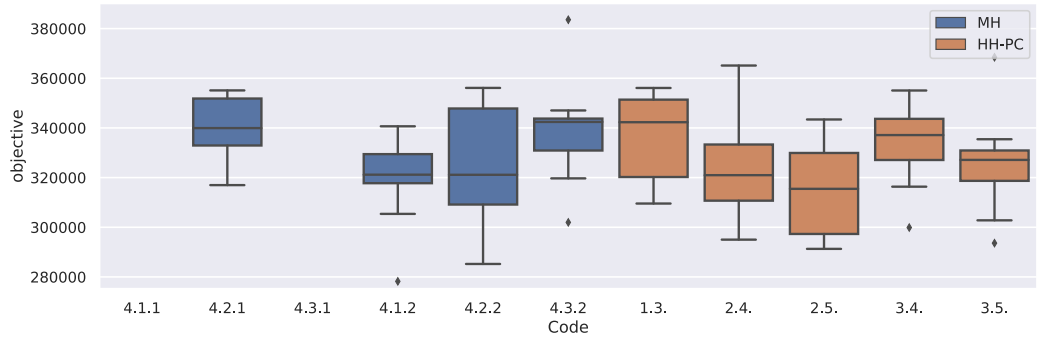


Figure 5.34 Final results of HH-PC compared with MH on pr439 (statistic of 9 runs).

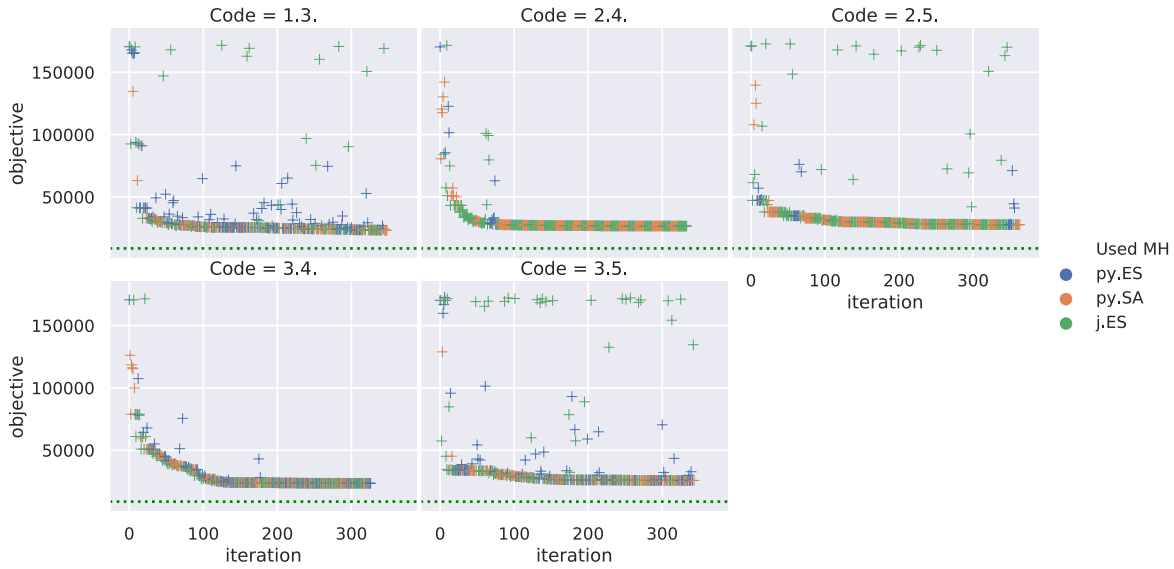


Figure 5.35 Intermediate performance of HH-PC on rat783 (single experiment).

rat783 TSP instance. With the third problem instance namely the solving process is becoming more demonstrative. Let us firstly draw attention towards HH-PC with FRAMAB for LLH and TPE for parameter sampling. At the beginning of solving process, j.ES was performing extremely well, for this reason FRAMAB was sampling it the most frequently (see beginning of experiment encoded by code 2.4 in Figure 5.35). When a solving process in j.ES reached its local optima, FRAMAB started to use other heuristics by means of exploration. From an absence of noise its may be seeing, that TPE has set the most perspective parameters to LLHs (codes 2.4 and 3.4), but they have reached their local optima. In such case, the parameter search should be biased towards exploitation. To do so, we may need to introduce new and different progress evidences, such as stagnation detection (which is used in EA parameter control [60]). Unfortunately, due to a lack of time, we postpone this enhancement for a future work.

