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, Ukraine
Course: Distributed Systems Engineering
Matriculation number: 4733680
Matriculation year: 2017

Master Thesis

to achieve the academic degree

Master of Science (M.Sc.)

Supervisors
M.Sc. 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

Abstract

The importance of balance between exploration and exploitation plays a crucial role while solving the combinatorial optimization problems. This balance is reached by two general techniques: by using an appropriate problem solver and by setting its proper parameters. Both problems were widely studied in the past and the research process continues up until now. The latest studies in the field of automated machine learning propose merging both problems, solving them at design time and later strengthening the results at runtime. To the best of our knowledge, the *generalized* approach for solving the parameter setting problem in heuristic solvers has not yet been proposed. Therefore, the concept of merging heuristic selection and parameter control has not been introduced.

In this thesis we propose an approach for generic parameter control in meta-heuristics by means of reinforcement learning (RL). Making a step further, we suggest a technique for merging the heuristic selection and parameter control problems and solving them at runtime using RL-based hyper-heuristic. The evaluation of the proposed parameter control technique on a symmetric traveling salesman problem (TSP) revealed its applicability by reaching the performance of tuned in offline and used in isolation underlying meta-heuristic. Our approach provides the results on par with the best underlying heuristics with tuned parameters.

Contents

1 Introduction

1.1 Motivation

Heuristic-based optimization is a popular research area. Various optimization problems (OPs) are defined and can be tackled by heuristic algorithms [14, 43, 63]. Unfortunately, an ideal algorithm that can solve every OP does not and cannot exist. This issue was formalized by the *no-free-lunch theorem for optimization* (NFLT) [121], which states that "all search algorithms have the same average performance over all possible optimization problems". Heuristic solver acts by means of *exploration* (effort diversification over a search space) and *exploitation* (effort intensification on a promising area) operations. The success of heuristic on the problem at hand is defined by the exposed strength of both operations (E&E) and the provided balance between them (EvE). Both E&E and EvE characteristics can be controlled in several ways.

Firstly, one could try to set proper values of hyper-parameters, exposed by the algorithm. This process is formalized under the notion of *parameter settings problem* (PSP), whose resolution can be done before running the algorithm (design time), or while it solves the OP (runtime). The former approach is also called *parameter tuning* and can be tackled by numerous universal tuning systems [41, 58, 59, 79, 93]. A key assumption of this software is an expensiveness of the target system evaluation in terms of computational resources. Expensiveness is tackled creating a surrogate learning model, which simulates direct evaluations. The latter approach *parameter control* was originally introduced for evolutionary algorithms [66] and nowadays appears in an *algorithm-dependent* manner. However, even a proper parameter setting may not lead to the best results for the problem at hand.

The second way for reaching the EvE balance is a proper solver selection. It was formalized as the *algorithm selection problem* (ASP) and defined as a process of searching appropriate solver for problem at hand. ASP resolves the direct consequence of NFLT, which states that single algorithm can not be used to tackle various problems. Hyper-heuristics are commonly used for solving ASPs. They may perform low-level heuristic selection before solving the actual problem, or at runtime [22]. To operate online, hyper-heuristics often utilize reinforcement learning (RL) techniques [84, 86], while for design time, a regular parameter tuning could be used.

The research is not at a standstill and nowadays the researchers are actively attempting to merge ASP and PSP into the united *algorithm selection and parameter setting problem* (APSP). For instance, in automatic machine learning such combination was formalized as the *combined algorithm selection and hyper-parameter optimization* problem (CASH) [112], while for heuristics the explicit studies of APSP merging and solving at runtime were not found. To tackle ML CASH problem several frameworks based on the existing parameter tuning systems were created [44, 88, 112]. However, those solutions are not applicable in case of heuristics, since they are (1) purely related to ML field and (2) acting at design time due to ML techniques nature. One may follow the ML approach of the united APSP search space definition and solving for heuristics, but it is applicable *only* at design time. However, when it comes to runtime, it turns out that the universal technique for setting the parameters online (parameter control) in heuristics has not yet been proposed. It is essential, since this (PSP) is one of two required methodologies for solving heuristic APSP at runtime. The other building block (ASP) is already available in online hyper-heuristics.

1.2 Research objective

The goal of this thesis is to improve the quality of online heuristic-based optimization. The research objective is to determine whether it is possible to solve both PSP and ASP, while solving the OP. To reach the research objective we need to answer the following RQs:

- **RQ1** Is it possible to perform the algorithm configuration at runtime on a generic level?
- **RQ2** Is it possible to simultaneously perform algorithm selection and parameters adaptation while solving an optimization problem?
- **RQ3** What is the effect of selecting and adapting algorithm while solving an optimization problem?

1.3 Solution overview

In this thesis we propose the unification of both ASP and PSP into a single problem. In order to do so, we firstly introduce a generic runtime PSP solution; secondly, we suggest joining several PSPs search spaces into a united APSP. The consequence of merging several PSPs into a single APSP is the appearance of 'sparse' search spaces, where the percentage of properly defined configurations is low due to the requiring and prohibiting dependencies among parameters (for instance, each algorithm has its own required set of parameters).

To overcome the sparseness issue we propose a complex solution, which is spread in both search space structure and prediction process. For the APSP representation we suggest using of a data structure, similar to feature trees from software product lines field. Doing so we treat a solver type and its hyper-parameters uniformly as a regular parameter in a search space. The dependencies between parameters are explicitly handled in form of a parent-child relationship. As a result, the search space could be viewed as a layered structure, where on the first level remains (categorical) parameter defining the algorithm type, and on the level(s) below settle its respective hyper-parameters (categorical and numerical). The prediction process is made sequentially for each level, utilizing the available performance evidences in form of already tried configurations in this optimization session. Therefore, in the united APSP we firstly build a surrogate model for the algorithm type prediction. Afterwards, when the solver type is selected, we filter the performance evidences to operate on data, which is relevant to the selected algorithm type. With this filtered data we build a surrogate for the second level and predict the selected algorithm parameters. Dependencies among algorithm-related parameters are also treated in form of the parent-child relationship, therefore, we proceed the level-wise prediction process until obtaining a completed configuration. Next, we continue solving the underlying OP with the defined algorithm type and its parameters to obtain new evidence and repeat configuration prediction process. This reinforcement learning techniques enables us to solve the APSP online, while iteratively tackling the OP at hand.

The structure of this thesis is organized as follows. Firstly, in Chapter 2 we refresh the reader's background knowledge in the field of optimization problems and solver types, focusing on heuristics. We also review the parameter setting and the available solutions for this problem. In Chapter 3 one will find a description of the proposed approach for generic parameter control and APSP problem unification. There we also present both structural and functional requirements for system components. Chapter 4 is dedicated to the review of implementation details, including a code basis selection, aforementioned requirements realization and the developed system workflow representation. We evaluate the proposed concept and discuss the results in Chapter 5. Chapter 6 concludes the thesis and Chapter 7 describes 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.

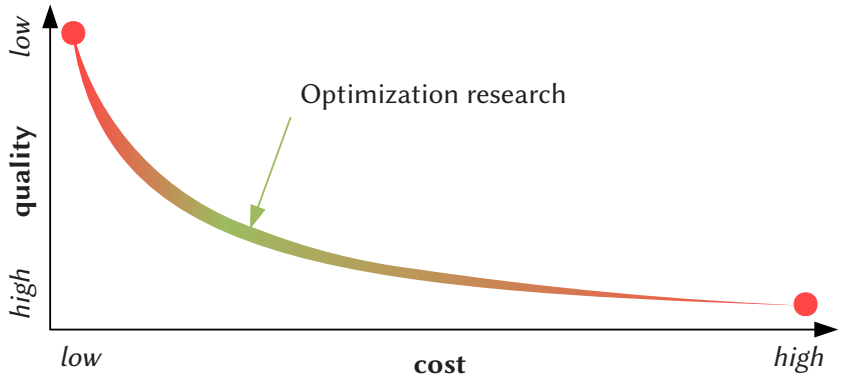


Figure 2.1 Optimization trade-off.

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 [51].

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.

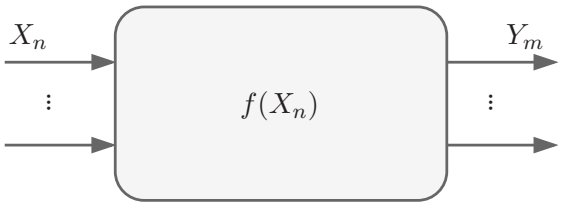


Figure 2.2 Optimization Target System.

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*.

Next, let us highlight the target system characteristics. In works [2, 14, 32, 46] 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 $X \rightarrow Y$ 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* [82], *job-shop scheduling* [17] or *vehicle routing* [113] optimization problems. All combinatorial problems are *NP-Complete* meaning they are in both *NP* and *NP-Hard* complexity classes [48]. 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 [73] 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 [27]. 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, 43, 63]. 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 [120], 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 [47], the objective of an exact algorithm is to perform much better (in terms of running time) than the exhaustive search. In both works [47, 120] 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 [120] 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* [100] 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 [48]) 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 [119]. 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 [119] 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 [119]:

- **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 [119].

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 [90].
- **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 [90].

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* [118]. 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 [119].

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 [107] 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. The static part of the algorithms is stable and problem independent, it forms the core of an algorithm and usually exposes *hyper-parameters*, which could be used for the algorithm configuration. The changeable parts are domain-dependent and should be adapted for problem at hand. 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 [117], biology [103], ethology [34, 102, 110], and even evolution [11, 39].

Meta-Heuristics Classification

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

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* [81] or *tabu search* [50] do exist. The second one allows large

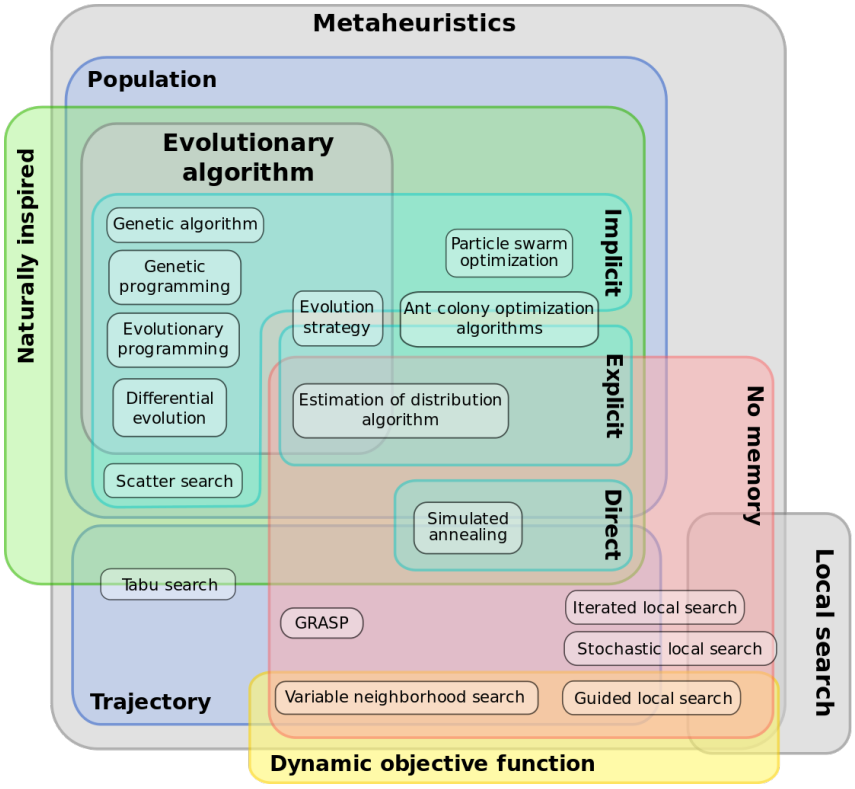


Figure 2.3 Meta-heuristics Classification [36].

jumps in the search space, where the examples are such MHs as *variable neighborhood search* [54] or *simulated annealing* [69].

- 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 [39], ant colony optimization [34] or particle swarm optimization [67] 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 we did not mention.

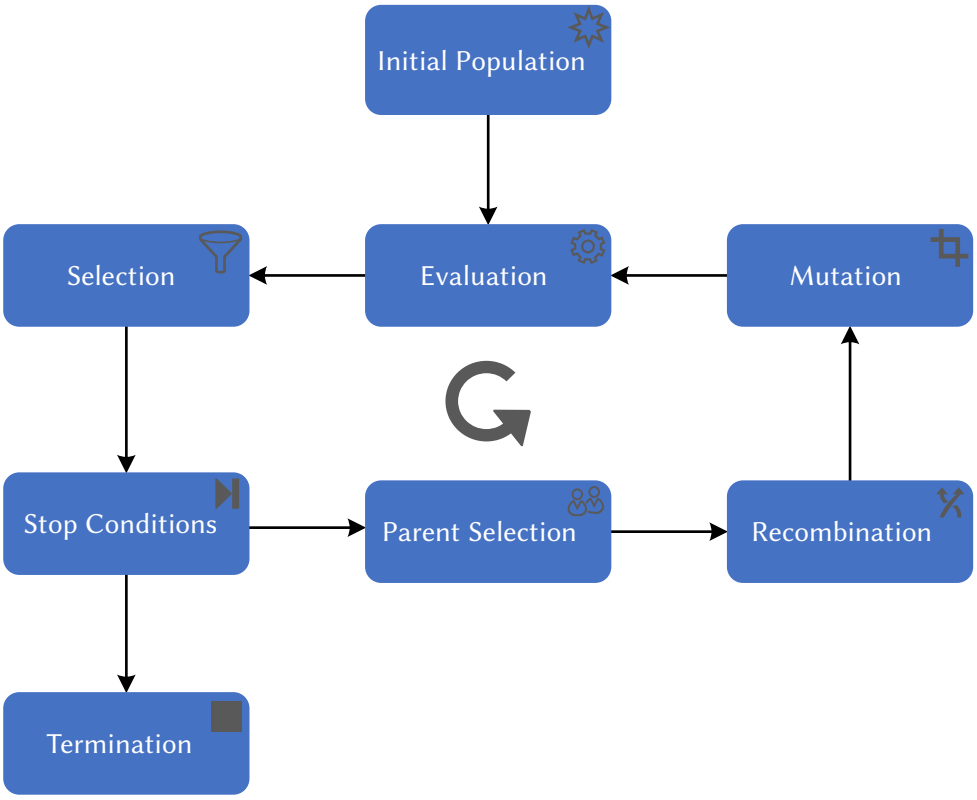


Figure 2.4 Evolutionary Algorithms Workflow.

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 [39].

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* [103], *genetic/evolutionary programming* [72], *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 [38]. 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* [38]. 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 [117]. 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 are the 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, while performing the depth-first search over these sub-neighborhoods, it ignores those, which do not make any improvements. 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 [114].

Direct Global and Local Search. This hybridization consists of two stages: the stochastic global coarse pre-optimization and the deterministic local fine-optimization. At the first stage, the authors apply one of the two abovementioned meta-heuristics — genetic algorithm or simulated annealing [56]. 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 [109].

Simulated Annealing and Local Search. After the brief explanation of previous two hybrids, it is not so difficult to guess, how the next hybridization works. The authors in their work [83] called this method 'Chained Local Optimization'. Therefore, it is a yet another representative of staged hybridization. Iteration starts with the current solution perturbation, called *kick* in [83], referring a dramatic change of a current position within the search space. Afterwards, the hill climbing algorithm is applied to intensify the obtained solution. After reaching 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 [89]. EMILI is a solver for permutation flow-shop problems (PFSP), also known as flow shop scheduling problems [96]. 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 called IRACE [79] in order 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

At this point, an obvious question may arise: "If there already exist excellent and well-performing heuristics, is there any need to put an effort in developing new algorithms instead of using existing ones?" The answer to this question is quite simple — the perfect algorithm suited for *all* OP does not exist and cannot exist at all. The empirical research has shown that some meta-heuristics perform better with some types of problems, but are less-performing 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 at different stages of the 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) [121].

In fact, one cannot 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 the optimization problem is solved, one probably would not try to solve it for the second time. Here, the *hyper-heuristics* come into play to intelligently pick heuristics suitable to problem at hand. We will proceed with their description outlining the way they deal with the NFLT consequences in the following section.

2.2.5 Hyper-Heuristics

Many of state-of-the-art heuristics and meta-heuristics are developed in a complex and very domain-dependent way, which causes problems with their reuse. It motivated the research community to raise the question of a generality level 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 when applied separately [35]. This behavior can be explained by the way of how the search process evolves in time. When one applies 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 one to those previously unreachable zones (2) in more rapid ways. However, usually it is hard to predict how one

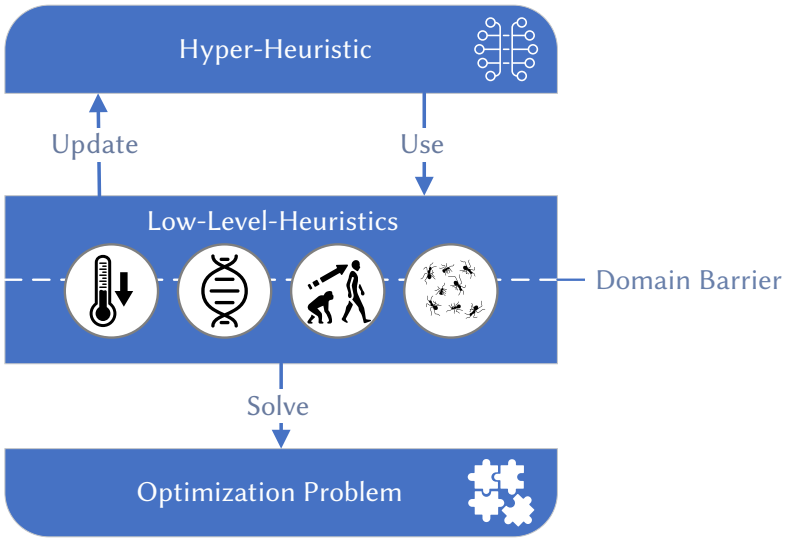


Figure 2.5 Hyper-Heuristics.[a]

[a]Icons are taken from `https://thenounproject.com/`

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 [86] the authors made an infer that HHs can be viewed as a form of reinforcement learning (RL), which is a logical conclusion, especially if rephrased 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). As was stated 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 data types, relationships, etc. within a domain. This information is rather encapsulated in LLHs, therefore, HHs could be applied 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 going on, many algorithm instances were already created and some trials to organize approaches were conducted in [22, 35, 68, 101].

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

In this section we unite of those important hyper-heuristics classification facets in order to better justify the goal of this thesis.

To begin with, there are two broadest classes, which differentiate HH *routine*, also called *a nature of high-level-heuristic search space* [22, 23, 35]. The first class consist of hyper-heuristics to *select* low-level-heuristic, in other words *selection hyper-heuristic*. Please note, previously in this thesis all discussions of HHs were referencing to this specific 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 of LLHs) based on the available prior knowledge and apply it to the OP underway. 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, 35] 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 *learning time*. From this perspective hyper-heuristic can be of type *online*, *offline* or *not learning* algorithm [22, 101]:

- In **online** case, the HH derives an information, used to select among LLH, while those LLH are solving the problem.
- In **offline** case, the learning happens before solving a certain OP. Here one should first train the HH solving other homogeneous problem instances by underlying LLHs (offline learning phase). After that, the HLH will be able to properly choose among LLHs, therefore, be applicable to problems at hand (online use phase). Note that this approach also requires a creation of meta-features extraction mechanism and its application to every optimization problem.
- 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 seem to be a weak approach, however, there exist meta-heuristics, which are similar to HH and perform well (variable neighborhood search).
- There also exist **mixed** cases, where the learning happens firstly in the offline and later also in the online phase. Definitely, it is a promising (in terms of results quality) research direction, despite its high complexity.

For 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, 35, 101].

Hyper-Heuristics Instance Examples

Hyper-heuristic for integration and test order problem (HITO). HITO [52] 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 [37] 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 [84] is an online 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 online 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 [87] 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 [35, 85, 101] 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** [108] 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** [116] 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** [92] 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 basic level 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 problems). However, some adaptations to understand a specific problem and parameter tuning for better performance still should be performed.

Along with MHs exist hybrid-heuristics. In short, they simply just took some survival abilities from several meta-heuristics with a hope 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.

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

Therefore, 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 specific problem using those best suited heuristics. Imposing such great abilities, hyper-heuristics tackle not only the certain 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. In short, it could be a heuristic, machine learning algorithm or any other system.
2. **Parameter** is one of the configuration 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.). Changes in their values dramatically affect the network's performance and results.

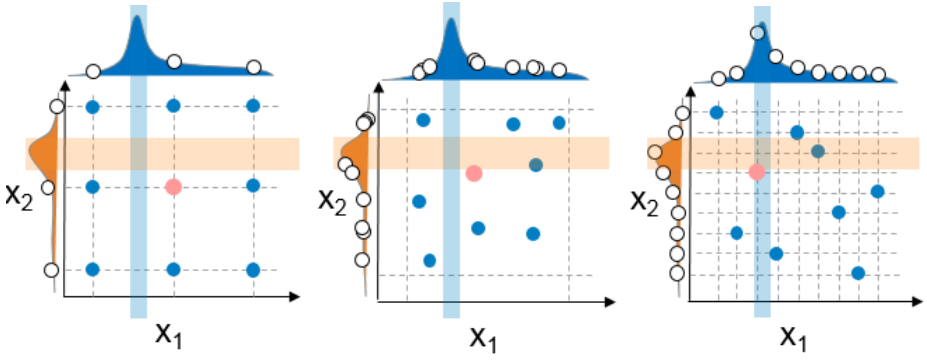
A frequently tackled optimization problem is a **parameter settings problem** (PSP): the process of searching hyper-parameter values that optimize some characteristic of TS. When talking about NN example, PSP could be defined as task of network's accuracy maximization with a given dataset, resulting 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 [74].

Up until now, there were formalized many approaches for solving task of settings hyper-parameters. One way is simply to the use intuition and to apply the parameters that seem more or less logical for a particular system and a problem instance. This error-prone technique was quickly abandoned 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: one is an offline *parameter tuning*, which is performed at the design time and the other is an online *parameter control*, performed at runtime.

2.3.1 Parameter Tuning

Roughly speaking, the offline 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.



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

Figure 2.6 Automated Parameter Tuning Approaches.

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. Hence, 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 a first sight, it might seem 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) [25, 29, 30]. 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.

Model-Based Search Parameter Tuning. In 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

[1]Original graphics are taken from [70]

it is shown in Figure 2.6c, after accumulating enough information the learning algorithm starts making more precise guesses, which in contrast to previously outlined model-free approaches are desirable and more robust.

Naturally, this optimization problem could be tackled by almost every approach discussed in this thesis. 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 aforementioned heuristics directly, since they require performing 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 is 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 the 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 certain crossover type, one will require providing parameters for this specific crossover type, as well as 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. In detail, 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. Please 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 one 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 specific 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. Secondly, a potential tuner should be able to tackle dependencies and conditions among parameters.

SMAC

Sequential Model-based Algorithm Configuration (SMAC[1]) [58] 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 online 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 to parameter tuning but also to optimize any expensive black-box function.

The development of this system was motivated by the limitations of all existing SMBO approaches. Among them an expanding the applicability to categorical, but not only to numerical parameters. Also, to reduce a variance influence the target algorithm performance optimization may be performed on a number of problem instances (benchmark set), instead of a single instance.

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 regarding 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) [105]. The EI is a measurement of possible solution quality improvement obtained by an underlying configuration, therefore, higher EI means that the candidate is better. ILS starts 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

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

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 [62].

IRACE

IRACE[1] is a hyper-parameter tuning package [79] 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 a realization of BOHB algorithm [41] in the software framework. While SMAC outperforms and partially reuses the decisions made in ParamILS [59], 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 naive 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].

A central part of BOHB, namely the Hyperband algorithm, is a promising bandit-based strategy for hyper-parameter optimization [77], in which the *budget* for entire parameter tuning session is

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

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

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 evaluation, while the minimal budget gives the least accurate approximation of configuration results. 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. Secondly, at each HB iteration instead of random sampling, the TPE model is used to pick next configurations. Please 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 the 2^{nd} version of BRISE[1] [93], since the very early BRISE versions (major version 1 [94]) 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 justifying the central terms used in the system. Please note that some of them are similar to the one defined above, but they were explicitly implemented in form of interacting entities.

- **Experiment** encapsulates the information about a certain 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), 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, contains number of *Tasks*. Naturally, the configurations are comparable in terms of the results averaged over performed runs.

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

- **Task** is a single evaluation of TS under provided configuration for specified in description problem scenario.

- **Parameter** is a meta-description of certain 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 other hand, the parameter control approaches are able to solve 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 at the very beginning of a problem solving process, but struggles in the later stages. 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 an *online* 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 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 [57].

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 [64] the authors proposed an adaptive strategy to change the cooling rate, based on the statistical information, evaluated on each optimization step. Specifically, if the statistical analysis named in research a *heat capacity* shows that the system is unlikely to be trapped in the local optima, the cooling rate is increased. On the contrary, it is decreased if the possibility of being trapped is high.

In [49] 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, 31, 60, 80, 111]. In many cases, the authors conclude that 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 EAs [66]. The motivation for such a number of performed studies lays in a strong dependence of an algorithm's performance on parameter values.

The deterministic and adaptive mechanisms in EAs are extended by the 3^{rd} type — a *self-adaptive* approach. It implies an encoding of parameter values in the solution genomes, allowing them to co-evolve with the solutions at runtime [33]. 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 *population size*, *selection strategies* and *variation aspects* (namely crossover and mutation operators). In case of interest, we propose the reader to analyze recently conducted reviews and researches dedicated to parameter control in evolutionary algorithms [1, 33, 66, 106].

There is one rather intriguing parameter control approach proposed for the evolutionary algorithms. In [65] the authors introduce a reinforcement learning (RL) parameter controller, whose goal is to select the EA parameters online 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 adheres the 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 a central part of this methodology — the knowledge. This set of proposed observables could be split into 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 [65] parameter control methodology may be 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 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 parameter tuning techniques. Secondly, the resources spent for offline 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 offline settings quality spending less resources. Unfortunately, nowadays the online 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 the ML framework Scikit-learn [91], Auto-sklearn system [44] operates over a 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 [112] system. Intuitively, it could be rephrased as follows: "For a given optimization problem, find the best performing algorithm and its hyper-parameters among available and solve the problem".

Please note, to the certain extent Auto-sklearn is similar to HHs, which use LLHs for traversing the search space and solving the OP. However, the automatic machine learning techniques operate on the completion algorithms, the results of which are evaluated at the end of their run. On the contrary, online HHs are able to evaluate the intermediate performance of low-level heuristics, since they are anytime algorithms. CASH problem seems to be a combination of problems solved by HHs and parameter tuning approaches. We also found that the architecture search problems [40] (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 offline and online learning, resulted an exceptional performance of Auto-sklearn in classification tasks.

During the offline phase, for each of available datasets published by OpenML community [45], search of the best performing machine-learning pipeline was done using the BO technique implemented in the discussed previously SMAC framework [58]. After that, the *meta-learning* was executed to derive the meta-features for each dataset.

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

During the online phase, another crucial improvement was introduced. Usually, while searching the best-performing pipeline, a lot of effort is spent in order to build, train and evaluate intermediate pipelines. After each evaluation, only the results and the 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 the problem at hand together. This means that the results of this architecture search is a set of models with different hyper-parameters and preprocessing techniques, rather than a single model. This ensemble starts from the worst performing ones (obtained at the beginning of the search) and ends with the best suited model for the respective dataset. Naturally, each ensemble member's influence on the final results is weighted.

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

In general, an empirical investigation of Auto-sklearn's universality would be rather intriguing, since the only cases of Auto-sklearn application we managed to find are the classification tasks but not regression problems [13, 44].

The field of automated machine learning is one of trending research directions, that is why there exist dozens of open-source systems, such as *Auto-Weka* [112], *Hyperopt-Sklearn* [71], *Auto-Sklearn* [44], *TPOT* [88], *Auto-Keras* [61], etc. Among open-source, there are many commercial systems, such as *RapidMiner*, *DataRobot*, 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 focusing on the heuristics. 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 derives from the exploration-exploitation balance and strength, revealed in a particular case. It is difficult to guess beforehand, which heuristic 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 the values of its 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, based on experience and intuition, (2) utilize the parameter tuning systems, which find the best values automatically and later use those found parameters, or (3) exploit the parameter control mechanisms. Among all strategies, the last seems to be the happy medium, since

tuning requires lots of expensive algorithm executions to produce a good parameter settings, while manually choosing hyper-parameters is an error-prone process that requires experienced guidance. The analysis of parameter control approaches showed that the existing techniques are heuristic-dependent, therefore, our first research question is defined as follows **RQ1** *Is it possible to perform the algorithm configuration at runtime on a generic level?*

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

The research in automatic machine learning has made a step further and tends to combine both algorithm selection and parameter tuning problems into a single CASH problem, formalized in [112]. The search space in CASH problem is formed of algorithm variants and their respective hyper-parameters. However, one solver cannot use the parameters of another, thus, the resulting search space happens 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 solvers for CASH problems are the parameter tuning systems: SMAC in Auto-sklearn and Auto-Weka, Hyperopt in Hyperopt-Sklearn and so forth. Not many surrogate models are able to handle the sparse search spaces: random forest machine learning model and Bayesian optimization approaches with exotic kernel density estimators [75]. 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 offline manner.

We believe that the solutions of both algorithm selection and parameter setting problems is highly dependent on a problem at hand. That is why a search for the best tool (solver) and its setting (parameters) should be performed in online manner, in other words, while solving the optimization problem. Since the generic parameter control concept was not proposed yet, naturally, we were not able to find the techniques to merge and solve both the algorithm selection and parameter control problems in runtime. Therefore, we defined our second research question **RQ2** *Is it possible to simultaneously perform algorithm selection and parameters adaptation while solving an optimization problem?*

As CASH merges algorithm selection and parameter tuning techniques to get the outstanding performance, we found a merge of online algorithm selection and parameter control an intriguing and worth-to-try idea. However, the amount of spent resources and the imprecision of surrogates estimations for simultaneous search of both the algorithm type and its parameter values may be discouraging. To explicitly evaluate this, we define the final research question as **RQ3** *What is the effect of selecting and adapting algorithm while solving an optimization problem?*

Research objective defined.

In this thesis we are trying to achieve the best of both online algorithm selection and parameter control worlds. The resulting approach should be able to solve an optimization problem, applying *the best suited low-level-heuristic* and *setting its parameters* at *runtime*. With this idea in mind, we investigate a possibility of turning existing parameter tuning system into an **online selection hyper-heuristic with parameter control in low level heuristics**.

3 Online Selection Hyper-Heuristic with Generic Parameter Control

While there exist no universal approach to control the parameters of the algorithm (Section 2.3.4), our conclusion was that there exist no approach to combine both online techniques for the algorithm selection and the parameter settings (Section 2.5).

In this Chapter we propose the approach to solve this problem, excluding the implementation details. In Section 3.1, we introduce a generic parameter control technique and expand it with the process 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 online algorithm selection and parameter settings, and we solve it on two levels: 1) in the search space structure and 2) in the prediction process. In Section 3.2 we present a joint search space of both algorithm selection and parameter control problems. We outline functional requirements for such search space. Next, we describe a related prediction process in Section 3.3. We decouple the learning models from the search space structure and provide a 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) — workhorses of our approach. We highlight the requirements to LLH that are crucial within the scope of this thesis.

3.1 Combined Parameter Control and Algorithm Selection Problem

The basic idea of 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 in the heuristics review (Section 2.2), the algorithm performance is highly dependent on the provided exploration-exploitation balance, which in its turn, depends on (1) the algorithm itself and (2) its configuration. The task of parameter control is to find such parameters, which provide the best performance.

In our work, we solve the parameter control problem by utilizing an approach for evolutionary algorithms similar to the one proposed in [65] reinforcement learning (RL). 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 to the parameter control problem, we should define what those *actions* are and how to estimate the *reward*. Therefore, in order to make the parameter control applicable to a broad range of algorithms, we analyze not the solver state itself but the optimization process (in [65], the authors use both algorithm-dependent and generic metrics). To realize the MAPE-K control loop, we should 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 $TTS \gg TTC$, therefore, we are restricted to use-cases with large TTS w.r.t. TTC.

To evaluate the gained in iteration i reward, instead of using the solution quality straightforwardly, we calculate the *quality improvement*, obtained with the 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 could confuse the learning models and, therefore, cause the prediction quality to struggle. To solve this issue, the relative improvement (RI) of solution quality is calculated:

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

In Equation (3.1) 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 a configuration for the next iteration C_{i+1}. Please note, here we utilize the notion of *sliding learning window* to follow a possibly changing trends of optimization process, therefore, we use only N (pcs., or %) of the latest available $C_i \rightarrow RI$ pairs. Moreover, we made two other decisions for the sampling process: (1) hide the search space shape and (2) use the surrogate models for finding configurations that lead to the highest RI. After sampling the C_{i+1} configuration, we set it as the solver's parameters. To proceed with the solving process, we initiate the solver with the solutions obtained at $i - 1$ iteration as well.

When it comes to the algorithm selection problem (discussed in Section 2.2.5), we treat the solver type itself as a 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 becomes sparse and requires a special treatment. There exist two commonly used approaches for tackling this problem. The first [41, 58, 93] requires special type of learning models, while the second [79] 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 most of the reviewed systems follow strictly the first idea. For instance, the surrogate models in BOHB [41] and BRISE [93] use the Bayesian probability density models. Those surrogates could naturally fit the sparse search spaces (described in the following section), but the proposed approaches are not able to make the predictions effectively, since most of predicted configurations will violate the dependencies. As an example, let us imagine after i^{th} iteration, the surrogate models learn about two superior parameters: one indicates a well-performing heuristic type (genetic algorithm), the other — an effective configuration for another algorithm type (an exponential cooling rate for simulated annealing). In this case, the reviewed systems sampling methods will tend to predict invalid configurations with those two parameter values.

In this thesis we adapt the second approach of problem transformation used in [79] for sampling the valid configurations only. The following section depicts a required preparation step made in the search space, while Section 3.3 is dedicated to the prediction process.

3.2 Search Space Structure

When time comes to selecting not only the solver parameters but also the solver itself, the united search space can no longer be presented as a 'flat' set of parameters, since it tends to produce a vast amount of invalid parameter combinations. Let us estimate the number of all possible configurations in comparison to the amount of meaningful ones. Supposing we have the N_s number solvers, 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 a 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).

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

For the better intuition, let us try some numbers. By setting all $N_s = N_{s,p} = N_{s,p,v} = 3$ (a rather small example), the number of configurations estimated separately for each solver equals to $N_c = 81$ (Equation (3.2)). However, if we join the parameter spaces of all three solvers, Equation (3.3) shows a significant growth in the search space size: $N_c = 19683$. Please note, the number of *unique and valid* 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 struggle.

To overcome this problem, we utilize a certain idea, similar to the one used in IRACE [79]: *explicitly indicate the dependencies as parent-child relationships among the search space entities, firstly predicting the parent parameter, and the children parameter afterwards.* This gives us an opportunity to treat the algorithm type as a regular categorical parameter, making the search space structure uniform and simplifying the prediction process.

This decision sets the following search space *structural requirements* (S.R.):

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

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

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

Figure 3.1 shows an example of such a 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.

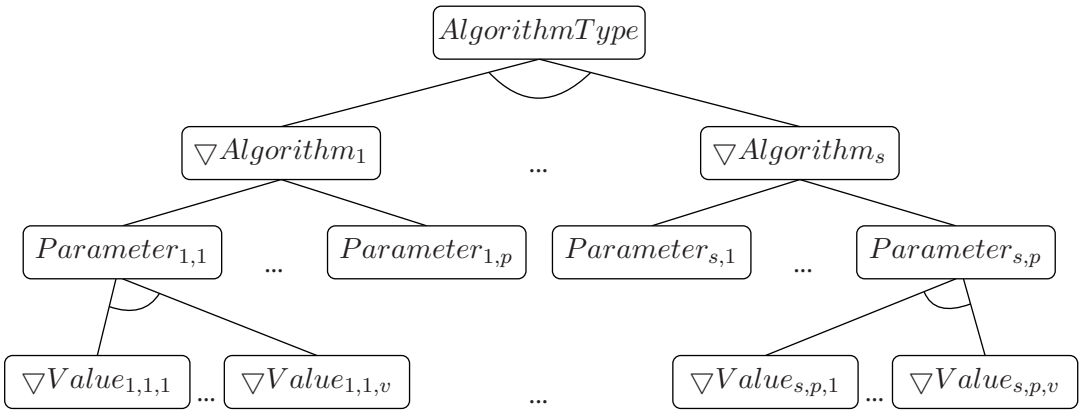


Figure 3.1 Search space representation.

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).

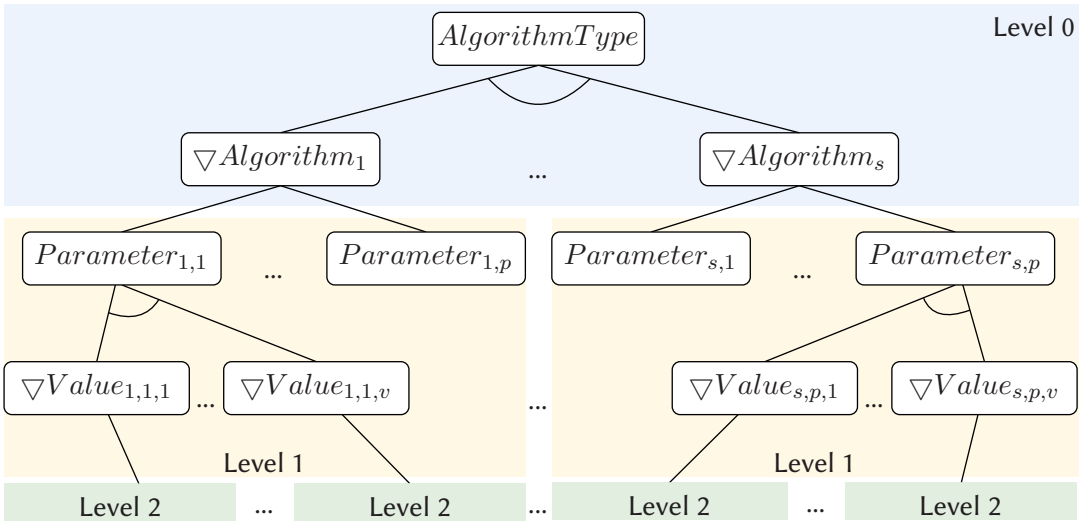


Figure 3.2 Level-wise prediction process.

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. In addition, when we predict on the parent level, it will not change on the descendant levels, therefore, 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 named *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 results in different outputs). In 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, while solving the problem, 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 else, the previously best-performing parameter values are not good enough and should be replaced by the other. These changes may be caused by the variety of 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 for search space (S.F.R) and prediction process (P.F.R).

- **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 the 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.

Yevhenii: check listing fit into single page

3.4 Low Level Heuristics

As we stated in Section 2.2.5, hyper-heuristics are built of two main components — the high level heuristic (HLH) and the low level heuristic (LLH). Please 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 **offline 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 **online 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 initialized with the obtained in previous external iteration solutions.
5. The **online 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.

Please note that the offline cases estimate the solution quality directly, while the online cases use the relative solution quality improvement.

It is worth mentioning that the proposed structure of search space representation is similar to the *feature model*, used to describe the software product lines (SPL) [104]. In Figure 3.1 and Figure 3.2 we used the notions from SPL feature models to denote *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 relies on an implementation effort minimization with help of already existing and well-tested code reuse. 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 the desired RL-based hyper-heuristic. We also reuse the low level heuristics (LLH) implementation from other meta-heuristics frameworks. The LLH basis may be used almost out-of-box, but the HLH basis requires changes to utilize the reinforcement learning approaches.

In Section 4.1 we analyze the parameter tuning frameworks from a perspective of required adaptation effort to implement listed in Section 3.3 HLH characteristics. We conclude the analysis selecting the best suited HLH code basis in Section 4.1.2. Afterwards, we split the HLH adaptation process on two similar to presented during the concept description logical parts: 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 code basis selection for LLH, present a set of reused meta-heuristics, their adaptations and the importing process into our hyper-heuristic.

4.1 Hyper-Heuristics Code Base Selection

To begin with the analysis of frameworks, let us firstly outline important characteristics from the implementation perspective.

The first two crucial criteria are *variability* and *extensibility* of framework. Since we are planning to use a possibly (but not obligatory) different models for the LLH selection and parameters control, the code basis should be variable in terms of models usage for each prediction level (see Figure 3.2). The desired HH and specifically HLH should be easily extensible in terms of not only the surrogate models, but also other features such as termination criteria, information filtering, data preprocessing and so forth.

The next criteria is the support for *online optimization*. It is a slightly complex system characteristic, which we are willing to distinguish. As it turns out, many parameter tuning systems require full evaluation of target system, other expose early termination mechanisms, but all of them are forcing TS to start solving of the problem at hand each time from scratch. In case of our hyper-heuristic, we treat the LLH configuration evaluation as a trial to improve the problem solution results. It implies an important LLH ability to tackle the (OP) using reinforcement learning approaches (Section 3.1).

The final characteristic is the *conditional parameters* support. This complex feature influence on not only the search space representation mechanisms, but also the prediction process. Therefore, we pay a close attention to both of them from the conditional parameters support perspective.

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 the ROAR mechanism with a model-based sampling algorithm. ROAR is a derivative from the FocusedILS

algorithm (solver in the parameter tuning framework ParamILS [59]), where each evaluation of a new solution candidate on a problem instance was performed sequentially and in isolation. Therefore, since the ROAR evaluation strategy is also used in SMACv3, we found it to require a considerable effort for synchronization the progress between different runs, which enables the online problem solving.

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 fit to complex dependencies among parameters. In the next configuration sampling process the *one-exchange* neighborhood of the best found so far configuration is traversed using the created 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. Since, to the best of our knowledge, the one-exchange neighborhood is unaware of the dependencies, it violates them while sampling and results in illegal parameter values combinations. Those cases are naturally controlled and rejected by the search space representation framework ConfigSpace [78], but we believe in case of sparse search spaces it could lead to ineffective sampling and system predictive abilities struggling. Unfortunately, we did not find any officially published empirical studies of such cases and could only make guesses based on own intuition, but SMACv3 developers advises for such use-cases[1] may serve as an evidence to our assumptions correctness. One of the possible solutions here can be the implementation from scratch of a conditional-aware one-exchange neighborhood definition and for sampling process, which requires much implementation effort.

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 opened source code.

The framework uses a *Friedman test* [28], or as an alternative *paired t-test* for statistical analysis of racing in parallel configurations. As the surrogate models, IRACE uses the probability distributions of those parameter values, which are proved to be good during the racing step. The configuration prediction process is defined as the step-wise sampling on previously constructed distributions. It elegantly handles the conditions among parameters and illuminates a possibility of invalid configuration appearance. Unfortunately, this solution is static in terms of variability and extensibility on the learning mechanisms.