The final performance evidences of the proposed concept implementation, unfortunately, do not reach the desired level. More concretely, in Figure 5.36 we clearly see the dominating j.ES MH with tuned parameters (code 4.3.2). As we claimed in the beginning of HH-PC performance review, the expected minimal final solution quality of HH-PC should be as good as tuned j.ES. But we observe an averaged performance among all available tuned LLHs (codes 4.1.2, 4.2.2, 4.3.2 for tuned MHs vs codes 1.3, 2.4, 2.5, 3.4, 3.5).

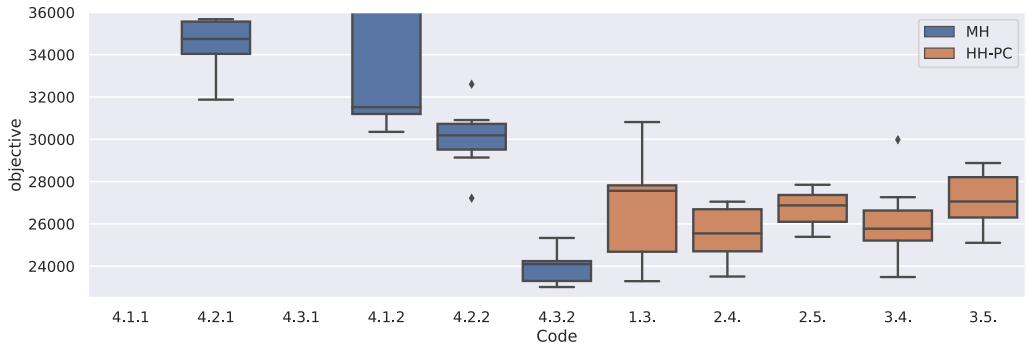


Figure 5.36 Final results of HH-PC compared with MH on rat783 (statistic of 9 runs).

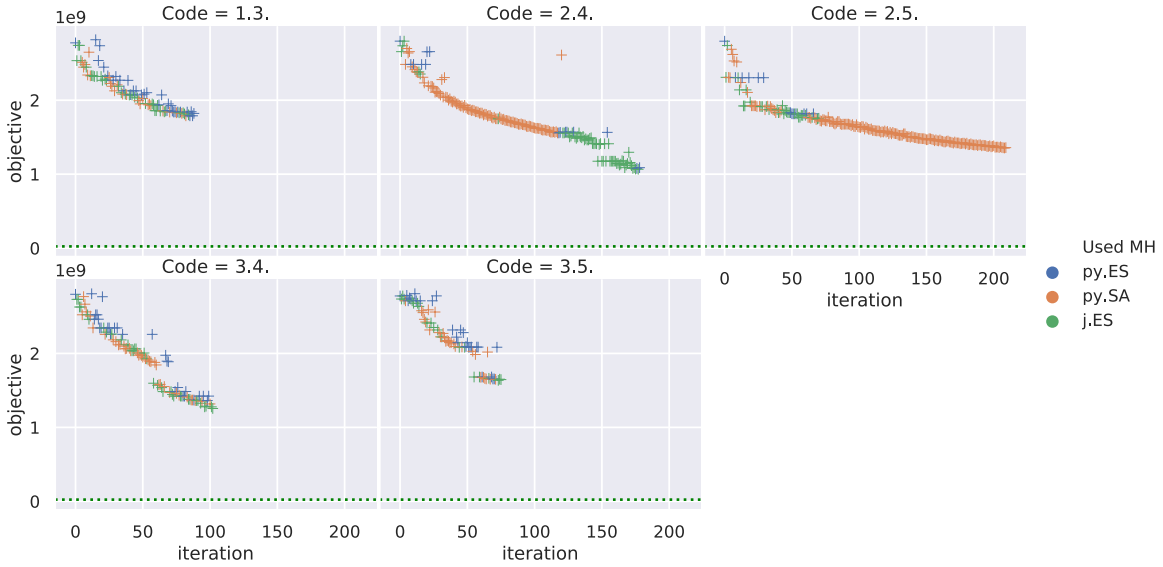


Figure 5.37 Intermediate performance of HH-PC on pla7397 (single experiment).

pla7397 TSP instance. The final and largest problem instance, which should relieve the proposed and implemented concept applicability fully.