From the perspective of parallel evaluations, the framework utilizes all available resources at the beginning of each racing step, but as the process continues, fewer evaluations are executed simultaneously, therefore, part of available resources is idling and not utilized completely at all stages of IRACE execution.

For the online problem solving support, let us discuss the racing algorithm. As mentioned in Section 2.3.2, this step is executed on (1) a set of TS configurations sampled for evaluation and (2) a benchmark set of optimization problems. Multiple instances of TS are initializing with the provided configurations and starting to solve the problem set, while the racing algorithm terminates the worst-performing settings. In case of hyper-heuristic, it is possible to define the benchmark set as a single problem instance divided into parts of TS running time. At each pause we may perform the synchronization of current solutions to proceed with the best found results. Doing so, we adapt the system to online problem solving cases, 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

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

are reported leaving the performance evidences of others behind, but we believe this information may be used to create a more precise surrogate models.

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.

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 each algorithm weak sides 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. On the contrary, a drawback of BO TPE lays in a configuration evaluation process. Naive Bayesian optimization approaches do not take into account the TS performance early evidences. Thus, even when the proposed configuration results in a poor TS initial and intermediate performance, which may be an evidence of a weak final performance, BO still continues the TS execution. These facts motivated authors to merge those two algorithms and 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 robust learning mechanisms (BO).

Let us discuss the process of handling the conditions between parameters. HpBandSter as well uses ConfigSpace framework for the search space representation. As we discussed during the SMACv3 review, ConfigSpace naturally allows to encode the dependencies and conditions among parameters. TPE learning models are also able to fit these dependencies by means of implemented *impuration* mechanism [75]. Shortly saying, to fill 'sparse' configurations with data, the disabled parameters are replaced with their default values. Later, while building the surrogate models those default values are ignored, therefore, the probability density estimations still represent a proper parameter values distributions. However, consider a case of two configurations families appearance: C_1 and C_2, such that some parameter P_i is forbidden in C_1 but required in C_2. On the contrary, another 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 utilized in HpBandSter 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. The possible treatment here is to change the sampling process by an intermediate layer addition, which will perform the 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. The provided surrogate models here 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 linear regression model with regularization [55], often used in machine learning field. Being a linear model, its abilities to fit sparse search spaces is poor and, therefore, a machine learning

community suggested treating such cases with *conditional linear regression models* [24]. The underlying idea is to split the search space into sub-spaces and to build separate regression models in each of them. Unfortunately, this approach was not built-in into the ridge regression model used in BRISEv2.

As for the online problem solving support, the routine of an optimization process implemented in BRISEv2 is very similar to 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 facilitates the implementation of problem solving with runtime system adaptations, presented in Section 3.1.

4.1.2 Conclusion on Code Base

Most among the reviewed parameter tuning systems share the same SMBO approach for problem solving. They utilize a rather similar techniques for the surrogate models creation and predictions making, however, the different system architectures are implemented. To sum um our review, we utilize the term *quality* to aggregate both (1) provided out-of-the-box the desired characteristic support and (2) the required effort to adapt it, if necessary. For the visual representation, we collect the reviewed characteristic qualities of each software framework into Table 4.1 and quantize them into three ordinal values:

1. **Poor** quality denotes a weak characteristic support and much effort required to improve it.
2. **Average** quality indicates a weak characteristic support, which requires less amount of effort to provide it.
3. **Good** quality means a good characteristic support out-of-the-box 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
Online optimization	Average	Average	Average	Good
Conditional parameters	Poor	Good	Poor	Poor

Most of the reviewed software systems were created as an implementation of some concrete algorithm (or combination of algorithms), which results in a system flexibility reduction. Every reviewed framework requires much adaptation effort and the preparation steps should be performed in different parts of a system. Between such features as a proper support of conditional parameters and variability-extensibility, the former plays a settle role in our case. Therefore, we conclude the BRISEv2 framework 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 relationships should be presented explicitly to allow combinations of different parameter types. For the prediction process support, 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 analyze the available ConfigSpace framework, how it fits to our requirements and 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 manner. In other words, there is no step-wise process of configuration construction, but only the final and valid results are produced. To the best of our knowledge, there is no straightforward way to expose the underlying parent-child dependencies among parameters and investigate a tiered search space structure, which is required for the prediction models. 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 partial one (flat validation). It is also worth to mention that the used in ConfigSpace configuration is 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 mostly motivated by the amount of required adaptation effort and partially by an involvement of obligatory external dependencies.

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 structure is 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 a numerical parameter family, nominal and ordinal as a categorical family. This fulfills specified in Section 3.2 S.R.2.

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

Search space construction. The S.R.3 (parent-child relationship) 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 should expose the child.

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

Listing 4.1 S.R.1 implementation.

Please note, currently we require the support of composite construction only by means of categorical parameters, therefore, *add_child_hyperparameter* requires a list of activation categories. We postpone an enhancement of composition on numerical ranges for the future work.

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

Search space role in prediction. Imagine several configurations were evaluated and their relative improvement is already estimated. For making the prediction in a tiered approach, the parameter values on a current level should be selected before moving to the next one. For it, we firstly filter data, which fits to this level by means of S.F.R.1. It is implemented in form of recursive hyper-parameter method *are_siblings*, presented in Listing 4.2. The filter accepts already selected 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 configuration's under comparison parameter values (*target_values*). 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
  ...
```

Listing 4.2 S.F.R.1 implementation.

After data filtering the time comes to find out, the values of which parameters we must predict. For doing so, the search space, must expose those parameters by means of S.F.R.2 implemented in *generate* method with signature presented 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 $parameter\ name \rightarrow parameter\ value$ 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
  ...
```

Listing 4.3 S.F.R.2 implementation.

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

The surrogate models creation requires an available data (parameters) description. Thus, the S.F.R.3 implementation is performed in method *describe* with signature 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 is a mapping from the parameter name to its type and 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) -> MutableMapping[Name: [Type, Values]]
  ...
```

Listing 4.4 S.F.R.3 implementation.

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

The described above process is controlled by means of S.F.R.4, implemented as method *validate* (signature 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 not finished

(deep validation), meaning all parameter values were chosen, and we have a valid configuration. Secondly, after making the prediction by models (flat validation), it verifies if the parameter boundaries are not violated. In case of violation, the prediction is discarded and the sampled randomly values are used instead. Since the implemented in search space sampling process (*generate* method) 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.

```
1 class Hyperparameter:
2   ...
3   def validate(self, values: MutableMapping, recursive: bool) -> bool
4   ...
```

Listing 4.5 S.F.R.4 implementation.

4.3 Prediction Process

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

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

As for P.F.R.2, the current implementation of BRISEv2 already provides some level of model unification with a required interface. However, during the implementation we found that it implies three logical steps binding: data preprocessing, surrogate models creation and surrogates optimization to predict a next configuration.

The following parts of this Section is dedicated to (1) P.F.R.1 and P.F.R.3 implementation in form of *Predictor* entity, P.F.R.2 fulfillment in form of the data preprocessing mechanisms decoupling from the prediction models. Please note, we postpone the implementation of an elegant surrogate optimization mechanism for future work. Instead, we utilize 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. Due to a cheap cost of configuration evaluation on surrogate models, we are allowed to do so.

4.3.1 Predictor Entity

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

To implement the information forgetting mechanism, we utilize the similar idea of *sliding window*, used in hyper-heuristics [42]. According to it, the predictor should use a specific 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 4). It fulfills the P.F.R.3. Naturally, more exotic approaches may arise such as a statistical analysis of diversification or the other types of meta-learning, but we leave it for the future work.

The next step is a prediction models decoupling from the search space structure by means off fulfilling P.F.R.1. As discussed in Section 4.2.2, to predict parameter values on each level, the models should be

built on only related to this level information. For this, after filtering the data (Listing 4.6, line 11), predictor propagates the sampling from a previous level to current (line 14) and derives a description for the obtained parameters on current level (line 17-18). Independently, it instantiates a specified in settings surrogate model for this level and fit it with the obtained information (lines 21-23). Afterwards, it requests a prediction from model (will be discussed later) and forwards the prediction for validation by the search space entity (lines 27-28). If either the model cannot properly fit the data, or the prediction is invalid, we keep the sampled randomly parameter values (lines 29-31).

```
1  class Predictor:
2    def predict(measured_configurations):
3      # Filter data according to sliding learning window
4      level_configurations = trim_in_window(measured_configurations)
5      prediction = Mapping()
6
7      # Continue prediction until get a valid configuration (Deep validation)
8      while not search_space.validate(prediction, recursive=True):
9
10       # Filtering the data for current level
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 model
21       data = trim_accodring_to_description(level_configurations, level_description)
22       model = get_current_level_model()
23       model.build(data, level_description)
24
25       # Predict current level and validate prediction (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 the data preprocessing and available surrogate models 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 parameter values probability densities (TPE), one should derive a numerical data by encoding those string values into numbers. 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 (learning complex dependencies in data), variance (generalization to the unforeseen data) or both. An encoding example 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 degrees of available features with an aim to disclose more complex dependencies. The decision on encoding type is often defined by the learning model. On contrary, the decision on data transformation is carried out by the user and depends on the concrete use-case and experience.

All reviewed in Section 2.3.2 parameter tuning systems implement the data preprocessing only by means of an obligatory encoding and omitting the possible data transformation. In most of the cases it is implemented as a simple label enumeration and is 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, it will encode such parameter values to numbers 0, 1 and 2 respectively. When such encoded data is passed to the surrogates for learning, some models may interpret it as follows: GA is closer to SA than to ES, the distances from SA to two others algorithms are equal within a search space. To prevent this, the other type of preprocessing should be used, for instance, binary encoding.

In any case, the intent of this discussion is to provide an insight of data preprocessing importance for the reader, but the discussion of possible cases and their influence are out of this thesis scope. Here 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 [91]. We omit the details of wrapper implementation since it is a single object decorator, instantiated with the provided 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 the models and data preprocessing units interfaces compatible, we store the data in form of DataFrames — tabular data representation carried by 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 tiered prediction process, 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, a previously used in BRISEv2 ridge regression with polynomial features is nothing else, but a combination of data preprocessing step with the ridge regression model from Scikit-learn framework. Later in this Section we discuss an implementation of a unified wrapper for Scikit-learn linear models.

As a step further, we also add the implementation of multi-armed bandit (MAB): a selection strategy proposed in [4]. It is motivated by a promising performance of the reviewed in [4] selection hyper-heuristics built on MAB. Please note, 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, there is no reason for the detained TPE implementation discussion here.

[1] ConfigSpace documentation `https://automl.github.io/ConfigSpace/master/API-Doc.html`
[2] Pandas Github repository `https://github.com/pandas-dev/pandas`

Scikit-learn Linear Model Wrapper

Scikit-learn is one among the most popular open-source machine learning frameworks. As a consequence of flexible architecture, Scikit-learn often plays a central role in other products providing implementations of numerous building blocks for machine learning pipelines. This advantage in combination with a comprehensive documentation resulted into a large and active framework community[1].

All available in Scikit-learn linear regressors implement the same interface and usage routines. For instance, before making a prediction, the regression model should be trained on a preprocessed data, providing *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 for finding the best parameter combination, one should still solve the original 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 creation process includes also a model accuracy verification step, which is performed by means of cross-validation: splitting the set of data into k disjoint folds, training k model each time excluding one fold for accuracy verification (line 9). If the potential model average accuracy is less the predefined threshold, the model is considered to be not precise enough and, therefore, rejected (line 15), forcing the predictor to use 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 by means of random search (if the model was built successfully), we firstly sample parameter values of this level uniformly at random (line 20). Afterwards, we transform them using the same preprocessors, applied during the model construction (line 21). Then, we make a prediction for randomly sampled features using the surrogate model and transform those predictions back into original labels (lines 23-24). Finally, we select the best features (encoded parameter values) by means of the predicted labels, reverse it transformation and return to *Predictor* (lines 27-28).

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

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

    # Build model and check its accuracy
    accuracy = cross_validation(model, transformed_features, transformed_labels)
    if accuracy > 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 surrogates optimization problem by means of random search
    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)

    # Select those parameter values, which maximize RI
    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 [98] 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, sequentially selecting several times among available choices c_i. The problem obtained its name as an analogy to one-hand slot machines in casino and naturally denotes the well-known exploration versus exploitation dilemma.

In most of the times, MAB is solved by reinforcement learning (RL) approaches, which analyze the already available evidences before performing each next step. It perfectly fits to our requirements of sequential LLH to tackle the problem at hand, therefore, those choices are nothing else, but LLH types (categories of categorical parameter). In [4] the authors proposed the Upper Confidence Bound algorithm as an intuitive MAB solution: in iteration k, among available categories 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 and evaluates 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 [76] Fitness-Rate-Average-based MAB (FRAMAB) with two

reasons: (1) it is an intuitive and robust approach, (2) according to the benchmarks in [42] it outperforms 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 [42] 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 as a standard deviation in improvement values. The motivation for this is following: if there exists an uncertainty in category domination, the deviation will be high and it should encourage 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 every algorithm starting from a single heuristic and ending with 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 meta-heuristic but to instantiate a set of underlying heuristics among available in relative frameworks. Thus, in this Section we present a review of several meta-heuristic frameworks with an intent to select the best suited one, implement a facade for framework usage and utilize the available algorithms as LLHs in our hyper-heuristic.

Before diving into description 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.

4. **Warm-startup**, which is required to continue the problem solving from a previously reached solution. The underlying LLH should not only report the finally found solution(s) but also to accept them as the starting points.

5. **Termination criteria**, which is needed to control the intermediate results of optimization process by HH. In our system we use the wall-clock time 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 basis candidates: Solid[1], mlrose[2], pyTSP[3], LocalSolver[4], jMetalPy[5] and jMetal[6].

Solid. A framework for gradient-free optimization. It comprises a 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 it requires changes in each algorithm as a consequence of the shared base class absence. As for the termination criteria, algorithms in this framework support the maximal number of TS evaluations-based and desired quality-based but not the time-based termination. Once again, to add new criterion, one should modify the code of all algorithms. The framework does not provide the problem instances, nor domain-dependent parts of algorithms, therefore, to use it one will need to carry out not only a domain-specific adaptation but also a problem description.

mlrose. A framework with implementation of various well-known stochastic optimization algorithms such as: naive and randomized hill climbing, simulated annealing, genetic and mutual-information-maximizing input clustering (MIMIC) algorithms. Each listed solver is implemented with an 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 criterion in particular.

pyTSP. A system, specially designed to tackle the traveling salesman problem. Together with visualization techniques, it also provides a wide bunch of different algorithms. Here they are divided into four groups. First are construction heuristics with nearest neighbor, nearest insertion, farthest insertion and cheapest insertion algorithms. Second is a linear programming algorithm. Third are perturbation heuristics among which pairwise exchange, also known as 2-opt, node insertion and edge insertion. Fourth group formed from meta-heuristics and 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 provided.

LocalSolver. A commercial optimization tool with free academic license. It is implemented in C++, and 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

[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`

detailed documentation, the desired TSP example could be found among numbers of other optimization problems. 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. Unfortunately, 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 task requiring much implementation 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 to be well-structured, therefore, in case of required changes they could be made with less effort. A functionality for heuristic solver warming-up is available out-of-the-box. The various termination criteria are ready as well, among which the wall-clock time-based and the number of TS evaluations-based criteria. The list of supported single-objective optimization problems consists of knapsack, traveling salesman and four other 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 SO OP, jMetal developers implemented the following algorithms: naive 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 not all solvers are universally applicable to a 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 jMetalPy, jMetal supports only one termination criterion, based on number of TS evaluations, and do not support algorithm warming-up at all.

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 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. For our experiments we decided to use three LLHs: 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 developing both jMetal and jMetalPy, the overall architecture of both frameworks is 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 onto two parts, one is dedicated to jMetalPy and in the other we discuss jMetal.

jMetalPy. During the analysis above we found 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. For doing so, we implement a framework wrapper (see Listing 4.8), which creates a desired optimization problem instance, MH solver instantiate with the provided hyper-parameters (line 3). Later, we call this wrapper to start a solver execution and report the results in a framework-independent way (line 5). To prevent an expensive problem instances loading within one optimization session, we cache it in memory (lines 8-9). 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 11-15).

```
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    def run_and_report() -> Mapping
6
7    # jMetalPy framework introspection helper methods
8    @lru_cache()
9    def _get_problem(problem_name, init_params)
10
11   @lru_cache()
12   def _get_algorithm_class(mh_name)
13
14   @lru_cache()
15   def _get_class_from_module(name, module)
```

Listing 4.8 jMetalPy framework wrapper pseudo-code.

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

jMetal. On the contrary to jMetalPy, this framework is implemented in Java, therefore, we can not perform the software instantiating straightforwardly, since 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 evolution strategy, we pack it into

[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

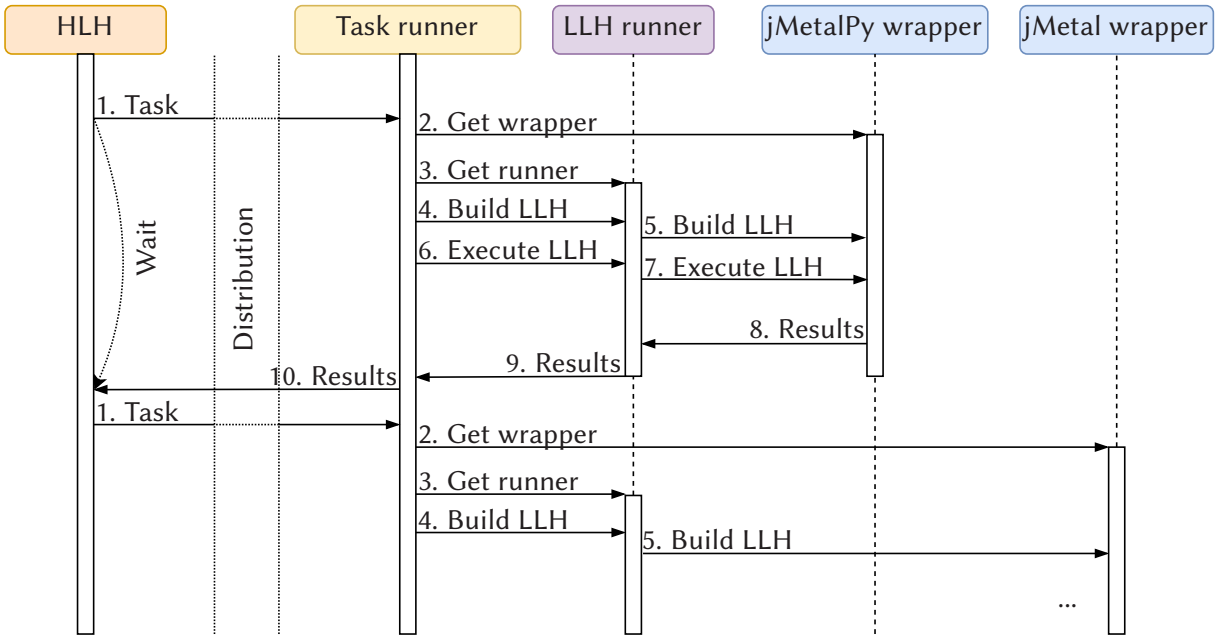


Figure 4.1 The low-level heuristic execution process.

an executable file with exposed parameters and call it from worker script, providing hyper-parameters settings and warming-up solutions.

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*. It forwards the construction and execution commands to the wrapper, tracks the state, make general information logging and pass the results after execution back to HLH. 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. Please note, within an implemented approach the meta-heuristics are reinitialized at each external iteration (between each task execution). Therefore, algorithms as simulated annealing are 'restarting' between tasks dropping such internal parameters as temperature (in SA) to it's initial state.

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 their LLHs. For the LLHs implementation we reuse the existing meta-heuristic frameworks jMetalPy and jMetal. Despite the selected code basis, the proposed approach could be implemented in most of the parameter tuning systems following SMBO methodology, however, requiring the adaptation according to our review in Section 4.1.

While adapting BRISEv2, we were forced to set aside the previously used search space representation and implement our own to handle the tiered configuration prediction process and the sparseness issue. An intermediate entity *predictor* was added to decouple the search space shape from the learning-prediction process. It allowed us to extend the previously available models with the several others: fitness-rate-average based multi-armed bandits (FRAMAB) for categorical parameter selection and linear regressors from Scikit-learn framework as surrogates models. We also decoupled data preprocessing step reusing 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 proposed in Chapter 3 and implemented in Chapter 4 approaches of search space representation, prediction process and based on them generalized parameter control approach, selection hyper-heuristic and the hyper-heuristic with parameter control should be broadly evaluated. The experiments may be performed in a number of investigation directions, starting from the developed RL performance comparison to the baseline and ending with the scalability to different problem sizes.

We start this Chapter with brief presentation of the optimization problem at hand in Section 5.1 and short environment description in Section 5.2. We perform a parameter tuning of low-level heuristics in Section 5.3, which will be used later through our tests for comparison.

The presented in this thesis evaluation could be divided into two main parts.

The first part (Section 5.4) is dedicated to the developed concept analysis in comparison to the baseline. We start the concept evaluation with planning in Section 5.4.1 and proceed firstly reviewing the baseline in Section 5.4.2, afterwards the generic parameter control is presented in Section 5.4.3, followed by the selection hyper-heuristic with static hyper-parameters in low-level heuristics review in Section 5.4.4 and ending with the selection hyper-heuristic with parameter control in low-level heuristics review presented in Section 5.4.5.

In the second part (Section 5.5) we investigate an influence of hyper-heuristic with parameter control settings on its performance. For doing so once again we firstly perform the experiment planning in Section 5.5.1. Afterwards, in Section 5.5.2 we investigate an influence of a learning granularity on the HH-PC performance. In Section 5.5.3 we check learning models configurations and in Section 5.5.4 we verify the influence of inter-LLH communication configuration.