As we observed in previous experiments, py.ES with non-tuned parameters may cause a struggling in hyper-heuristic performance (see Figure 5.13 and the discussion below Figure 5.21). Thus, during these benchmarks py.ES made a crucial change in the overall number of iterations where it was used many times (codes 1.3, 3.4, 3.4 in Figure 5.37). While the case of fully randomized HH-PC (code 1.3) is clear, a BRR LLH selection did use this LLH frequently, since it produced a good results as a result of parameter control behavior. In a contrary, FRAMAB LLH selection in combination with TPE parameter tuning (code 2.4) managed to find good parameters for py.SA, therefore, allocated frequently. When py.SA reached its local optimum, FRAMAB triggered other LLHs usage and as a result, switched to j.ES's usage, which gave dramatic result improvements. The FRAMAB LLH selection with BRR parameter control (code 2.5) at the beginning of run was using the mixture of mainly two j.ES and py.SA, but later switched to py.SA-only mode. As we observe, this gave a fast coarse-grained solution improvement in the beginning, and stable, but rather slow improvement in a later stage.

On the final results quality charts (Figure 5.38) we observe a dominance of FRAMAB-based LLH selection (codes 2.4, 2.5) and TPE-based parameter control (codes 2.4 and 3.4) usage. However, the results obtained with BRR-based parameter control approach (codes 2.5 and 3.5) are more stable than TPE-based (codes 2.4 and 3.4). Once again, the final results did not reach a desired performance of the

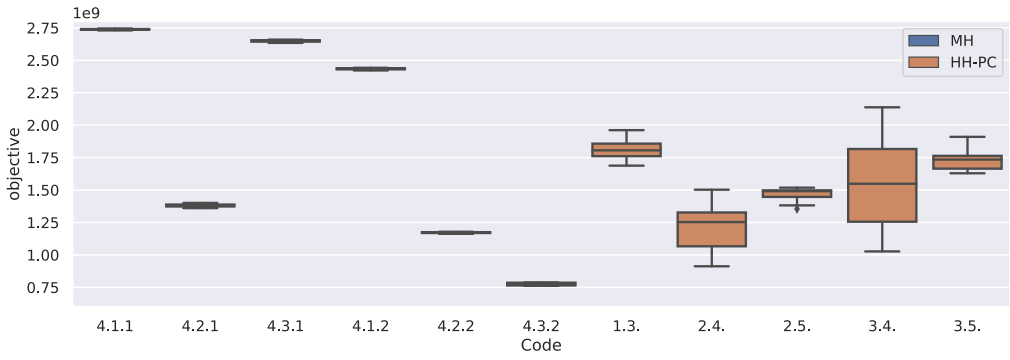


Figure 5.38 Final results of HH-PC compared with MH on pla7397 (statistic of 9 runs).

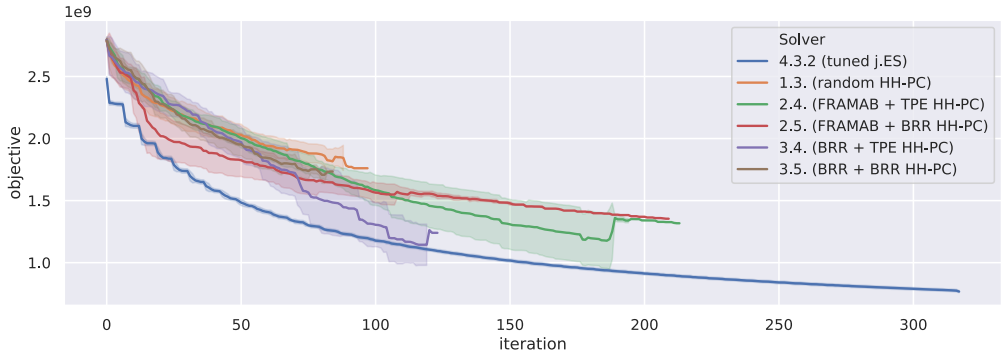


Figure 5.39 HH-PC and tuned jMetal ES solving process comparison on pla7397 (statistic of 9 runs).

best performing tuned j.ES (code 4.3.2). However, since the optimization process did not settle in a local optima, we can not doubt that HH-PC will not outperform j.ES. given more time.

For a better intuition, let us draw the reader attention to Figure 5.39. Note how HH-PC with BRR LLH selection and TPE parameter tuning is approaching j.ES. If not the issue with a number of iterations, it would outperform the j.ES. However, in the current implementation given 15 minutes for all solvers j.ES managed to make more iterations and moved further. We postpone this discussion to the second main part of this chapter, dedicated to a coarse-grained HH-PC parameter settings influence evaluation.