Finally, Section 5.6 concludes our evaluation with a discussion of the obtained results.

5.1 Optimization Problem

Through this thesis we are tackling a vehicle routing problem — the traveling salesman OP. We present its short definition here including the related to benchmark details, however, the detailed explanation could be found in Section 2.1.1: "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. In our 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 in research TSP benchmark set TSPLIB95[1]. The advantage of choosing this benchmark set lays in a broad compatibility of solvers and frameworks (including jMetal and jMetalPy) with this standardized TSP instance description. The TSP in this case is defined as a set of city coordinates. Therefore, before starting to solve a problem, the distance matrix is usually built by heuristic, calculating the Euclidean distances between each pair of cities. For more detailed explanation of the proposed in TSPLIB95 problem instance description please refer to [95].

For our benchmarks we select four problem instances: *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 the exact solvers and reported in aforementioned library. We present the optimal solutions in Table 5.1.

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

Table 5.1 TSP instances optimal tour length.

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

5.2 Environment Setup

We run our experiments with an enhanced by our approach BRISEv2 and deploy it in Docker containers on 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 engine version 1.13.1.

We deploy 6 homogeneous BRISEv2 workers with LLHs on the same host machine and run each experiment 9 times to gather the statistics. Each execution was performed with wall-clock-based termination criterion configured to shut down the optimization session after 15 minutes.

5.3 Meta-heuristics Tuning

As we conclude in Section 2.3.4, the goal of parameter control is at least to reach the parameter tuning approaches results quality. 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 use our concept implementation in the tuner mode. As mentioned in Section 3.5, to enable the parameter tuning mode we built a search space based on the singe LLH with its parameters and disable the solution transfer between configurations. In our particular case we define three search spaces for each underlying meta-heuristic. We run the tuning for 8 hours on 10 deployed worker nodes and give three minutes for each task (configuration) evaluation. The underlying prediction mechanism was configured to use TPE with 100% window size. We disable the repetition strategy and outliers detection leaving each configuration evaluated only once, since our preliminary experiments showed that the variance among evaluations is negligible.

5.3.2 Target Optimization Problem and Search Space of Parameters.

The role of optimization problem at hand is played by one of the selected TSP instances: *rat783*. We base the algorithms tuning on this instance, since it is a middle-size problem among the selected for evaluation instances.

jMetalPy evolution strategy. This meta-heuristic is implemented as a naive evolution strategy however, we found an important recombination mechanism missing, therefore, the heuristic is acting mostly by means of mutation operations. As a configuration, it requires several hyper-parameters of different types. Integer μ (*mu*), which denotes a number of parents in the population. Integer λ

(*lambda*) defines a number of offspring. We tune both parameters in range $[1..1000]$. Boolean *elitist* defines a selection strategy, where *true* value enables elitist selection $(\mu + \lambda)$, while *false* value disables it (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*. The probability of mutation is tuned in range $[0..1]$ respectively.

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.

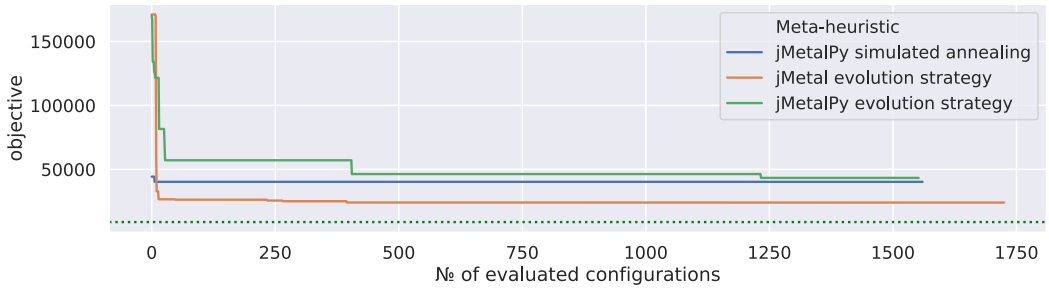


Figure 5.1 The low level heuristics parameter tuning process.

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.

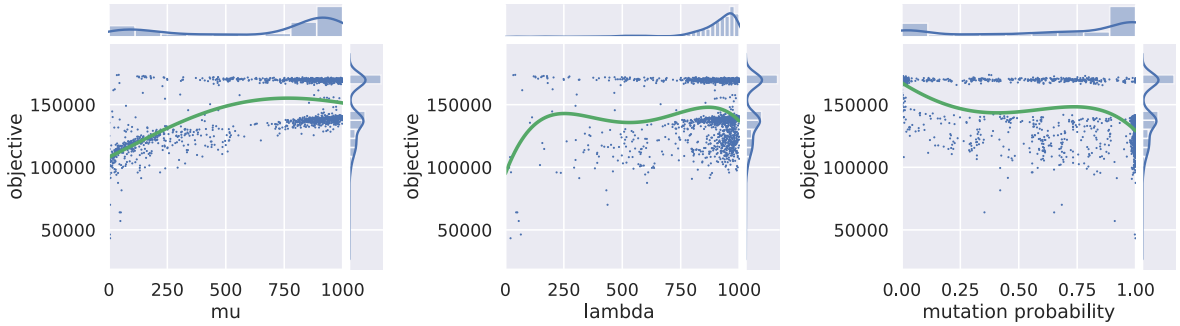


Figure 5.2 jMetalPy evolution strategy numeric parameters values.

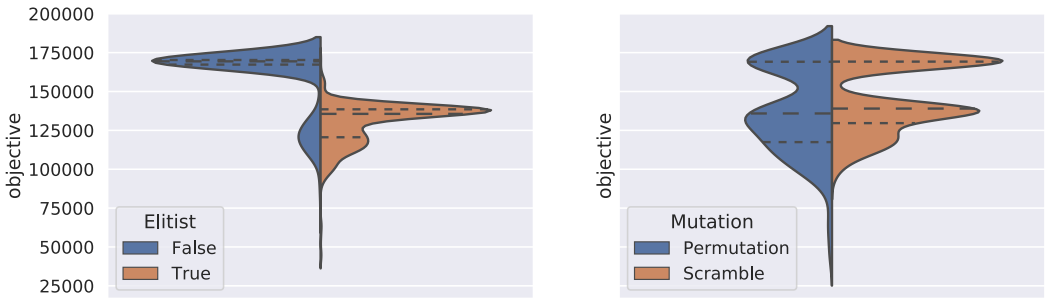
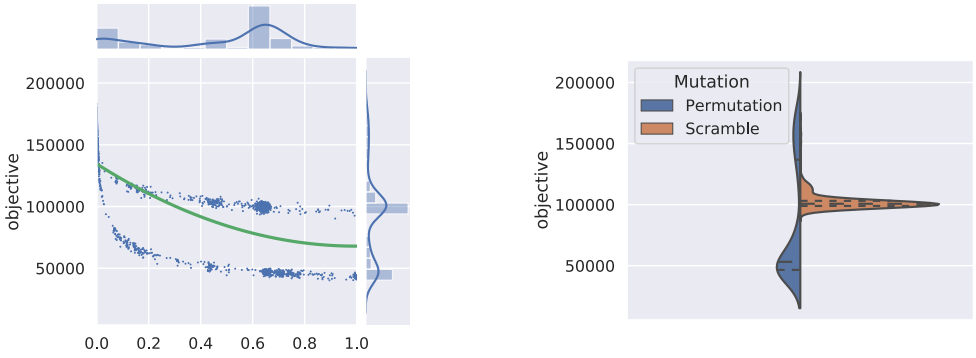


Figure 5.3 jMetalPy evolution strategy categorical parameters values.

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 a small amount of observations we may 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.

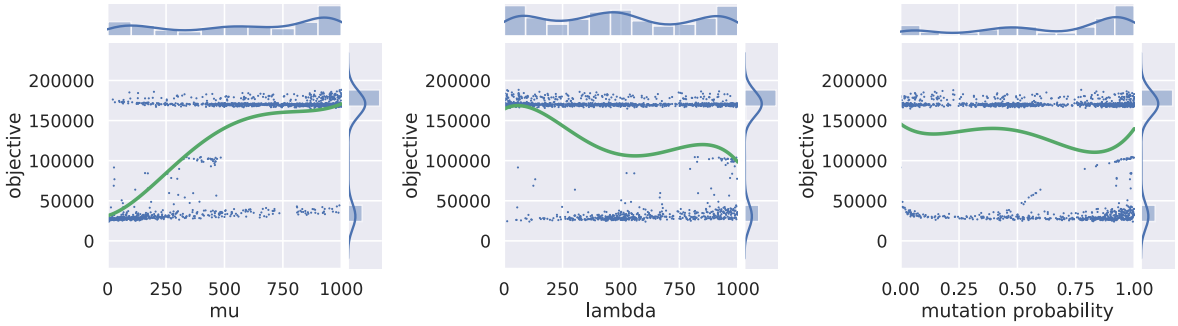


Figure 5.5 jMetal evolution strategy numeric parameters values.

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.

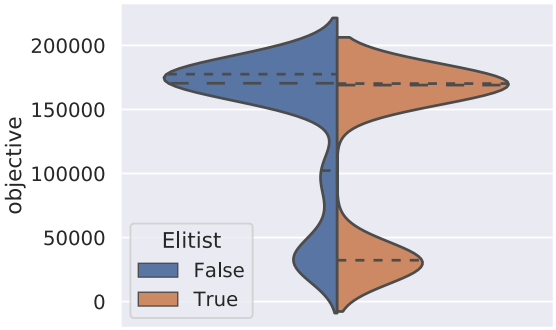


Figure 5.6 jMetal ES elitist parameter.

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 choose 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 default and tuned hyper-parameters: *4.1.1* and *4.1.2* for jMetalPy evolution strategy; *4.2.1* and *4.2.2*

[1] https://scikit-learn.org

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	

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 online 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 online 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. The required running time of this experiment set is approximately 9 days and 18 hours on a single machine.

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 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 doted line.

kroA100 and pr439 TSP instances. Both TSP for 100 and 439 cities are a relatively small problem instances. Therefore, all underlying MHs reach a local optimum after few first external iterations. The difference between the external and internal iteration is following: the first one happens, when the main node selects configuration and sends it to the worked node, which carries out the LLH (MH) execution (see detailed description in Chapter 4). On a contrary, the internal iteration occurs inside LLH itself, since it is an any-time algorithm. Thus, while the MHs reach a local optimum, there is no reason to spend much time for such cases review. We put a visual representation of benchmarks for kroA100 and pr439 into the thesis appendix (Appendix A.1.1).

The only worth to mention observation is a worse SA results with tuned parameters, in contrast to default values on kroA100 TSP instance. It is explained by the fact that for algorithm tuning we used a different problem instance (rat783). It only confirms a motivation of the parameter control approaches: tuning is not problem-instance-universal technique.

rat783 TSP instance. This is an average size problem among reviewed in the thesis. In contrast to previous instances, a behavior of solvers in this case changes slightly. For instance, ES from jMetal framework (j.ES) with default parameters is performing extremely slowly on a problem, however, while using an optimized hyper-parameters it quickly reaches a local optima (after 50 iterations) with the best produced results among other heuristics (Figure 5.7). Analyzing performance evidences of Python-based solvers, we may conclude that they almost reached a local optima in a given 15 minutes, therefore, their final results are slightly worse than the produced by j.ES (Section 5.4.2).

Please note, the perturbation of trends in the end of tuned py.ES run is caused by the difference in number of iterations among all runs (left figure in Figure 5.7, tuned parameters). The software [1], used in this thesis for results presentation estimates an average and the result deviation at each iteration. Thus, if one (or several) experiment execution(s) managed to perform more iterations than the majority of others, the average among and their deviation will be changed respectively. The number of such

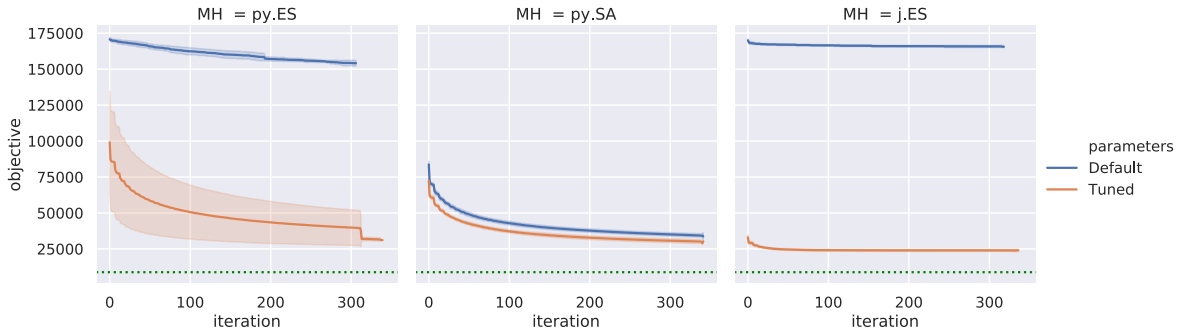


Figure 5.7 Intermediate results of meta-heuristics with static parameters on rat783.

[1] Python Seaborn data visualization framework web page: seaborn.pydata.org

external iterations varies, since as a termination criterion we used the wall-clock time. Unfortunately, this perturbation appear in most of the *progress charts*, therefore the comparison of final results is done by means of presented separately box-plots.

The bold lines in Figure 5.7 is a statistical mean of all 9 experiment runs with default parameter values (blue line) and tuned parameter values (orange line). A shadow around these lines is a confidence interval. One may observe how differently the parameter setting affects MHs: in evolution strategies the changes in performance is dramatic, while simulated annealing is almost not affected (Section 5.4.2). We also observe a decreased stability of tuned jMetalPy ES meta-heuristic (py.ES). It is reflected in a large confidence interval not only of the intermediate results, but also in a statistic of final solution quality.

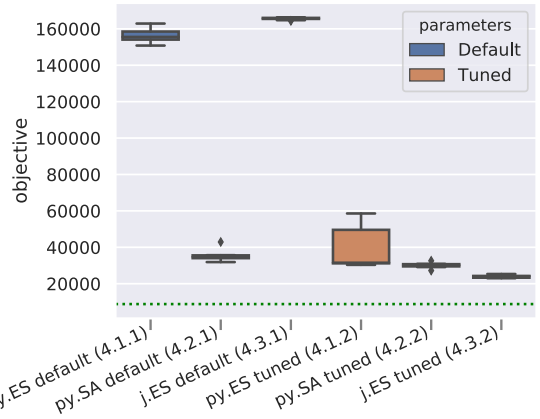


Figure 5.8 Final results of meta-heuristics with static parameters on rat783.

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

Python-based version of ES provide the worst results with both default and tuned parameters. Note the number of performed iterations by this MH with default parameters in a given 15 minutes is less 50. It is affected by a several reasons. Firstly, the amount of time required to perform an internal iteration increased dramatically. Thus, with specified 15 seconds for one task run, it actually takes much more time (up to 1 minute) to accomplish the task. We have a several guesses, what caused such a behavior. Firstly, it is a general Python performance issues, in most of the times caused by a global interpreter lock (GIL) usage. The explanation of GIL is out of this thesis scope, but roughly speaking, a multi-threading in Python causes struggle in comparison with single-threaded execution. Secondly, it may be caused by the algorithm code basis implementation, performed 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 finishing each iteration. Therefore, stopping criteria may be evaluated only after finishing this internal iteration, which in case of running py.ES with TSP instance for $7.4k$ cities, may take a while depending on the algorithm configuration. For instance, with parameters `{mu=5, lambda=10}` the

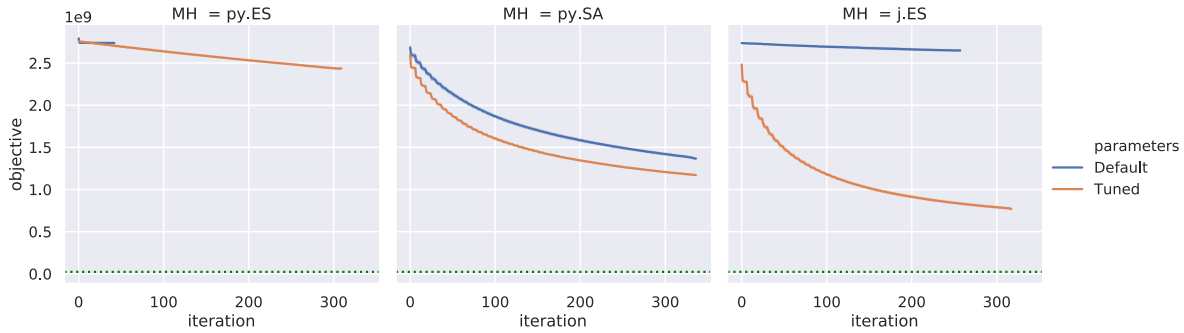


Figure 5.9 Intermediate results of meta-heuristics with static parameters on pla7397.

algorithm will terminate in time, while setting `{mu=500, lambda=500}` the algorithm will perform a higher number of computations on arrays of $7.4k$ integers long and as a result, may struggle. We observed the ES algorithm termination after a very first internal iteration. This also causes a poor solution quality improvements. Naturally, there is also an overhead in the results sending through the network, but as we observe on the Java-based ES performance with default parameter values, this overhead caused the decrease only in 20 external iterations. We also eliminate the possible issue in a required time for problem loading (building the TSP distance matrix), since with caching implemented in jMetalPy wrapper (see Section 4.4.2), 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 reporting delays and therefore, results in higher number of external iterations, but the solution improvements are still weak (see left picture on the Figure 5.9).

As in the previous case, jMetalPy simulated annealing produce good quality improvements at each external iteration, least depending on the hyper-parameter values. Even with a default configuration, py.SA outperforms the final results of py.ES after the first 50 external iterations. Setting the tuned parameter values, the performance of algorithm increases, but not dramatically. A resulting progress curve, presented in the middle of Figure 5.9 shows that py.SA requires more time to converge that was provided and is still far from its potential local optima.

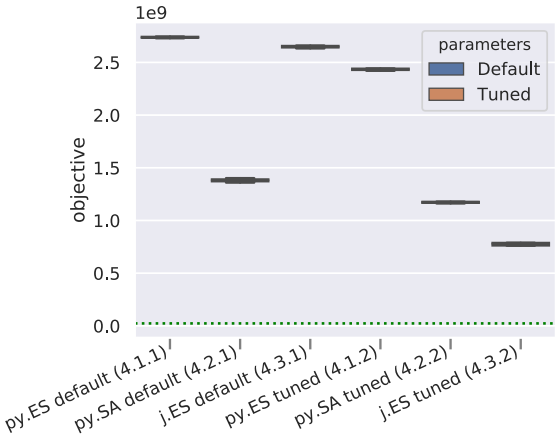


Figure 5.10 Final results of meta-heuristics with static parameters on pla7397.

jMetal evolution strategy is a perfect candidate to show, how important is a parameter setting. With default configuration j.ES is struggling in making improving steps and can not compete with other algorithms. Our guess here is the same as for the Python-based version: the number of internal iterations is extremely low for making a good search space traversal. However, a tuned version of j.ES outperforms all other solvers (see Figure 5.9 and Section 5.4.2 respectively).

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 the proposed in Chapter 3 generic parameter control approach to these algorithms is rather intriguing and may partially reveal the overall methodology benefits.

From the other side, we observe of 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 hyper-parameters will reveal the implemented approach applicability. In this case we expect to observe the results close to provided by pure SA. On a contrary, when we switch to tuned parameter usage, j.ES is preferred. We see it clearly when the MHs are applied to the biggest TSP instance with tuned hyper-parameters. Thus, we expect to observe such a behavior of selection hyper-heuristic.

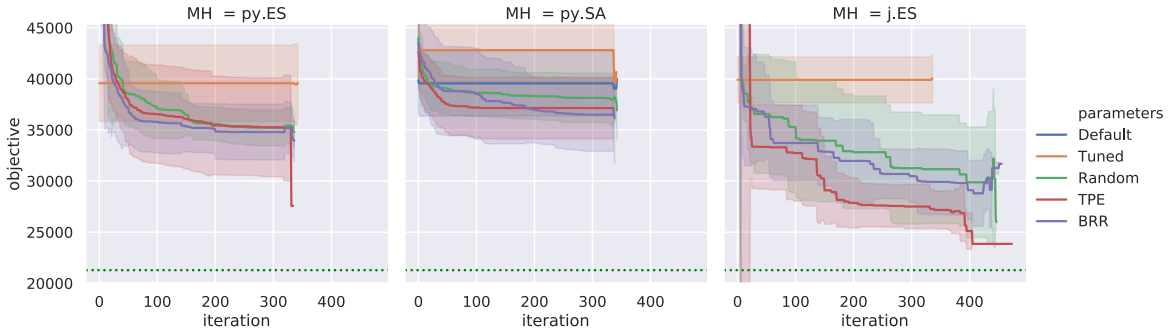


Figure 5.11 Intermediate results of meta-heuristics with parameter control on kroA100.

5.4.3 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 meta-heuristics in a runtime. Here is a brief reminder: at each RL step HLH is analyzing the past performance evidences of solver depending on its configuration 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, attaching the previously best obtained solutions. Please note, according to our setup, we have 6 simultaneously running and reporting workers.

In this part of evaluation we compare the performance of algorithms with statically defined default and tuned hyper-parameters to dynamically changing parameter values by means of RL control.

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 5.11). We may observe that even random changes of the heuristic parameters in a runtime results in finding a better solutions comparing with statically defined case (Figure 5.12). 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 assignment outperforms random allocation.

Note the amount of configurations (iterations) performed by jMetal ES, which could be seen in Figure 5.11. 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

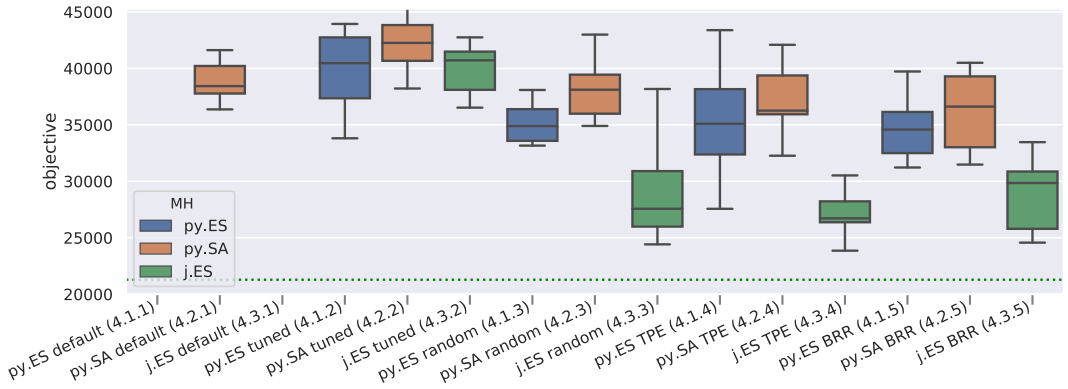


Figure 5.12 Final results of meta-heuristics with parameter control on kroA100.

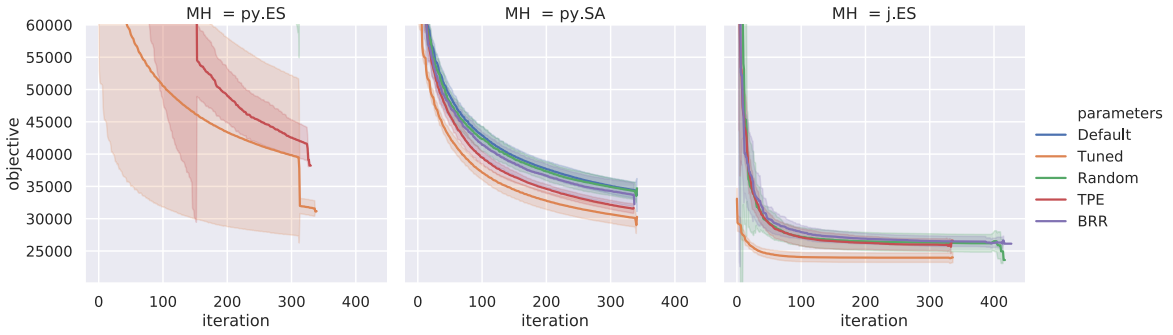


Figure 5.13 Intermediate results of meta-heuristics with parameter control on rat783.

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 flaw, which should be fixed in a future work.

pr439 and rat783 TSP instances. In this two cases, the behavior of solvers were similar, therefore, we decided to join their discussion and present only the plots for a larger instance (rat783). Still, the graphical representation of intermediate and final performance on pr439 TSP instance is presented in Appendix A.1.2.

When the parameter control is applied to a larger problem, it is starting to require more evidences for finding a good-performing settings of jMetalPy evolution strategy. Concretely, only the TPE-based parameter control was the closest in approaching the solution quality of tuned parameters. All techniques produced highly unstable intermediate and final results: please, draw your attention to the left side of Figure 5.13, and filled with blue boxes in Figure 5.14 respectively.

The results of parameter control application to less sensitive py.SA are following: randomized parameter sampling settled on the level of default parameters quality. BRR-based parameter control yielded a slightly better results, but not such stable, while TPE model approached the quality of tuned parameters (Figure 5.14).

On contrary, applying generic parameter control to j.ES MH leads to a comparable with the quality of

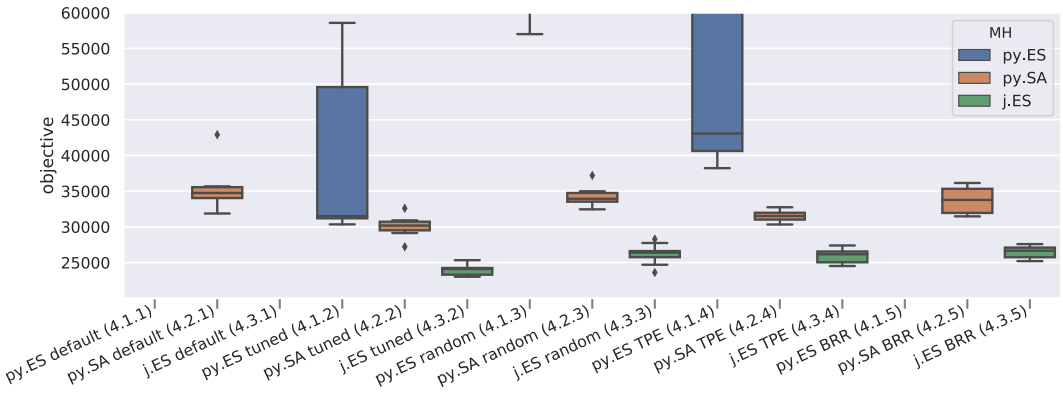


Figure 5.14 Final results of meta-heuristics with parameter control on rat783.

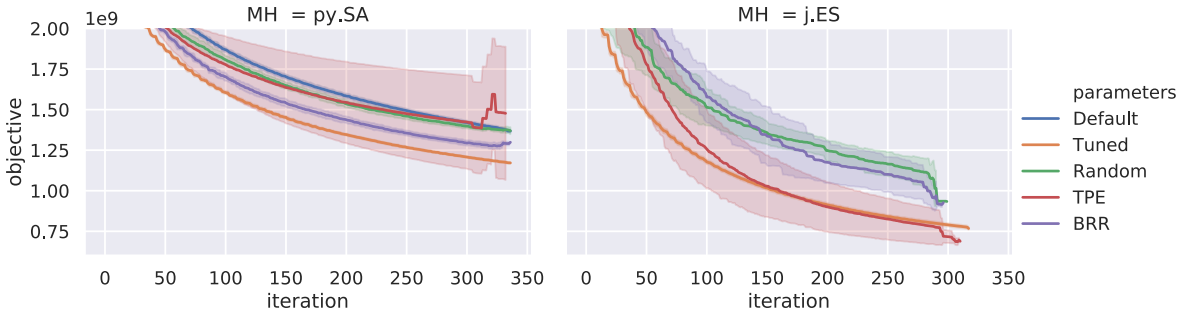


Figure 5.15 Intermediate results of meta-heuristics with parameter control on pla7397.

tuned parameters (Figure 5.14). Note, as for previous problem instance, even a random-based parameter sampling outperforms default parameters.

pla7397 TSP instance. For the final problem instance we omit the parameter control results of py.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 a prediction models. Thus, is case of py.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 in a given 15 minutes.

As in previous cases, the least parameter-settings-sensitive py.SA shows an ability to perform almost equally with any parameter settings (Figure 5.15). Neither among available control techniques was able to outperform tuned beforehand algorithm configuration in the final results quality (Figure 5.16). Among used models, only the BRR stably settled in terms of quality between default and tuned parameters.

As for j.ES, model-based approaches are outperforming the randomized parameter values allocation, not even talking about default parameters. Moreover, TPE-based control outperformed even the results of tuned beforehand hyper-parameters.

Discussion. In general, the review of meta-heuristic performance on different problem instances showed that the proposed generic parameter control approach is applicable and able to yield not only the near-tuned parameters quality, but in sometimes even outperforming results: all MHs with kroA100 TSP instance jMetal evolutionary strategy with pr439 and pla7397.

Taking into account the results with random parameter allocation we are making two conclusions. Firstly, even a randomized parameters changes are able to improve a potentially bad static hyper-

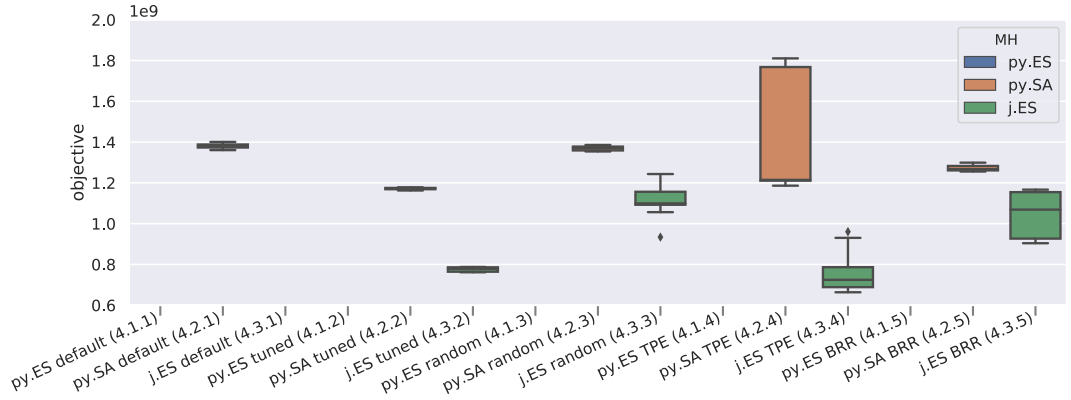


Figure 5.16 Final results of meta-heuristics with parameter control on pla7397.

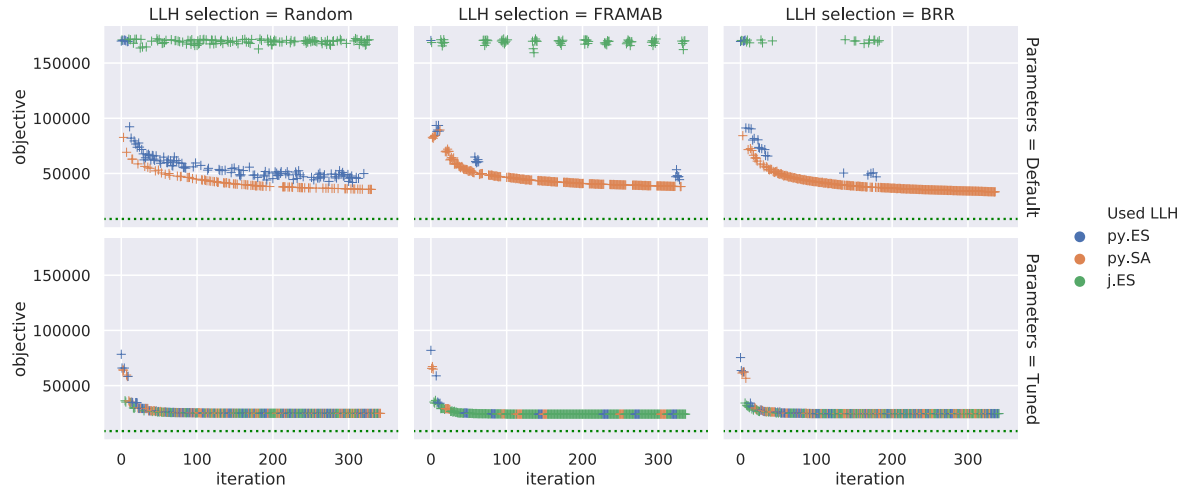


Figure 5.17 Intermediate performance of HH-SP on rat783 (single experiment).

parameter setting (j.ES case). Secondly, the learning mechanisms should and must be improved further by means of different surrogate models usage and proper technique for surrogates optimization to more resize parameter search. 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 online.

5.4.4 Selection Hyper-Heuristic with Static LLH Parameters

The second mode of developed approach and at the same time a main goal of this thesis is a process of dynamic heuristic selection. It was implemented in form of RL-based online selection hyper-heuristic, described in Section 3.5. Here we use three available LLHs (mentioned above py.ES, py.SA and j.ES) with static parameter (default and tuned) into selection hyper-heuristic (HH-SP). Three approaches to select the LLH were investigated: randomized, FRAMAB- and BRR-based HLH.

We present the problem solving process in two forms. Firstly, we distinguish the selected at each iteration LLH and the results, which it gave. For doing so, we took only the first repetition (out of 9 available). Secondly, we present the final results of all runs in form of box-plots, comparing them to the underlying LLHs performance. The left group of box-plots presents the final solution quality, obtained with the default parameter values, while on the right site the results of tuned beforehand LLHs are outlined.

kroA100, pr439 and rat783 TSP instances. Once again, we are grouping relatively small problem instances, on which the implemented HH-SP performs similarly. For the analysis we selected the largest instance among them: rat783. Nevertheless, the figures depicting kroA100 and pr439 TSP instances may be found in Appendix A.1.3.

Firstly, we would like to draw the reader's attention to HH-SP cases, in which the LLHs are used with the default parameter values (upper row in Figure 5.17). We do so, while according to the presented earlier baseline evaluation, there is only one algorithm with a strong performance dominance: py.SA. Thus, in Figure 5.17 we may observe a high frequency of py.SA sampling by both learning-based selection strategies. One may distinguish a repetitive pattern in LLH allocation with FRAMAB (middle column). It is caused by a deterministic essence of the algorithm. When it reaches the critical point of changing the favor of utilizing py.SA and exploiting other heuristic, the 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, which turns to be the second best performing LLH: j.ES. Therefore, in such a setup we conclude FRAMAB behaves slightly inertly. One may argue this will cause a performance struggling, which is rather a logical conclusion, but it requires a further investigation, which we are forced to postpone for the future work due to a lack of time. In case of BRR usage (right column in Figure 5.17), the bias is strongly shifted towards j.ES, which in some cases may cause performance issues due to lack of exploration. According to presented in Figure 5.18 final results statistics, given at least one dominating LLH (default parameters values case), even a random LLH selection could utilize enough times to obtain a good solution quality, however, it may require more time to converge. The model-based LLH selectors produce a better results quality.

The next setup is a set of LLHs with tuned parameters (lower row in Figure 5.17). According to the baseline evaluation, all among available LLHs are able to tackle the problem and produce a similar solution quality, however, the light difference in a dominance present and is as following (descending): j.ES, py.SA, py.ES. As a consequence, FRAMAB HLH frequently utilizes the best j.ES (see middle of lower row in Figure 5.17). On contrary, BRR HLH, similarly to random-based, samples all LLH types almost evenly. We may conclude that BRR is not as sensitive to performance evidences and was 'confused' since the process quickly converged into a local optima. The quality of final result presented in Figure 5.18 of all HP-SP versions with tuned LLHs are at least as good, as the solution quality provided by the best underlying LLH due to its fast convergence (see Figure 5.7).

pla7397 TSP instance. Our observations of different MH heuristics on the largest tackled in this thesis problem are as following. During the baseline evaluation py.ES with default parameters had the worst performance and not able to accomplish external iteration in time, while the tuned algorithm version was able to outperform only default j.ES. On contrary, j.ES with tuned parameters produced the best results, outperforming both middle-quality py.SAs. The best performing meta-heuristic with default parameters was py.SA. According to this information, we observe an expected behavior of HH-SP with default LLHs (upper row in Figure 5.19): the most frequently sampled by learning-based HLH was py.SA. However, the number of py.ES usages is suspiciously high in BRR case. According to baseline evaluation, the optimization process was not able to advance in a search using the py.ES (which we observe in a random-based sampling case, upper left cell in Figure 5.19). Till the present time we do not have a comprehensive explanation of this behavior. Referring to the final results, presented in Figure 5.20 we observe a high diverse in quality when py.ES is allocated frequently (codes 1.1. and 3.1.), which is an expected behavior, taking into account the performance of py.ES with default parameters.

Talking about the tuned LLHs case, we observe almost equal performance of all LLH sampling

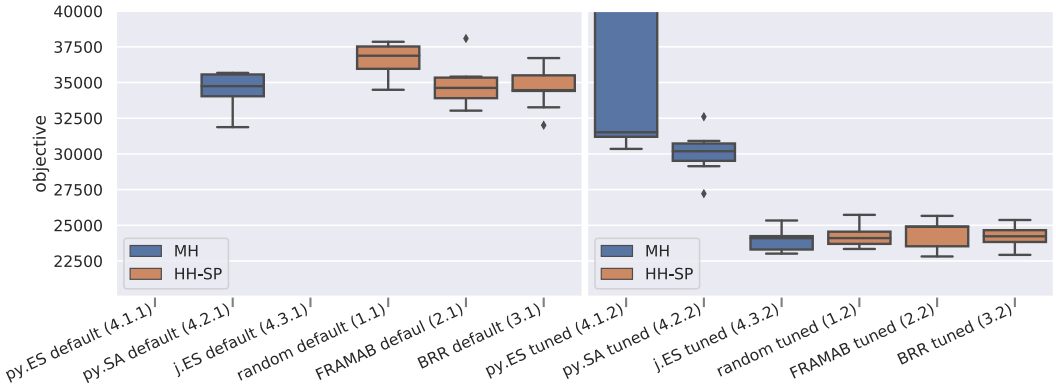


Figure 5.18 Final results of HH-SP on rat783 (statistic of 9 runs).

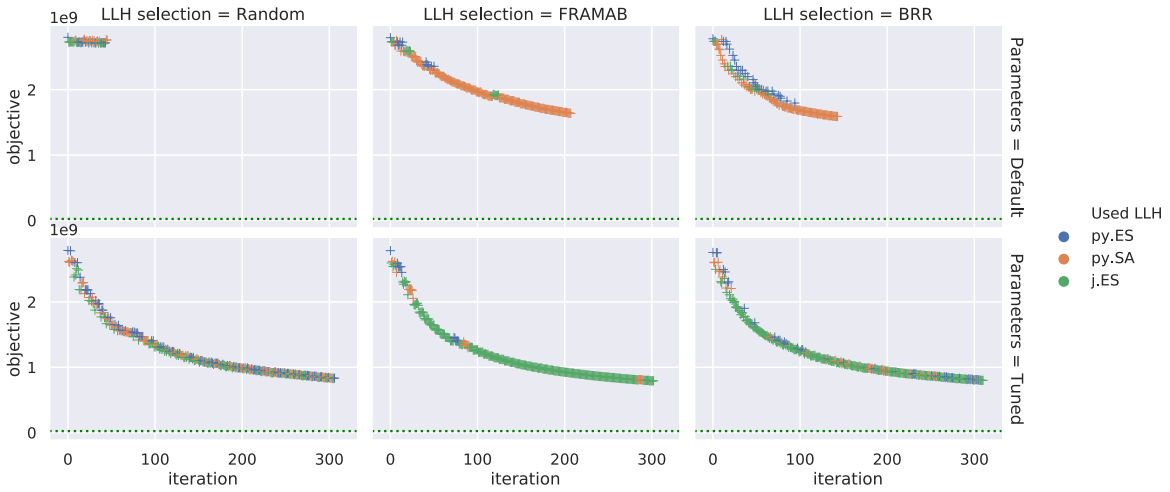


Figure 5.19 Intermediate performance of HH-SP on pla7397 (single experiment).

approaches, comparable to the best available LLHs final results. The solution quality of random-based HLH is slightly worse, in comparison to the results of FRAMAB- and BRR-based HH-SP due to their frequent usage of j.ES (see right chart in Figure 5.20).

Discussion. According to our observations of the developed selection hyper-heuristic performance we conclude that the proposed concept implementation operates as expected: HH-SP exposes similar performance to the best available underlying LLH. Two implemented selection HLH are performing slightly differently when reaching a local optimum. We claim the FRAMAB is a more perspective HLH, since it starts to balance between previously seeing good performing LLH exposing a good exploration abilities. On a contrary, BRR continues to utilize only the best performing heuristic. In case when the advantage of one LLH changes to another, BRR may need more time to learn this. Nevertheless, our guesses require more evaluation proves.

The observed issues call for not only a thorough investigation (pla7397 code 1.1, 3.1), but also a generic approach to handle a potential flaws in LLH implementation 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 for finding a good LLHs vs pure performance of these LLHs.

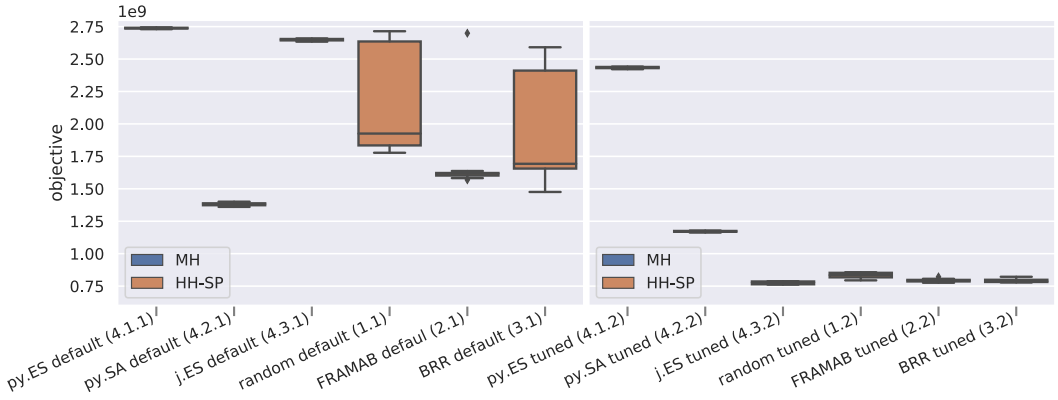


Figure 5.20 Final results of HH-SP on pla7397 (statistic of 9 runs).