5.4.3 Conclusion on Concept Evaluation

In previously performed analysis the suggested generic parameter control (MH-PC) proved its concept, while implemented dynamic selection of heuristics with static parameters (HH-SP) performed according to our expectations. As discussed during the literature review in Sections 2.2.5 and 2.3.1, each approach has a lack of another's advantageous features. Selecting good MH parameters, MH-PC will not outperform a better heuristic, while selected good algorithm with a poor parameter setting will not be able to compete with properly tuned one. Therefore, in proposed approach to merge both algorithm selection and parameter control in HH-PC we expected to observe

We preform a comparison of HH-PC with MH-PC and HH-SP afterwards.

5.5 Hyper-Heuristic with Parameter Control Settings Evaluation

5.5.1 Evaluation Plan

5.5.2 Evaluation Results

5.6 Conclusion

6 Conclusion

Reviewer: answer research questions

comparison to HITO [48]

Hyper-Heuristics with parameter tuning (or better say, control?), Constrainted Parameter Tuning and Architecture search problems all are the same? All these problems are seems to talk about the same thing, and trying to solve it in the same ways, while calling it differently. In one hand, it could be the result of relatively young research direction (in all cases). In other hand we could make such an assumption because knowledge lack ⌣.

Hyper-Heuristics and Automatic Machine Learning are the same.

Other worth-to-try approaches Selecting LLH and parameters according to Markov Decision Process: use discretization of parameters and predict reward using MCMC.

7 Future work

In this Chapter we discuss the investigations that we postponed as a future work. They could be grouped in the several sets therefore, we discuss them in a separate sections. Thus, in the Section 7.1 we discuss a set of enhancements for the search space implementation that should be performed to better suite for users needs. The Section 7.2 is dedicated to the prediction process and learning models, which guide the system performance. Finally, in the section Section 7.3 we discuss a benchmark experiments to obtain a better evidences about the implemented solution applicability.

7.1 Search Space

Composition on numerical parameters In the Section 4.2.2 we highlighted that the implemented search space is able to form the parent-child relationship only when parent is of the categorical type. However, it may happen, when the dependencies among parameters are based on their numeric values. As instance, for one range the first child type should be exposed, while for other range — another child. As a hint for future implementation we propose utilizing a similar to used in categorical parameters approach however, instead of activation values, use the ranges of values. Thus, during the prediction propagation step the parameter will simply check all ranges and trigger the related children (for more details see description of the Listing 4.6).

Constraints among parameters Sometimes, the prohibitions for a specific values may arise with respect to other parameters. For instance, the value of one numeric parameter should be at least as high as value of the other.

7.2 Prediction Process

add more sophisticated models

technique to optimize obtained surrogate model should be generalized I did not investigate decoupling the surrogate models from the search algorithm to optimize those surrogates (done in Sasha's thesis).

investigate more deeply decoupling of data preprocessing and learning algorithm auto-sklearn

influence for warm-start onto this kind of HH (by off-line learning) Although, the influence of meta-learning, applied in Auto-Sklearn system [40] to warm-start the learning mechanism proved to worth the effort spent, as well as it was reported by developers of Selective Hyper-Heuristics with mixed type of learning [108], it is intriguing to check an influence of metal-learning onto Selective Hyper-Heuristics with Parameter Control. Also, $https://ml.informatik.uni-freiburg.de/papers/20-ECAI-DAC.pdf$

Random Forest HLH surrogate model

add other learning metrics Inspiration could be found at: - EAs in [60]: progress stagnation,

7.3 Evaluations and Benchmarks

add new class of problem (jmetalpy easly allows it)

evaluation on different types and classes

interesting direction: apply to automatic machine learning problems solving, compare to auto-sklearn.

bounding LLH by number of evaluations, not time

adaptive time for tasks

influence of adding new LLHs onto the required time to (1) find good LLHs in HH-SP and good configuration in HH-DP

Bibliography

[1] Aldeida Aleti and Irene Moser. "A systematic literature review of adaptive parameter control methods for evolutionary algorithms". In: *ACM Computing Surveys (CSUR)* 49.3 (2016), pp. 1–35.

[2] Satyajith Amaran et al. "Simulation optimization: a review of algorithms and applications". In: *Annals of Operations Research* 240.1 (2016), pp. 351–380.