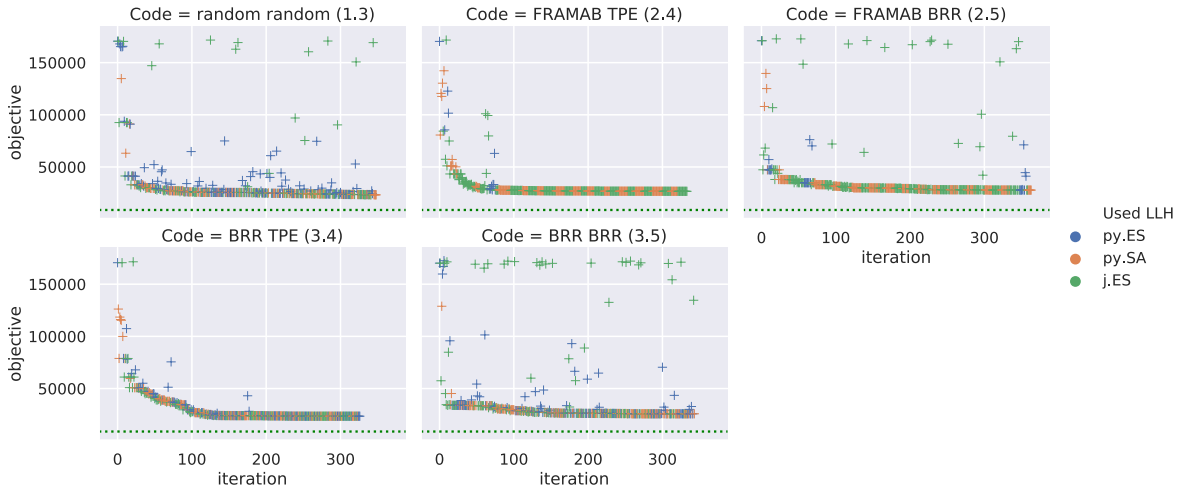


Figure 5.21 Intermediate performance of HH-PC on rat783 (single experiment).

5.4.5 Selection Hyper-Heuristic with Parameter Control

The final evaluation is dedicated to performance analysis of the suggested approach of merging the selection hyper-heuristic with the generic parameter control technique. A minimal goal is to obtain a performance of the best underlying LLH algorithm with tuned hyper-parameters. In this evaluation set we follow a similar to used for HH-SP method of intermediate results review distinguishing allocated LLH types at each iteration for one repetition and comparing the quality of final results over all repetitions with a baseline. 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 LLH parameter control we use the random, TPE and BRR sampling.

kroA100, pr439 and rat783 TSP instances. The decision to join all three TSP instances is motivated by a similar to mentioned in HH-SP discussion reasons: the system intermediate and final performance are rather similar among problem instances, therefore, here we review only a single (rat783) case. The results for all other problems may be found in Appendix A.1.4.

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 slightly differing, since the parameter control has started to search for good LLHs settings (Figure 5.21).

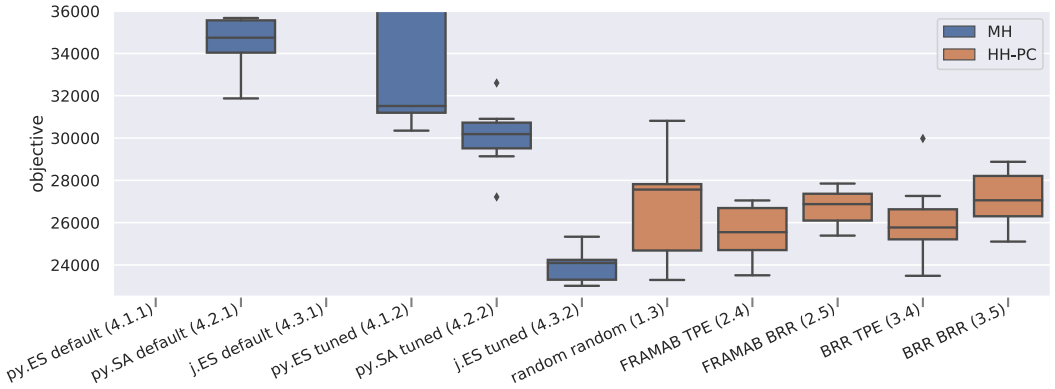


Figure 5.22 Final results of HH-PC compared with MH on rat783 (statistic of 9 runs).

Figure 5.23 Intermediate performance of HH-PC on pla7397 (single experiment).

Let us firstly draw the reader's attention to HH-PC with FRAMAB for LLH and TPE for parameter sampling (code 2.4 in Figure 5.21). At the beginning of solving process, j.ES was performing extremely well, for this reason FRAMAB was sampling it with a higher frequency. When a solving process reached its local optima (nearly 50^{th} iteration), FRAMAB started exploration of the other heuristics. The appeared 'noise' in the results of both j.ES and py.ES heuristics is caused by a boolean parameter *elitist*, which defines the selection strategy and may result in the solution quality degradation (a more detailed description of ES parameters could be found in Section 2.2.2). From an absence of noise in later stages of 2.4 and 3.4 its may be seeing that TPE has found *elitist=False* parameter value to be perspective in both ESs. On a contrary, BRR-based parameter controller did not find these parameters and glancing on the quality of final results (Figure 5.22) we conclude the BRR statistically finds worse performing parameters, comparing to TPE on rat783 problem instance. Also, we must not ignore the fact of early reaching a local optimum by the search process (see shapes of progress curves in Figure 5.21). In such case, the parameter search should be biased towards exploration. For that we need to introduce other progress metrics, such as stagnation detection (which is used in EA parameter control in [65]) and perform multi-objective RL optimization, maximizing improvement and minimizing stagnation. We postpone this enhancement for a future work due to a lack of time.

In Figure 5.22 we clearly see the dominating j.ES MH with tuned parameters (code 4.3.2). The quality of final solution, produced by the proposed concept implementation is statistically slightly lower the best performing tuned j.ES. This may be explained by a lack of time for parameter control to find a good performing setting, since the optimization reached its local optima too quickly to find good parameters for the underlying j.ES.

pla7397 TSP instance. As we observed in previous experiments, py.ES with non-tuned parameters causes struggling in hyper-heuristic performance (see Figure 5.9 and the discussion below Figure 5.15). That is why 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, 3.5 in Figure 5.23). While the case of fully randomized HH-PC (code 1.3) is clear, a BRR LLH selection did use this LLH frequently because it may have produced a good solutions quality as a result of parameter control behavior. On a contrary, FRAMAB LLH selection in combination with TPE parameter tuning (code 2.4) managed to find good parameters for py.SA, therefore, utilized it most often. When the FRAMAB's exploration component weight reached exploitation's, the other LLHs usage was triggered and as a result, HH-PC switched to j.ES usage. This

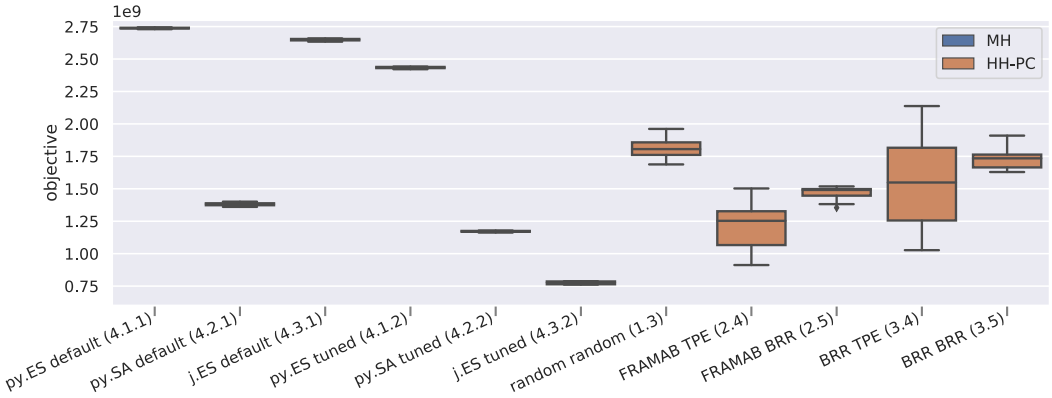


Figure 5.24 Final results of HH-PC compared with MH on pla7397 (statistic of 9 runs).

switch gave a dramatic result improvements (code 2.4 Figure 5.23). The FRAMAB LLH selection with BRR parameter control (code 2.5) at the beginning was using the mixture of mainly two j.ES and py.SA, but later switched to simulated annealing-only mode. As we may see, it gave a fast coarse-grained solution improvements in the beginning, and stable, but rather slow fine-grained improvements in a later stage.

On the final results quality charts (Figure 5.24) we observe a dominance of FRAMAB-based LLH selection (codes 2.4, 2.5) over BRR-based and TPE-based parameter control (codes 2.4 and 3.4) over BRR-based (codes 2.5, 3.5). On a contrary, the results obtained with BRR-based parameter control are more stable than TPE-based. The quality of final results did not reach the best performing tuned j.ES (code 4.3.2). However, since the optimization process did not settle in a local optima, we have a doubt that HH-PC will not outperform j.ES when given same number of external iterations.

For a better intuition, let us draw the reader's attention to Figure 5.25. Note how HH-PC (code 3.4) approaches j.ES progress curve. If it were not for the issue with a number of external iterations, the results would be better and, probably, outperforming tuned j.ES. Nevertheless, for current implementation and given 15 minutes for all solvers j.ES managed to make more iterations and moved further.

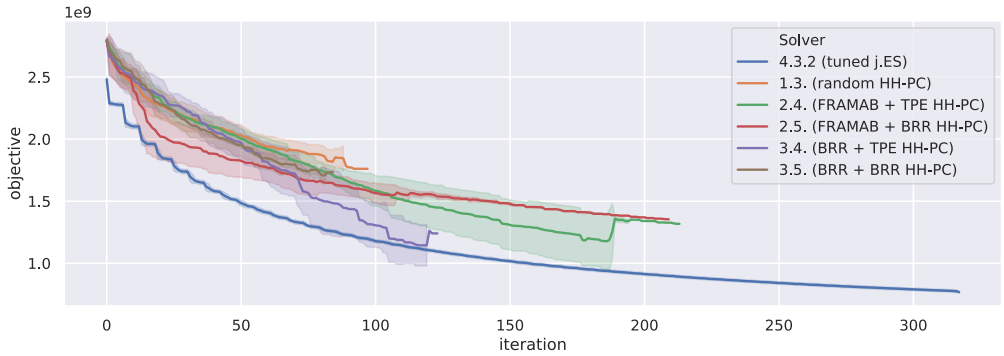


Figure 5.25 HH-PC and tuned jMetal ES solving process comparison on pla7397 (statistic of 9 runs).

Discussion. The observed results of solving the united ASP and PSP problems by HH-PC are encouraging. With small problem instances (kroA100, pr439, rat783) HH-PC managed to approach and in some cases even outperform the best underlying LLH with tuned hyper-parameters. When the problem size significantly grows (pla7397), the gap between HH-PC and the best performing LLH started to increase. An explanation for this behavior is simple: the system needs to build surrogate models not only for single, but for multiple LLHs and, therefore, requires more performance evidences in comparison to

MH-PC or HH-SP. On a contrary, the amount of information only decreases as a consequence of an issue with py.ES. Since the parameter values are not selected to properly reflect the LLHs performance, the algorithm selection process also straggles. To overcome the problem with lack of information we propose to execute an additional meta-learning step before the beginning of optimization session. Unfortunately, we are forced to postpone an investigation of this rather intriguing idea for the future work. Nevertheless, HH-PC significantly exceeds the results of MHs with static default parameters in all cases and the guided by models approach outperforms fully randomized HH-PC.

5.4.6 Concept Evaluation Numeric Results Discussion

We aggregate the results of each implemented mode in Tables 5.5 and 5.6. In each of them the baseline is defined as the best results obtained by underlying meta-heuristics used separately with static parameters. As an example, the baseline for kroA100 TSP instance is defined by jMetalPy.SA with default parameters, while tuned jMetal.ES produced the best solutions for pla7397. The results for each experiment separately are presented in Appendix A.2.

In Table 5.5 we compare the baseline with the *best* results in each mode. For instance, consider meta-heuristics with parameter control (MH-PC). Statistically, the best average result for kroA100 were produced by jMetal.ES with TPE control (code 4.3.4), which is equal to 27178, while the baseline is 39560. Therefore, the average result of the best MH-PC is equal to 0.687 of the baseline. We perform a similar aggregation for selection hyper-heuristic with static parameters (HH-SP), excluding the experiments, where LLHs were used with tuned parameters (therefore, selecting the best among 1.1, 2.1 and 3.1 codes). As for the selection hyper-heuristics with the parameter control (HH-PC), we pick the best averaged results in all available experiments, for instance, on kroA100 it was code 3.4 (HH-PC with FRMAB LLH selection and TPE parameter control) with averaged results in 30396.

Table 5.5 The best solution quality obtained by each mode compared with the best underlying meta-heuristic (baseline) on four TSP instances (lower is better).

TSP Instance	Baseline	MH-PC	HH-SP	HH-PC
kroA100	1	**0.687**	0.961	0.768
pr439	1	**0.973**	1.072	0.988
rat783	1	1.081	1.448	**1.064**
pla7397	1	**0.985**	2.228	1.546

Our conclusion on Table 5.5 is as follows: if one **knows** which among available meta-heuristics statistically produces better results (with properly selected parameters), the preference is to use the proposed generic parameter control (**MH-PC**) approach.

However, the situation changes dramatically if one **does not know**, which meta-heuristic is the best among available. In Table 5.6 we aggregate the average results over *all* experiments in each mode. It means that the results of all meta-heuristics with parameter control are taken into account to estimate MH-PC gain. However, we exclude several experiments to perform a fair comparison. For MH-PC we exclude the random-based parameter setting (4.1.3, 4.2.3, 4.3.3 codes). For both HH-SP and HH-PC modes we similarly exclude the random-based LLH and parameter selection (1.1 and 1.3 codes). Also, for HH-SP we ignore the experiments with tuned in offline meta-heuristics (1.2, 2.2, 3.2).

Therefore, our conclusion on cases, when one has several meta-heuristics, but **does not know**, which is the best choice, the complex approach of simultaneous online algorithm selection and parameter tuning (**HH-PC**) should be preferred due to its strong dominance over MH-PC and HH-SP modes.

Table 5.6 Average solution quality obtained by each mode compared with the best underlying meta-heuristic (baseline) on four TSP instances (lower is better).

TSP Instance	Baseline	MH-PC	HH-SP	HH-PC
kroA100	1	0.841	0.975	**0.773**
pr439	1	1.106	1.1	**1.017**
rat783	1	2.034	1.45	**1.1**
pla7397	1	2.139	2.36	**1.93**

5.5 Parameter Analysis of Hyper-Heuristic with Parameter Control

The second benchmark is dedicated to an evaluation of system settings influence on the solving process. Due to limited time we decided to perform the benchmarks only for the most complex system mode: online selection hyper-heuristic with parameter control in low-level heuristics (HH-PC).

5.5.1 Evaluation Plan

As with the concept evaluation, this set of benchmarks we also start from the planning. For comparison basis we selected one statistically better performing HH-PC setup, in which FRAMAB was used for the algorithm selection, and TPE for parameter control respectively. Their default configuration of which was as follows. Implemented FRAMAB algorithm exposes only one parameter: the balancing coefficient C. In Section 4.3.3 we discussed it values and also proposed an approach to replace this static value by one, derived from the deviation of results. In all previous tests we used exactly this approach. TPE implementation exposes *split size* parameter, which defines the percentage of available data, used to create a distribution of good-performing parameter values (see TPE description during HpBandSter framework review in Section 2.3.2). In all our experiments we used $1/3$ of available data to construct a 'good' distribution. The amount of information, which was used to construct the surrogate models was controlled by a shared parameter *window size*. In our evaluations we were using 80% of available information each time. As one may remember, since we did not implement a proper optimization algorithm over surrogates, but rather used a random sampling and afterwards selected the best parameter set by means of their results with surrogates, we used a predefined number of randomly sampled parameter values on each level (more details in Section 4.3.3). During our experiments we sampled 96 combinations of parameter values on each level (the default value in BRISEv2). Also, in our setup we used a predefined number of workers (6 ps) and static time for task execution (15 seconds). The implemented RL approach required continuing the solving process between iterations. For doing so, after accomplishing a task the workers were sending a complete bunch of obtained solutions to main node, which is then attached them to the new tasks. This difference is caused by the usage of single-solution simulated annealing and population-based evolution strategy meta-heuristics.

All aforementioned characteristics form a three groups of experiments:

1. **Learning granularity** group is designed to investigate the influence of performance evidences amount and quality, obtained between external iterations. It is formed by such parameters as *window size*, *task time* and *number of workers*.

2. **Learning models configuration** group of experiments are dedicated to investigate the influence of HLH parameters on the results quality and includes *FRAMAB C coefficient*, *TPE split size*, *random search size* over the surrogates.

3. **Generic low-level heuristic configuration** group is currently formed only from one characteristic for investigation: amount of solver warming-up information, which is defined by the number of solutions, reported in the end of LLH external iteration.

We aggregate the proposed parameters and define the values for evaluation in Table 5.7.

Table 5.7 Prediction techniques used for the concept evaluation.

Parameter	Investigated values	Default value
Learning granularity		
Window size	30%, 50%, 100%	80%
Task time	5, 10, 30 seconds	15 seconds
Number of workers	3, 9, 12	6
Learning models configuration		
FRAMAB C coefficient	Static 0.001, 0.01, 0.1	STD-based
TPE split size	10%, 50%, 70%	30%
Random search size	50, 200	96
Generic LLH configuration		
Warming-up solutions	one	all

We set all other parameters as they were configured during HH-PC evaluation, in particular the experiment running time is set to 15 minutes, number of experiment repetition is 9, the search space is unchanged (3 LLHs with the same parameter ranges and default values).

The idea of performing the full factorial design was quickly abandoned since it requires 46 days of non-stop experiment running. Therefore, we performed a one-exchange benchmark design, which resulted in 18 experiments, which needed 40,5 hours to perform 9 repetitions.

5.5.2 Learning Granularity

In this experiment we investigate the influence of RL routines configuration on learning process. The idea is as follows. Changing the *window size*, *task time* and *number of workers*, the underlying models may learn a different picture of dependencies. For instance, if we increase a task time, a sampled configuration will be measured more thoroughly, however, given a fixed experiment running time, the overall number of iteration will be decreased. On a contrary, by increasing a number of workers for the same fixed experiment time a size of investigated *parameter space* will be increased, therefore, the underlying learning models will construct a more precise surrogates.

Please note, the perturbations at the end of runs on progress charts are caused by different number of performed iterations. More detailed explanation could be found in Section 5.4.2.

Window size. According to our expectations in Section 3.3, this parameter is responsible for the forgetting mechanism, which is required for system adaptation to changes in optimization process and domination of one parameters over others (including LLH). In Figure 5.26 we presented the system running results with four different window sizes. We observe that the best results were obtained, when the learning models used all available information, in other words, with disabled forgetting mechanism. On a contrary, with the smallest window size the quality of final results are the worst. A trend could be observed, according to which with a larger window size system provides a better quality of results. During the benchmarks, dedicated to the concept evaluation we used 80% of available information, which corresponds to the middle-quality parameter value. Our conclusion is as following: with this problem instance and system setup, the changes in a learning process and parameter preference are

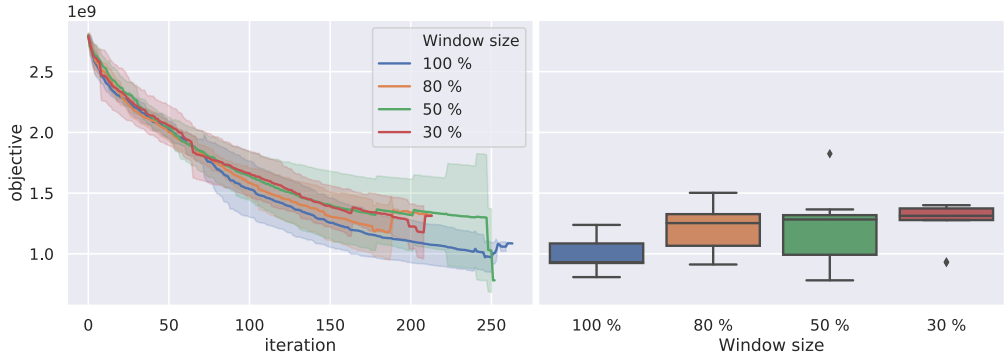


Figure 5.26 Influence of HH-PC (code 2.4) window size on pla7397 TSP instance solving process.

mostly negligible, therefore, the forgetting mechanism should be disabled. In a future, it would be rather intriguing to investigate the influence of this parameter with other, potentially dynamic problem and/or instance, where the parameter preference is changing.

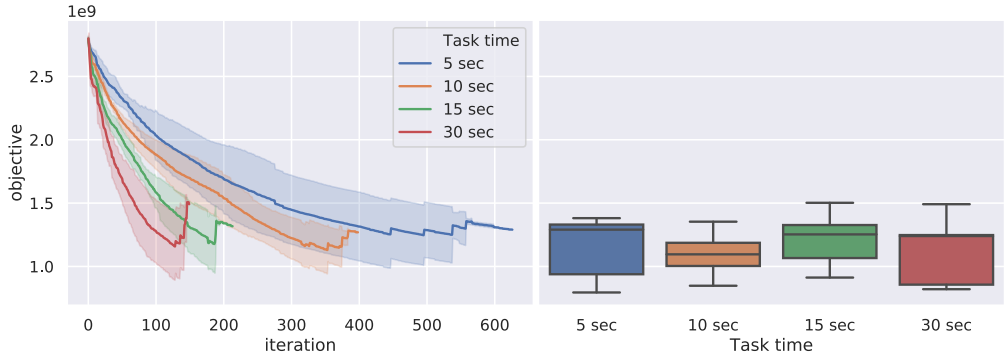


Figure 5.27 Influence of HH-PC (code 2.4) task time on pla7397 TSP instance solving process.