[3] David L Applegate et al. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[4] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47.2-3 (2002), pp. 235–256.

[5] Nader Azizi and Saeed Zolfaghari. "Adaptive temperature control for simulated annealing: a comparative study". In: *Computers & Operations Research* 31.14 (2004), pp. 2439–2451.

[6] Antonio Benitez-Hidalgo et al. "jMetalPy: a python framework for multi-objective optimization with metaheuristics". In: *Swarm and Evolutionary Computation* 51 (2019), p. 100598.

[7] Thierry Benoist et al. "Localsolver 1. x: a black-box local-search solver for 0-1 programming". In: *4or* 9.3 (2011), p. 299.

[8] Thierry Benoist et al. "Toward local search programming: LocalSolver 1.0". In: *CPAIOR workshop: open source tools for constraint programming and mathematical programming*. Vol. 49. 2010.

[9] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *Journal of machine learning research* 13.Feb (2012), pp. 281–305.

[10] James S Bergstra et al. "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems*. 2011, pp. 2546–2554.

[11] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategies–A comprehensive introduction". In: *Natural computing* 1.1 (2002), pp. 3–52.

[12] Leonora Bianchi et al. "A survey on metaheuristics for stochastic combinatorial optimization". In: *Natural Computing* 8.2 (2009), pp. 239–287.

[13] A. Biedenkapp et al. "Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework". In: *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20)*. June 2020.

[14] Lorenz T Biegler and Ignacio E Grossmann. "Retrospective on optimization". In: *Computers & Chemical Engineering* 28.8 (2004), pp. 1169–1192.

[15] Mauro Birattari et al. "Classification of Metaheuristics and Design of Experiments for the Analysis of Components Tech. Rep. AIDA-01-05". In: (2001).

[16] Mauro Birattari et al. "F-Race and iterated F-Race: An overview". In: *Experimental methods for the analysis of optimization algorithms*. Springer, 2010, pp. 311–336.

[17] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. "The job shop scheduling problem: Conventional and new solution techniques". In: *European journal of operational research* 93.1 (1996), pp. 1–33.

[18] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. "A survey on optimization metaheuristics". In: *Information Sciences* 237 (2013). Prediction, Control and Diagnosis using Advanced Neural Computations, pp. 82–117.

[19] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[20] Yuriy Brun et al. "Engineering self-adaptive systems through feedback loops". In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48–70.

[21] Edmund Burke et al. "Hyper-heuristics: An emerging direction in modern search technology". In: *Handbook of metaheuristics*. Springer, 2003, pp. 457–474.

[22] Edmund K Burke et al. "A classification of hyper-heuristic approaches: revisited". In: *Handbook of Metaheuristics*. Springer, 2019, pp. 453–477.

[23] Edmund K Burke et al. "Hyper-heuristics: A survey of the state of the art". In: *Journal of the Operational Research Society* 64.12 (2013), pp. 1695–1724.

[24] Diego Calderon et al. "Conditional Linear Regression". In: *CoRR* abs/1806.02326 (2018). arXiv: 1806.02326.

[25] Taesu Cheong and Chelsea C White. "Dynamic traveling salesman problem: Value of real-time traffic information". In: *IEEE Transactions on Intelligent Transportation Systems* 13.2 (2011), pp. 619–630.

[26] Wikimedia Commons. *File:Metaheuristics classification fr.svg — Wikimedia Commons the free media repository*. [Online; accessed 16-March-2020]. 2017.

[27] William Jay Conover and William Jay Conover. "Practical nonparametric statistics". In: (1980).

[28] Juan De Vicente, Juan Lanchares, and Román Hermida. "Placement by thermodynamic simulated annealing". In: *Physics Letters A* 317.5-6 (2003), pp. 415–423.

[29] Kalyanmoy Deb. "Multi-objective optimization". In: *Search methodologies*. Springer, 2014, pp. 403–449.

[30] Benjamin Doerr and Carola Doerr. "Theory of parameter control for discrete black-box optimization: Provable performance gains through dynamic parameter choices". In: *Theory of Evolutionary Computation*. Springer, 2020, pp. 271–321.

[31] Marco Dorigo. "Ant colony optimization". In: *Scholarpedia* 2.3 (2007), p. 1461.

[32] John H Drake et al. "Recent advances in selection hyper-heuristics". In: *European Journal of Operational Research* (2019).

[33] Juan J Durillo and Antonio J Nebro. "jMetal: A Java framework for multi-objective optimization". In: *Advances in Engineering Software* 42.10 (2011), pp. 760–771.

[34] AE Eiben and JE Smith. "Popular Evolutionary Algorithm Variants". In: *Introduction to Evolutionary Computing*. Springer, 2015, pp. 99–116.

[35] Agoston E Eiben and James E Smith. "What is an evolutionary algorithm?" In: *Introduction to Evolutionary Computing*. Springer, 2015, pp. 25–48.

[36] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey". In: *arXiv preprint arXiv:1808.05377* (2018).

[37] Stefan Falkner, Aaron Klein, and Frank Hutter. "BOHB: Robust and efficient hyperparameter optimization at scale". In: *arXiv preprint arXiv:1807.01774* (2018).

[38] Alexandre Silvestre Ferreira, Richard Aderbal Gonçalves, and Aurora Pozo. "A multi-armed bandit selection strategy for hyper-heuristics". In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2017, pp. 525–532.

[39] P Festa. "A brief introduction to exact, approximation, and heuristic algorithms for solving hard combinatorial optimization problems". In: *2014 16th International Conference on Transparent Optical Networks (ICTON)*. IEEE. 2014, pp. 1–20.

[40] Matthias Feurer et al. "Efficient and robust automated machine learning". In: *Advances in neural information processing systems*. 2015, pp. 2962–2970.

[41] Matthias Feurer et al. "OpenML-Python: an extensible Python API for OpenML". In: *arXiv* 1911.02490 ().

[42] Goncalo Figueira and Bernardo Almada-Lobo. "Hybrid simulation–optimization methods: A taxonomy and discussion". In: *Simulation Modelling Practice and Theory* 46 (Aug. 2014).

[43] Fedor V Fomin and Petteri Kaski. "Exact exponential algorithms". In: *Communications of the ACM* 56.3 (2013), pp. 80–88.

[44] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[45] Kambiz Shojaee Ghandeshtani and Habib Rajabi Mashhadi. "An entropy-based self-adaptive simulated annealing". In: *Engineering with Computers* (2019), pp. 1–27.

[46] Fred Glover. "Tabu search—part I". In: *ORSA Journal on computing* 1.3 (1989), pp. 190–206.

[47] Oded Goldreich. *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press, 2010.

[48] Giovani Guizzo et al. "A hyper-heuristic for the multi-objective integration and test order problem". In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 2015, pp. 1343–1350.

[49] Pierre Hansen and Nenad Mladenović. "Variable neighborhood search". In: *Handbook of metaheuristics*. Springer, 2003, pp. 145–184.

[50] Arthur E Hoerl and Robert W Kennard. "Ridge regression: Biased estimation for nonorthogonal problems". In: *Technometrics* 12.1 (1970), pp. 55–67.

[51] Robert Hooke and Terry A Jeeves. ""Direct Search"Solution of Numerical and Statistical Problems". In: *Journal of the ACM (JACM)* 8.2 (1961), pp. 212–229.

[52] Changwu Huang, Yuanxiang Li, and Xin Yao. "A Survey of Automatic Parameter Tuning Methods for Metaheuristics". In: *IEEE Transactions on Evolutionary Computation* (2019).

[53] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *International conference on learning and intelligent optimization*. Springer. 2011, pp. 507–523.

[54] Frank Hutter et al. "ParamILS: an automatic algorithm configuration framework". In: *Journal of Artificial Intelligence Research* 36 (2009), pp. 267–306.

[55] Lester Ingber. "Adaptive simulated annealing (ASA): Lessons learned". In: *arXiv preprint cs/0001018* (2000).

[56] Haifeng Jin, Qingquan Song, and Xia Hu. "Auto-Keras: An Efficient Neural Architecture Search System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2019, pp. 1946–1956.

[57] Donald R Jones, Matthias Schonlau, and William J Welch. "Efficient global optimization of expensive black-box functions". In: *Journal of Global optimization* 13.4 (1998), pp. 455–492.

[58] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. *Combinatorial Optimization–Eureka, You Shrink!: Papers Dedicated to Jack Edmonds. 5th International Workshop, Aussois, France, March 5-9, 2001, Revised Papers*. Vol. 2570. Springer, 2003.

[59] Mariia Karabin and Steven J Stuart. "Simulated Annealing with Adaptive Cooling Rates". In: *arXiv preprint arXiv:2002.06124* (2020).