Task time. Varying the time for one task evaluation (external iteration), one will dramatically change the granularity of obtained results. For instance, in our experiments with the least amount of time (5 seconds) HH-PC managed to perform more than 600 external iterations, while with the largest budged (30 seconds) approximately 150 (Figure 5.27). However, as we mentioned previously, the more information does not necessary imply a more precise surrogate models. The used by us during the concept evaluation 15 seconds provided in the worst results. Boundary cases with 5 and 30 seconds task time gave rather unstable result (see box-plots in Figure 5.27). The former is full of too approximately evaluated parameters, while the later simply did not manage to perform enough iterations. Balancing between results stability and quality we conclude that for the current setup statistically better choice would be to set 10 seconds for a task running time.

Number of workers. The influence of workers number was rather surprise to us. According to our expectations, with a growing number of LLH runners, the amount of evaluated configurations increases. However, instead of getting surrogate models with a better quality and as a result, improving sampled parameters performance, we observe the opposite behavior. For increasing number of workers the results became worse and worse, see box-plots in Figure 5.28. Currently, we do not have a comprehensive explanation of the observed behavior, except of possible surrogate models over-fitting. It is broadly studied problem in ML, according to which the models create a too complex hypothesis, which can not generalize well to unforeseen data. Thus, in our case it is possible that being over-fit, models did not adequately predict a possible results for sampled parameters, therefore, guided prediction process in a

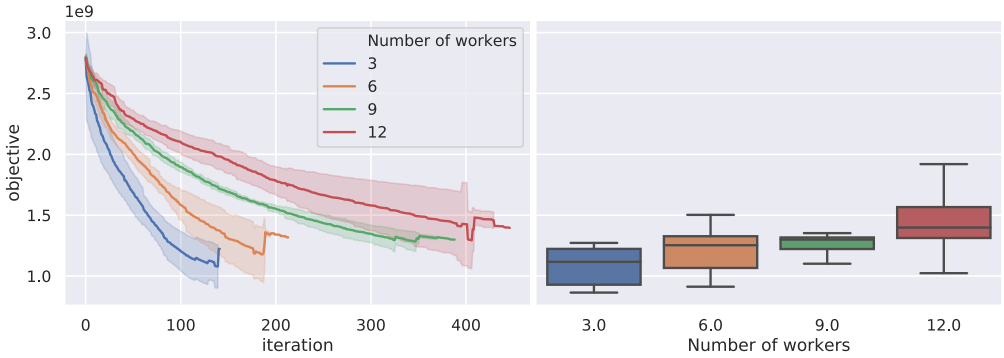
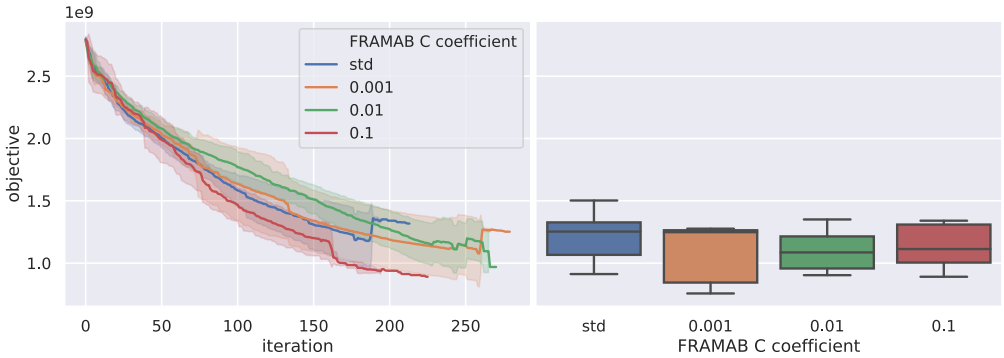


Figure 5.28 Influence of HH-PC (code 2.4) workers number on pla7397 TSP instance solving process.

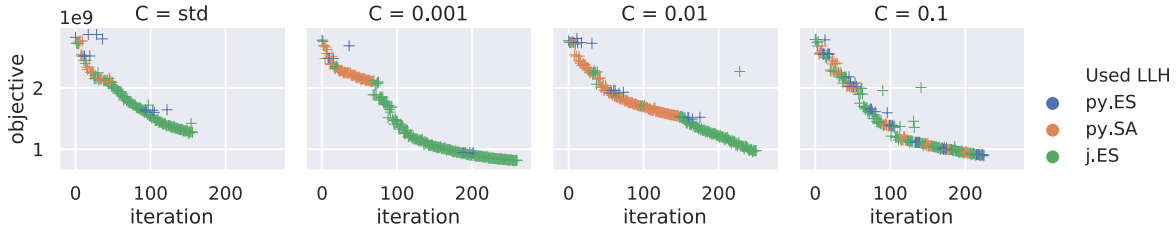
wrong direction. In any case, this behavior requires more comprehensive investigation.

5.5.3 Learning Models Configuration

We perform this set of experiments for getting an intuition about the influence of underlying high-level heuristic configuration on general performance. As mentioned above, in HH-PC code 2.4 we used following learning models: FRAMAB for LLH selection and TPE for parameter tuning.



a) Statistical results comparison.



b) One run with distinguished LLH allocation.

Figure 5.29 Influence of HH-PC (code 2.4) FRAMAB C coefficient on pla7397 TSP instance solving process.

FRAMAB C coefficient. As you remember from FRAMAB description, presented in Section 4.3.3, the role of coefficient C is to give the user possibility to control the EvE balance while selecting the LLH. In our implementation we proposed the usage of result improvement standard deviation, motivated by (1) the exploration encourage, while the uncertainty in results exist and (2) possibility of FRAMAB

parameter-less usage. Therefore, in first set of benchmarks we exclusively used this, STD-based FRAMAB regime.

The results of performed benchmarks with possibly different C values are presented in Figure 5.29. More concretely, in Figure 5.29a we compare statistic of all repetitions with different parameter value. Here we may conclude, the proposed parameter-less (STD-based) FRAMAB performs slightly worse in comparison to other algorithms. Glancing on the sequence of LLH allocation, presented in Figure 5.29b, we conclude that parameter reflects its intent, especially, comparing $C = 0.1$ and $C = 0.001$. When C is large, the exploration-related part of FRAMAB's UCB value is increased, therefore, more different algorithms are used for optimization. However, when the C value is decreased, only few switches may be observed. An intriguing idea arises to introduce the technique for FRAMAB parameter control, according to which, the entire LLH portfolio utilized at the beginning with high C value, while approaching the end C is increased, to more concentrate with the best-performing LLH.

TPE split size. The internals of Bayesian TPE approach for parameter sampling were described during BOHB algorithm discussion in Section 2.3.2. Evaluated in this experiment *split size* parameter defines the proportion of information, used to construct the probability distributions of good parameter values.

With small split size, only elite parameter values form distribution, therefore, an overall sampling process happens to be more greedy or, in other words, more exploitation-biased. According to our observations, presented in Figure 5.30, the least greedy parameter allocation produces statistically the worst results (70 %), while usage of 10 % split relieved the best potential. Used in previous experiments 30 % split produced slightly worse results than 50 %, which still could be caused by the randomized processes. Probably, this behavior should be investigated using more statistical data.

Random search size. This HLH parameter configures the random search process, performed over the created surrogate models. With given random search size equal N, for selecting the parameter values of LLH on each level N randomized samples are taken. Afterwards, these samples are compared with each other using surrogate models and one with the best results became level prediction. Therefore, according to our expectations, by increasing the random search size, the parameter values prediction process become more precise and as a consequence, performance increases.

After evaluating two additional to default sizes of 50 and 200 samples respectively, the obtained results happen to be not as we expected. In general, we observe a quality fluctuation, which is caused by randomized processes. When the sampling size was decreased, the quality statistics was slightly improvement, which is non-logical behavior (see intermediate and final results in Figure 5.31). It only motivates us to implement a proper optimization technique over surrogates for improving a robustness

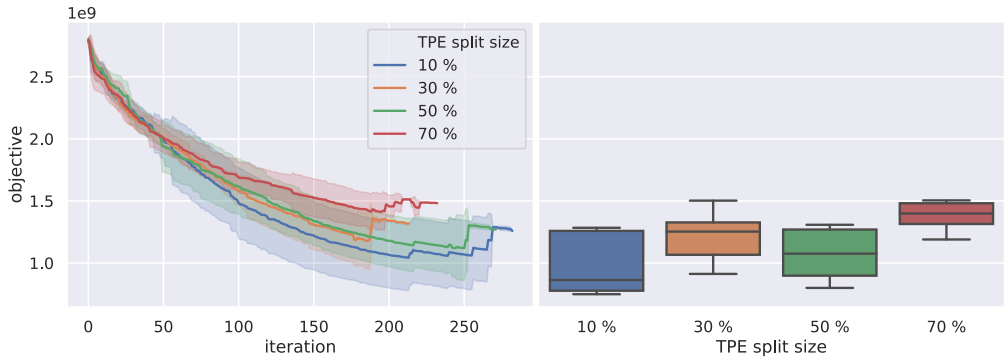


Figure 5.30 Influence of HH-PC (code 2.4) TPE split size on pla7397 TSP instance solving process.

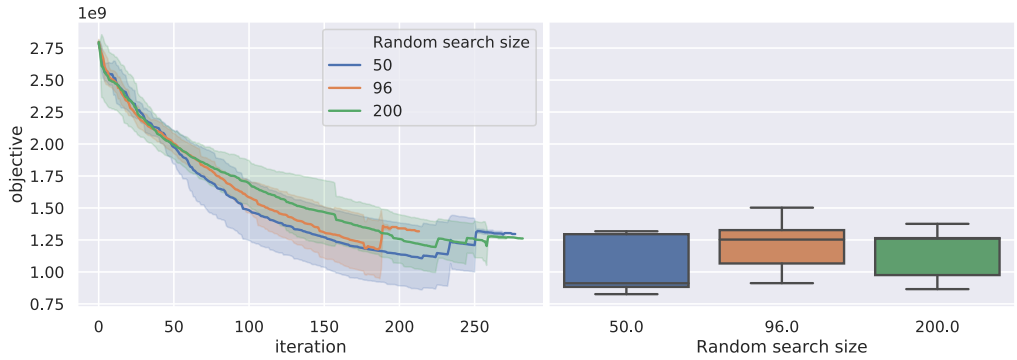


Figure 5.31 Influence of HH-PC (code 2.4) random search size for surrogate optimization on pla7397 TSP instance solving process.

of the prediction process.

5.5.4 Generic Low-Level Heuristics Configuration

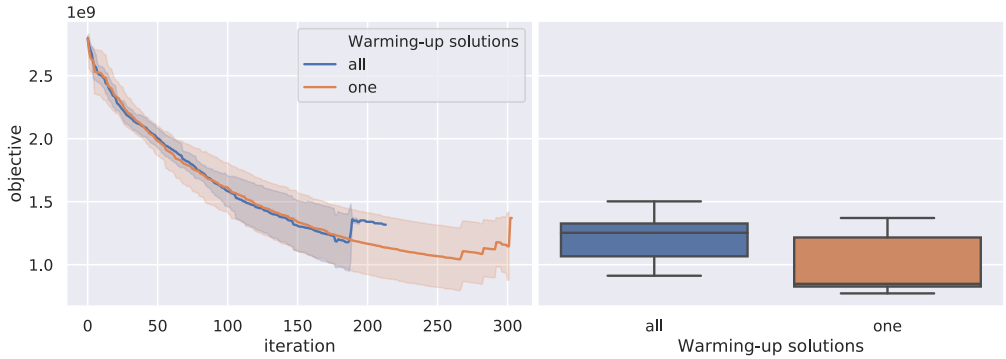


Figure 5.32 Influence of HH-PC (code 2.4) warming-up solution number on pla7397 TSP instance solving process.

The generic LLH configuration benchmark is represented by only one experiment, which we performed to evaluate how does the *number of warming-up solutions* affects the search. Our intuition during the implementation was following: we have to initialize the solver with all previously available solutions not to lose the obtained trajectory and traversal velocity. It is relevant only to the population-based algorithms, such as evolution strategy or potential genetic algorithm. The implemented py.SA simply used the best one among available population to start. When SA completes its run, it reports only a single obtained solution. If the following LLH is population-based ES, we do not copy SA-based solution, but rather allow ES to sample all the rest uniformly at random.

However, after changing this behavior to the only one warming-up solution usage, we surprisingly found out that the results quality was almost not affected: please, pay attention to overlapping progress curves in a left side of Figure 5.32. Moreover, passing only one solution between LLHs the overall number of external iterations were increased dramatically (from 200 to 300). It is caused by the reduced overhead for information processing and sending through network. We conclude that changing the behavior to only one warming-up solution usage provides a positive impact on the final results quality and do not affect the intermediate progress.

5.6 Conclusion

The evaluation of proposed concept was presented in this chapter and performed in two stages. At the first stage an analysis of the implemented concept was performed. We compared it with a baseline, defined by the executed in isolation underlying meta-heuristics with static hyper-parameters. Our conclusions on the concept applicability are following:

- Firstly, the proposed reinforcement learning-based generic parameter control approach (MH-PC) is able to significantly improve the performance of meta-heuristic's static hyper-parameters and in some cases even outperform the results of tuned beforehand parameters. From the implementation perspective, the system has lack of mechanism for bad-performing external iteration termination. On the generic level, it may be implemented in form of tasks termination mechanism, carried out by BRISEv2.
- Secondly, the developed heuristic selection technique (HH-SP) is able to reach a quality of the best performing underling low-level heuristic.
- Finally, the approach for simultaneous algorithm selection and parameter tuning (HH-PC) in runtime outperforms the best underlying algorithm with tuned parameters on rather small problem instances (kroA100, pr439). With growing complexity, the aforementioned issue of LLH struggling results in a gap between the desired performance of the best tuned algorithm and our approach. But, the guided by RL and surrogate models HH-PC considerably outperforms randomized selection of both LLH and parameter, which is the only approach available in runtime. A more confident conclusion for larger problem instances should be done after thorough investigation, where low-level heuristics are given more time to converge in local optimum and aforementioned issues are resolved.

In the second evaluation part we investigated an influence of the proposed united approach (HH-PC) configuration setting on its performance. The influence of some among evaluated parameters was not as we expected, which only motivates the importance of proper parameter values search and usage. More concretely, an insertion of relatively 'light' *forgetting mechanism* instead of improving an optimization process made the results statistically sightly worse. The increased number of workers only decreased the surrogate models accuracy. The decision of all available solutions usage for LLH initialization introduced a redundant overhead, but did not improve final results quality, therefore, it should be reconsidered. Nevertheless, a set of experiments from the second evaluation stage should also be performed over the other system operating modes (MH-PC and HH-SP) for making a confident conclusion. Up until now, it reviled the system adaptation ability with help of exposed parameters.

Yevhenii: add evaluation of 2.4 on a largest problem instance will all reviewed fixes?

6 Conclusion

Before making a final conclusion, let us briefly remind our objective. The task of this thesis was defined as follows: using an existing parameter tuning software propose a concept to (1) perform the parameter control in meta-heuristics on a generic level and (2) making both algorithm selection and parameter control solve the optimization problem at hand.

During our research the complex objective was split into several compound tasks, which were formulated in three research questions (Section 1.2), which we answer here explicitly.

RQ1 *Is it possible to perform the algorithm configuration at runtime on a generic level?*

During the review of dedicated to parameter setting problem studies, we found that most of the generic approaches are related to parameter tuning. In other words, the search of a proper configuration is made at design time. The idea of adapting algorithm parameters arose in field of evolutionary algorithms and spread into other meta-heuristics, however, in algorithm-dependent manner.

Yevhenii: is it needed?

The proposed in this thesis generic parameter control approach relies on two aspects. Firstly, it should be possible to evaluate the performance of system under control with specified configuration at any time. Ideally, is to limit the system run with specified beforehand budget (number of iterations, wall-clock time, etc). Secondly, it should be possible to change the target algorithm configuration and proceed with execution basing on previously obtained results. We use the reinforcement learning methodologies to traverse the parameter space evaluating the performance of unforeseen configurations iteratively, while solving the problem at hand. The proposed concept of generic parameter control was examined in Section 5.4.3 with three meta-heuristics: two Python-based algorithms, namely, simulated annealing and evolution strategy and one Java-based evolution strategy. The proposed approach revealed its applicability by reaching, and in some cases even outperforming the results of tuned in offline algorithm parameters. Please note, the use-cases of our concept are defined by the algorithms, which execution time is much larger than parameter control routines (see Section 3.1).

RQ2 *Is it possible to simultaneously perform algorithm selection and parameters adaptation while solving an optimization problem?*

Our idea of merging those two problems lays in treating the algorithm type as a regular categorical parameter in the search space. By utilizing the parent-child relationships we define dependent parameters in such search space and perform the selection by means of firstly sampling the independent parameters and hiding the children, and secondly fixing the selected for parents values and exposing the *activated* children parameters to proceed with prediction. The proposed step-wise process of configuration construction provides a possibility to utilize a wide range of surrogate models for learning the dependencies among parameter values on each level in isolation. The requirements and use-cases of the proposed approach remain the same as for the defined above generic parameter control technique.

RQ3 *What is the effect of selecting and adapting algorithm while solving an optimization problem?*

According to our expectations, the proposed approach should expose the performance comparable to the best among available for selection algorithms with tuned parameters. The performed in Section 5.4.5

evaluation and analysis of the simultaneous runtime algorithm selection and parameter control revealed its applicability. More concretely, given the same wall-clock time and environment setup, HH-PC was able to produce a better results than the best underlying tuned meta-heuristics for kroA100 TSP instance. For slightly larger pr439 example, HH-PC results were comparable with the tuned in offline meta-heuristics performance. With rat783 TSP instance HH-PC only in some cases reached the best available solver, used in isolation with tuned parameters. The experiments with pla7397 TSP instance shown an increasing gap between the HH-PC final performance and the best tuned meta-heuristic. However, analyzing the system and meta-heuristics behavior we found several issues not only in our implementation, but also in one of the underlying heuristics. Our evaluation of the developed HH-PC parameters influence in Section 5.5 showed that: (1) we did not use the best-performing settings for the used surrogate models; (2) RL routines configuration was not correct by means of LLH execution budget (task time), number of transferred solutions and forgetting mechanism. Therefore, our conclusion on the outperforming results of merging both problems for solving at runtime cannot be confidently assured yet. Nevertheless, the positive effect was definitely obtained even with the largest TSP instance: our approach is doing worse than the best tuned meta-heuristic, however, dramatically outperforms randomized LLH and its parameters allocation, which is the only available by this time online approach.

An explicit list of this thesis contributions is following:

1. The concept of reinforcement learning-based generic parameter control in meta-heuristics was proposed. Empirical evaluation on three meta-heuristic implementations proved its ability to reach the performance of tuned in offline parameters.

2. We proposed the concept of reinforcement learning-based online selection hyper-heursitic with parameter control in low-level meta-heuristics. The experiments demonstrated its applicability, however, the concept requires more thorough evaluation.

3. The usability of an existing SPL for parameter tuning BRISEv2 was extended with aforementioned cases without losing the flexibility of usage a wide range of learning models. Also, the concept of data preprocessing was encapsulated.

We consider the thesis task to be accomplished and the proposed reinforcement learning-based generic parameter control and algorithm selection approaches to be useful for solving the optimization problems with help of meta-heuristics.

7 Future work

In this chapter we discuss the postponed for future work investigations. We organized them into several groups and sort each of them in descendant by means of urgency and importance. Section 7.1 is dedicated to the prediction process and learning models used in HLH of developed approach. In Section 7.2 we discuss a set of enhancements for the search space that should be performed for better usability. Finally, in the section Section 7.3 we discuss a benchmark experiments, required to obtain a better evidences about the proposed approach applicability and HH-PC in particular.

7.1 Prediction Process

Surrogate optimization generalization. In Section 4.3 we discussed the process of parameter values prediction based on surrogate models. A classical approach of optimization with surrogates implies two steps. Firstly, the models should be constructed and evaluated by means of their accuracy. If the model is not accurate enough, it could result in a wrong prediction and as a consequence in wrong optimization guidance. After getting the proper model, the second step should be performed, namely, surrogate optimization. It is an actual process of prediction making, which bases on construction of configuration and using surrogate to estimate it's quality. In our work we used a random search surrogate optimization technique instead of implementing a more sophisticated algorithm due to the lack of time. However, the evaluation of random search intensity (Figure 5.31) revealed the urgency of this question, since the quality of random search results is not stable. Therefore, one of the first steps in future work should be the implementation of proper algorithm for the surrogate model optimization.

Process metrics for reinforcement learning. The proposed concept of reinforcement learning-guided parameter space traversal is based on the estimation of relative improvement, performed by the selected parameters (Section 3.1). However, the amount of obtained information may not be enough to truly estimate the configuration quality. We would like to emphasize that it is not a problem of the surrogates, but of the reinforcement learning. Thus, the possible improvement of results may be obtained from adding additional RL metrics. For instance, a RL-based parameter control for EAs [65] besides algorithm-dependent metrics, such as genotypic and phenotypic diversity, fitness standard deviation (in population) used also algorithm-independent metrics such as fitness improvement and stagnation counter. While improvement estimation is already used in our concept, the later, namely, a stagnation estimation is a possible candidate for additional learning metrics. Please note, using several such several metrics, the surrogate optimization process turns to be a multi-objective OP.