[60] Giorgos Karafotias, Agoston Endre Eiben, and Mark Hoogendoorn. "Generic parameter control with reinforcement learning". In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 2014, pp. 1319–1326.

[61] Giorgos Karafotias, Mark Hoogendoorn, and Ágoston E Eiben. "Parameter control in evolutionary algorithms: Trends and challenges". In: *IEEE Transactions on Evolutionary Computation* 19.2 (2014), pp. 167–187.

[62] James Kennedy and Russell Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95-International Conference on Neural Networks*. Vol. 4. IEEE. 1995, pp. 1942–1948.

[63] Pascal Kerschke et al. "Automated algorithm selection: Survey and perspectives". In: *Evolutionary computation* 27.1 (2019), pp. 3–45.

[64] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. "Optimization by simulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[65] Patrick Koch et al. "Automated hyperparameter tuning for effective machine learning". In: *Proceedings of the SAS Global Forum 2017 Conference*. 2017.

[66] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn". In: *ICML workshop on AutoML*. Vol. 9. Citeseer. 2014.

[67] John R Koza. "Evolution of subsumption using genetic programming". In: *Proceedings of the First European Conference on Artificial Life*. 1992, pp. 110–119.

[68] Gilbert Laporte. "The vehicle routing problem: An overview of exact and approximate algorithms". In: *European journal of operational research* 59.3 (1992), pp. 345–358.

[69] Niklas Lavesson and Paul Davidsson. "Quantifying the impact of learning algorithm parameter tuning". In: *AAAI*. Vol. 6. 2006, pp. 395–400.

[70] Julien-Charles Lévesque et al. "Bayesian optimization for conditional hyperparameter spaces". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 286–293.

[71] Ke Li et al. "Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition". In: *IEEE Transactions on Evolutionary Computation* 18.1 (2013), pp. 114–130.

[72] Lisha Li et al. "Hyperband: A novel bandit-based approach to hyperparameter optimization". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6765–6816.

[73] M. Lindauer et al. "BOAH: A Tool Suite for Multi-Fidelity Bayesian Optimization & Analysis of Hyperparameters". In: *arXiv:1908.06756 [cs.LG]* (2019).

[74] Manuel López-Ibáñez et al. "The irace package: Iterated racing for automatic algorithm configuration". In: *Operations Research Perspectives* 3 (2016), pp. 43–58.

[75] Zhihao Lou and John Reinitz. "Parallel simulated annealing using an adaptive resampling interval". In: *Parallel computing* 53 (2016), pp. 23–31.

[76] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. “Iterated local search”. In: *Handbook of metaheuristics*. Springer, 2003, pp. 320–353.

[77] Silvano Martello and Paolo Toth. “Bin-packing problem”. In: *Knapsack problems: Algorithms and computer implementations* (1990), pp. 221–245.

[78] Olivier C Martin and Steve W Otto. “Combining simulated annealing with local search heuristics”. In: *Annals of Operations Research* 63.1 (1996), pp. 57–75.

[79] Kent McClymont and Edward C Keedwell. “Markov chain hyper-heuristic (MCHH) an online selective hyper-heuristic for multi-objective continuous problems”. In: *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 2011, pp. 2003–2010.

[80] Mustafa Mısır et al. “An intelligent hyper-heuristic framework for chesc 2011”. In: *International Conference on Learning and Intelligent Optimization*. Springer. 2012, pp. 461–466.

[81] David E Moriarty, Alan C Schultz, and John J Grefenstette. “Evolutionary algorithms for reinforcement learning”. In: *Journal of Artificial Intelligence Research* 11 (1999), pp. 241–276.

[82] Gabriela Ochoa et al. “Hyflex: A benchmark framework for cross-domain heuristic search”. In: *European Conference on Evolutionary Computation in Combinatorial Optimization*. Springer. 2012, pp. 136–147.

[83] Randal S Olson and Jason H Moore. “TPOT: A tree-based pipeline optimization tool for automating machine learning”. In: *Automated Machine Learning*. Springer, 2019, pp. 151–160.

[84] Federico Pagnozzi and Thomas Stützle. “Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems”. In: *European journal of operational research* 276.2 (2019), pp. 409–421.

[85] Judea Pearl. “Intelligent search strategies for computer problem solving”. In: *Addision Wesley* (1984).

[86] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[87] Nelishia Pillay and Derrick Beckedahl. “EvoHyp-a Java toolkit for evolutionary algorithm hyper-heuristics”. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2017, pp. 2706–2713.

[88] Dmytro Pukhkaiev and Uwe Aßmann. “Parameter Tuning for Self-optimizing Software at Scale”. In: (Nov. 2019).

[89] Dmytro Pukhkaiev and Sebastian Götz. “BRISE: Energy-Efficient Benchmark Reduction”. In: *Proceedings of the 6th International Workshop on Green and Sustainable Software*. GREENS ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 23–30.

[90] Gerhard Reinelt. “Tsplib95”. In: *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg* 338 (1995).

[91] S Reza Hejazi* and S Saghafian. “Flowshop-scheduling problems with makespan criterion: a review”. In: *International Journal of Production Research* 43.14 (2005), pp. 2895–2929.

[92] Herbert Robbins. “Some aspects of the sequential design of experiments”. In: *Bulletin of the American Mathematical Society* 58.5 (1952), pp. 527–535.

[93] Tomas Roubicek. *Relaxation in optimization theory and variational calculus*. Vol. 4. Walter de Gruyter, 2011.

[94] Patricia Ryser-Welch and Julian F Miller. "A review of hyper-heuristic frameworks". In: *Proceedings of the Evo20 Workshop, AISB*. Vol. 2014. 2014.

[95] S Salcedo-Sanz et al. "The coral reefs optimization algorithm: a novel metaheuristic for efficiently solving optimization problems". In: *The Scientific World Journal* 2014 (2014).

[96] Kumara Sastry, David Goldberg, and Graham Kendall. "Genetic algorithms". In: *Search methodologies*. Springer, 2005, pp. 97–125.

[97] Julia Schroeter, Malte Lochau, and Tim Winkelmann. "Multi-perspectives on feature models". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2012, pp. 252–268.

[98] Bobak Shahriari et al. "Taking the human out of the loop: A review of Bayesian optimization". In: *Proceedings of the IEEE* 104.1 (2015), pp. 148–175.

[99] Jim Smith. "Self adaptation in evolutionary algorithms". PhD thesis. 2020.

[100] Kenneth Sörensen, Marc Sevaux, and Fred Glover. "A History of Metaheuristics". In: *Handbook of Heuristics* to appear (Jan. 2017).

[101] Jerry Swan, Ender Özcan, and Graham Kendall. "Hyperion–a recursive hyper-heuristic framework". In: *International Conference on Learning and Intelligent Optimization*. Springer. 2011, pp. 616–630.

[102] Michael Syrjakow and Helena Szczerbicka. "Efficient parameter optimization based on combination of direct global and local search methods". In: *Evolutionary Algorithms*. Springer. 1999, pp. 227–249.

[103] Dusan Teodorovic et al. "Bee colony optimization: principles and applications". In: *2006 8th Seminar on Neural Network Applications in Electrical Engineering*. IEEE. 2006, pp. 151–156.

[104] Jonathan Thompson and Kathryn A Dowsland. "General cooling schedules for a simulated annealing based timetabling system". In: *International Conference on the Practice and Theory of Automated Timetabling*. Springer. 1995, pp. 345–363.

[105] Chris Thornton et al. "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2013, pp. 847–855.

[106] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.

[107] Edward Tsang and Chris Voudouris. "Fast local search and guided local search and their application to British Telecom's workforce scheduling problem". In: *Operations Research Letters* 20.3 (1997), pp. 119–127.

[108] Gönül Uludağ et al. "A hybrid multi-population framework for dynamic environments combining on and offline learning". In: *Soft Computing* 17.12 (2013), pp. 2327–2348.

[109] Enrique Urra Coloma et al. "hMod: A software framework for assembling highly detailed heuristics algorithms". In: *Software Practice and Experience* 2019 (Mar. 2019), pp. 1–24.

[110] Peter JM Van Laarhoven and Emile HL Aarts. "Simulated annealing". In: *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[111] Christos Voudouris and Edward Tsang. "Guided local search and its application to the traveling salesman problem". In: *European journal of operational research* 113.2 (1999), pp. 469–499.

[112] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[113] Gerhard J Woeginger. “Exact algorithms for NP-hard problems: A survey”. In: *Combinatorial optimization—eureka, you shrink!* Springer, 2003, pp. 185–207.

[114] David H Wolpert and William G Macready. “No free lunch theorems for optimization”. In: *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 67–82.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 1st May 2020