Meta-/offline learning phase. Relying fully on the online learning, in early stages (while surrogate models can not be constructed properly due to lack of information) our concept behaves as a random search. However, many studies, somehow related to proposed technique reported a significant performance boost in early stages, if the offline or in other words meta-learning was performed. More concretely, in [44] (also discussed in Section 2.4) the authors performed meta-learning to pre-train their surrogate models for guiding the search at the beginning. In [115] the authors developed selection hyper-heuristic with mixed learning type, therefore, before an actual run, the offline phase were executed

to guide the online selection at early stages. In our system we could use the results of previously executed experiments for similar scenario case to guide right from start not only the LLH selection, but also the parameter control.

Separately we would like to highlight a recent study [13], in which the authors also suggest to use reinforcement learning approach to solve a similar *dynamic algorithm configuration* problem. In their work the problem was enclosed in Markov decision processes which includes meta-learning step: learning across the problem instances.

7.2 Search Space

Composition on numerical parameters. In current implementation of the search space we highlighted that it is able to form the parent-child relationship only when parent is of the categorical type Section 4.2.2. However, it may happen, when the dependencies among parameters are based on their numeric values. As an example, imagine the algorithm in which for one parameter values range the first child type should be exposed, while for other range — another child. As a possible solution we propose utilizing a similar to used in categorical parameters approach, but instead of a single activation value, use ranges. Thus, during the prediction propagation step the parameter entity will check all ranges and expose 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. In this case along with activation values the notion of deactivation values may be introduced.

7.3 Evaluations and Benchmarks

The presented in Chapter 5 evaluation contains only the coarse-grained set of experiments, however, the proposed in this thesis concept of merging algorithm selection and parameter control comprises several building blocks, each of which should be thoroughly evaluated separately. Therefore, here we propose a set of fine-grained directions for future evaluation.

7.3.1 Use-Case Evaluation

Optimization problems. The advantage of hyper-heuristics lays in their ability to tackle a *family* or *class* of optimization problems, defiled by underlying low-level heuristics. Due to this flexibility of hyper-heuristics in general and the proposed concept in particular, by changing the domain-dependent components of low-level heuristics one will be able to tackle numbers of other optimization problems. Our evaluation in Chapter 5 is performed only on a traveling salesman problem. However, the combinatorial problems also include other types such as *flow-shop scheduling* [53], *nurse rostering* [26], *knapsack* [99], *n-queens* [97] and many other real-life and synthetic optimization problems. Used in our implementation jMetalPy framework [6] out of the box includes domain-dependent components for aforementioned knapsack problem, but thanks to its flexibility it is relatively easy to add other problem types.

Construction hyper-heuristics. The discussed in Section 2.2.5 idea of construction hyper-heuristics implies the algorithm creation from the building blocks. In our work we treated the *mutation, crossover, selection types* as the categorical parameters of underlying LLHs. However, in used jMetal and jMetalPy frameworks these parameters are implemented as separate operators that should be specified during

the algorithm instantiation. Therefore, the proposed concepts of search space and RL-based parameter assignment may be evaluated also for the construction hyper-heuristic cases.

Automatic machine learning. Making a step further from construction hyper-heuristic, an orthogonal research direction of an automatic machine learning field is exposed. The reviewed in Section 2.4 framework deal with the construction of machine learning pipelines that operate on datasets. In our approach, the proposed representation of the search space may be used to define the ML pipeline structure. For instance, the first several levels of our search space may encode the data preprocessing in ML pipeline. The successive level denotes the ML algorithm instances, while the final level is dedicated to validation technique. This system use-case should be evaluated against the already proposed solutions in automatic machine learning field, for instance, AutoSklearn [44], TPOT [88] and many others.

7.3.2 System Configuration Evaluation

The benchmark set, presented in Section 5.5 is dedicated to the implemented concept configuration evaluation. Due to the time constraints, we were able to probe only a few system modes and settings. Nevertheless, a vast bunch of experiments should be conducted urgently.

Modes of operation. In Section 5.5 only to HH-PC mode was benchmarked. But, it includes two other modes, namely MH-PC and HH-SP, which configuration should be evaluated separately by means of (1) underlying surrogate models settings, (2) adding more LLHs. While the influence of first direction was partially relieved in Section 5.5, the second direction, which is relevant only to HH-SP and HH-PC modes requires more clarification. By extending the search space with more LLHs, the RL will require more information (in terms of external iterations) for tuning available LLHs and differentiating among them by means of their performance. It is clear that with LLH number growth, the final performance gap between the best performing (tuned) LLH and HH-PS (HH-PC) will increase. The goal of this experiments will lay in a dependency estimation between search space complexity and the introducing RL-based search overhead.

Reinforcement learning configuration. Other worth-to-mention course of investigation is the influence of currently implemented RL-based optimization approach. The experiments with different TSP instances showed inefficiency of strictly defined time-based external iterations. For instance, with kroA100 TSP most of the used LLHs reached the local optimum after a couple of first external iterations and settled there till the end of optimization process. Thus, the *adaptive external iteration time* should be introduced, which analyzing the process stagnation will be able to terminate the optimization session earlier.

On a contrary, instead of time-based mechanism for external iteration termination, one may also limit the number of internal iterations performed by LLH. Using this mechanism the set of use-cases may be extended with the expensive for evaluation optimization problems.

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. URL: http://www.sciencedirect.com/science/article/pii/S0020025513001588.

[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. URL: http://arxiv.org/abs/1806.02326.

[25] Emilio Carrizosa, Belén Martín-Barragán, and Dolores Romero Morales. "A nested heuristic for parameter tuning in support vector machines". In: *Computers & operations research* 43 (2014), pp. 328–334.

[26] Brenda Cheang et al. "Nurse rostering problems—-a bibliographic survey". In: *European journal of operational research* 151.3 (2003), pp. 447–460.

[27] 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.

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

[29] Broderick Crawford et al. "Parameter tuning of a choice-function based hyperheuristic using particle swarm optimization". In: *Expert Systems with Applications* 40.5 (2013), pp. 1690–1695.

[30] Broderick Crawford et al. "Parameter tuning of metaheuristics using metaheuristics". In: *Advanced Science Letters* 19.12 (2013), pp. 3556–3559.

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

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

[33] 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.

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

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

[36] J Dreo. *Dreaming of Metaheuristics*. 2007. URL: http://nojhan.free.fr/metah/.

[37] 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.

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

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

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

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

[42] 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.

[43] 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.

[44] Matthias Feurer et al. “Efficient and Robust Automated Machine Learning”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2962–2970. URL: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf.

[45] Matthias Feurer et al. “OpenML-Python: an extensible Python API for OpenML”. In: *arXiv* 1911.02490 (). URL: https://arxiv.org/pdf/1911.02490.pdf.

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

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

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

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

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

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

[52] 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.

[53] Jatinder ND Gupta and Edward F Stafford Jr. “Flowshop scheduling research after five decades”. In: *European Journal of Operational Research* 169.3 (2006), pp. 699–711.

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

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

[56] 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.

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

[58] 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.

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

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

[61] 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.

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

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

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

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

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

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

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

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

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

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

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

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

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

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

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

[92] 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.

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

[94] 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. URL: https://doi.org/10.1145/3194078.3194082.

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

[96] 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.

[97] Igor Rivin, Ilan Vardi, and Paul Zimmermann. “The n-queens problem”. In: *The American Mathematical Monthly* 101.7 (1994), pp. 629–639.

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

[99] Keith W Ross and Danny HK Tsang. “The stochastic knapsack problem”. In: *IEEE Transactions on communications* 37.7 (1989), pp. 740–747.

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

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

[102] 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).

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

[104] 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.

[105] 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.

[106] Jim Smith. “Self adaptation in evolutionary algorithms”. PhD thesis. 2020.

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

[108] 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.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

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

[114] 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.

[115] 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.

[116] 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.

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

[118] 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.

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

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

[121] 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.

List of Figures

List of Tables

A Evaluation Results

A.1 Results in Figures

A.1.1 Baseline

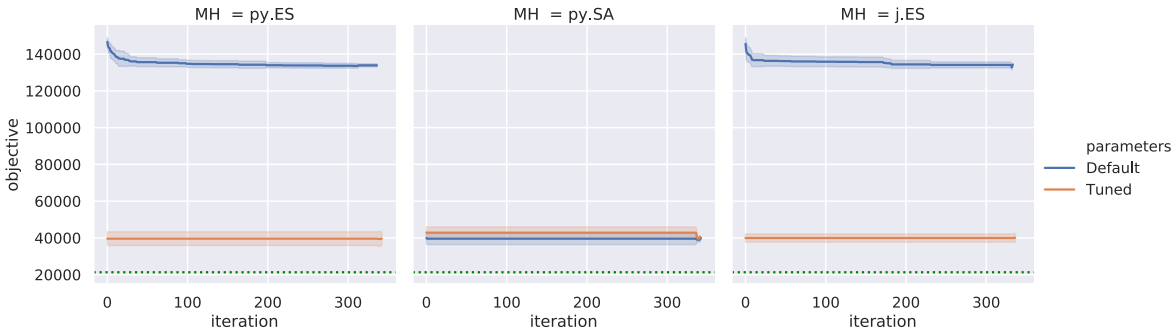


Figure A.1 Intermediate results of meta-heuristics with static parameters on kroA100.

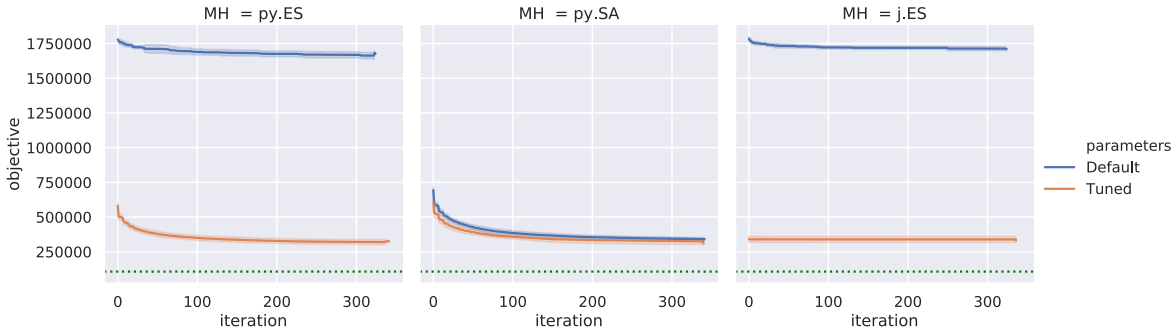
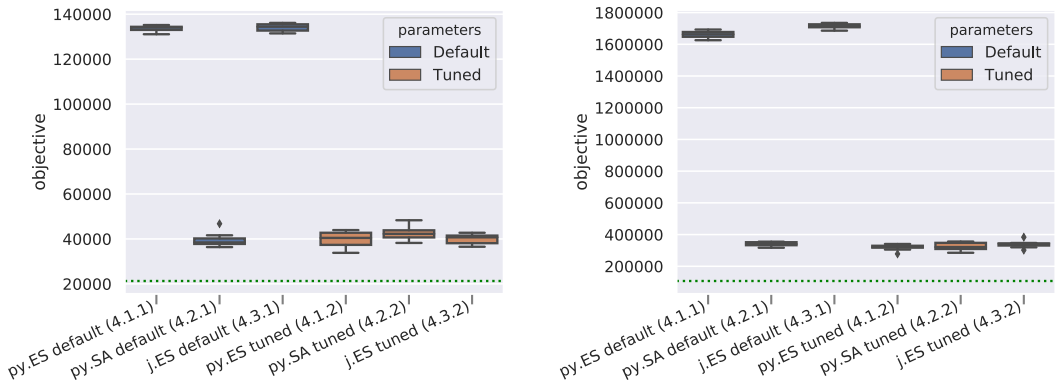


Figure A.2 Intermediate results of meta-heuristics with static parameters on pr439.



a) kroA100 TSP instance. **b)** pr439 TSP instance.

Figure A.3 Final results of meta-heuristics with static parameters.

A.1.2 Parameter Control

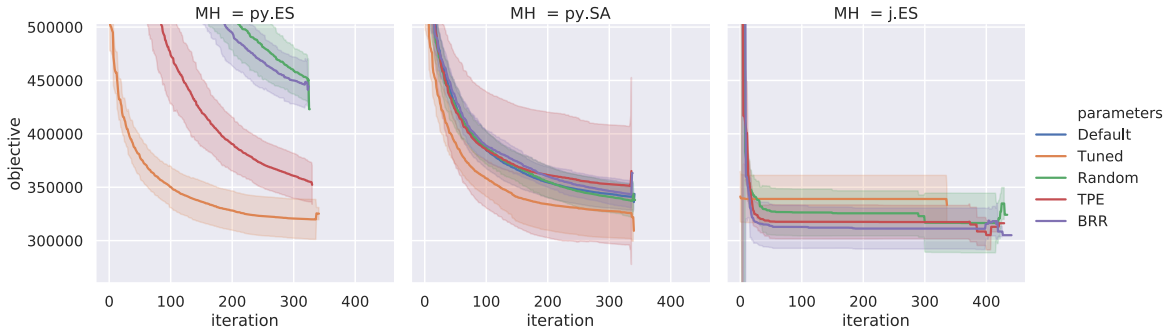


Figure A.4 Intermediate results of meta-heuristics with parameter control on pr439.

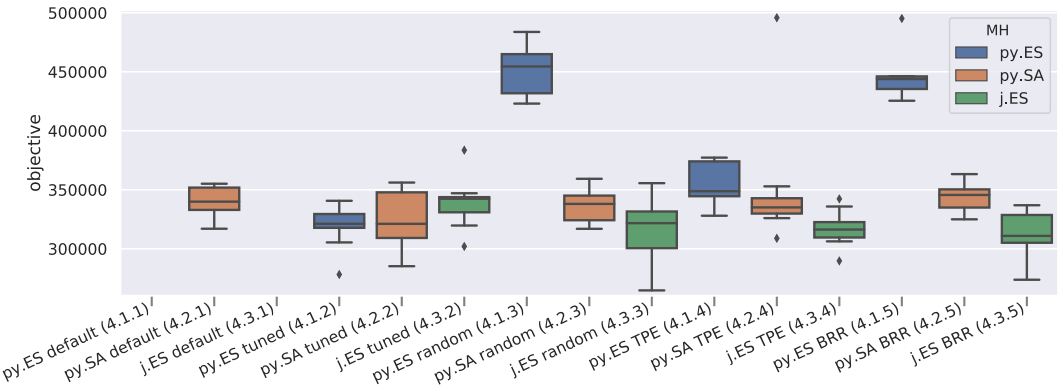


Figure A.5 Final results of meta-heuristics with parameter control on pr439.

A.1.3 Selection Hyper-Heuristic with Static LLH Parameters

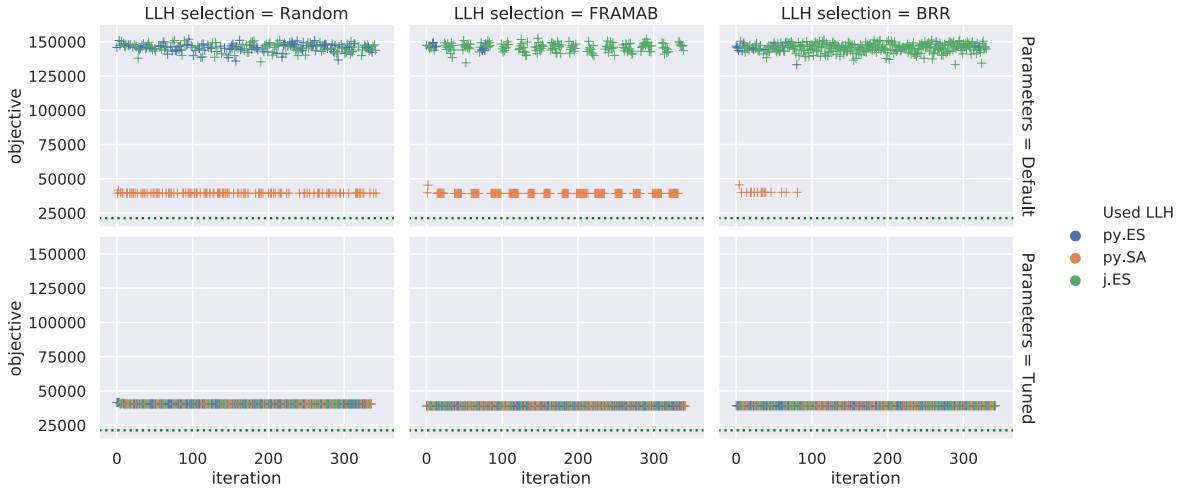


Figure A.6 Intermediate performance of on-line selection hyper-heuristic with static hyper-parameters on kroA100 (single experiment).

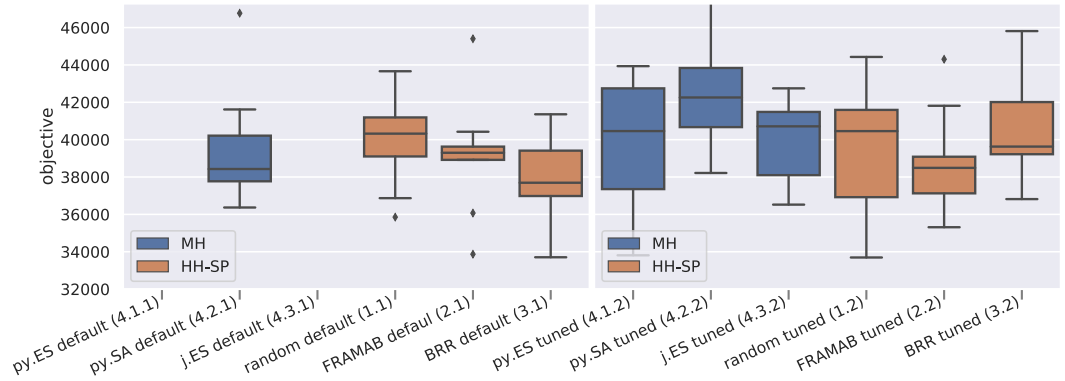


Figure A.7 Final results of on-line selection hyper-heuristic with static hyper-parameters on kroA100 (statistic of 9 runs).

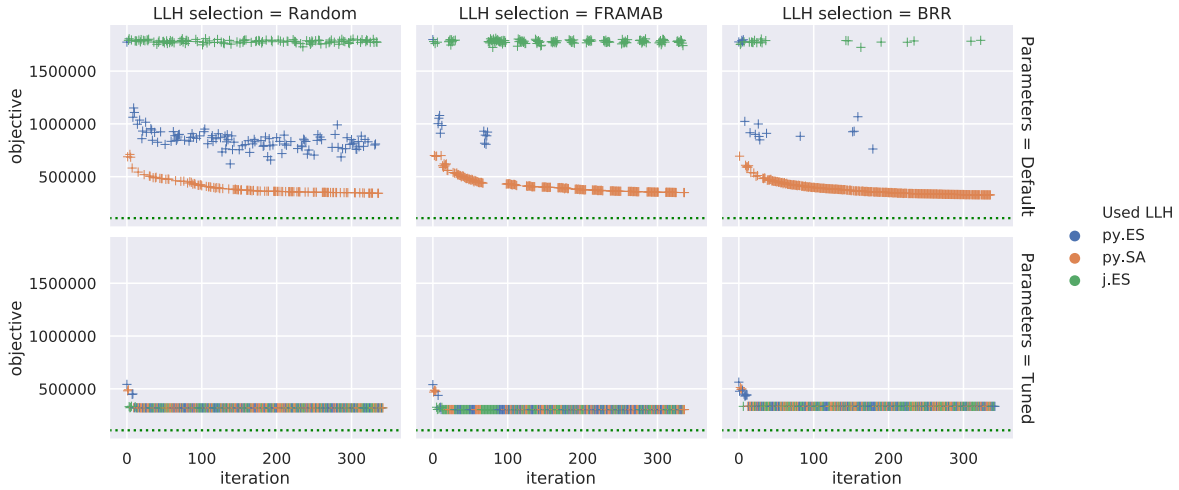


Figure A.8 Intermediate performance of on-line selection hyper-heuristic with static hyper-parameters on pr439 (single experiment).

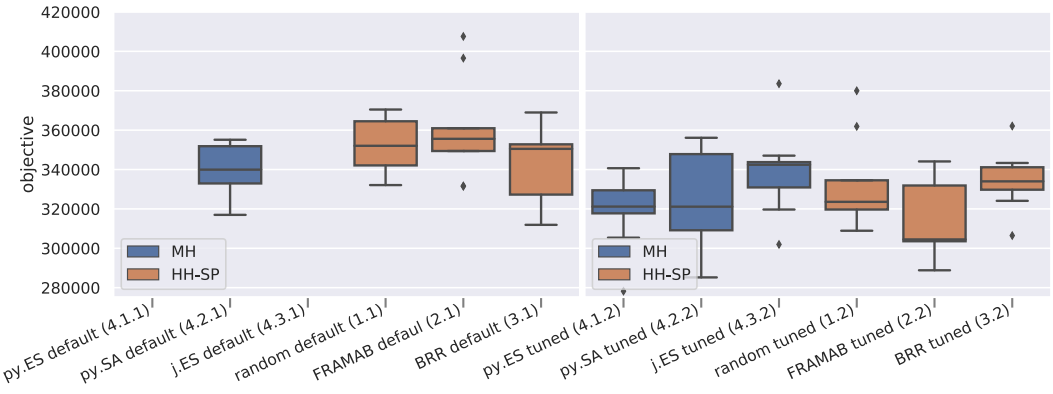


Figure A.9 Final results of on-line selection hyper-heuristic with static hyper-parameters on pr439 (statistic of 9 runs).

A.1.4 Selection Hyper-Heuristic with Parameter Control

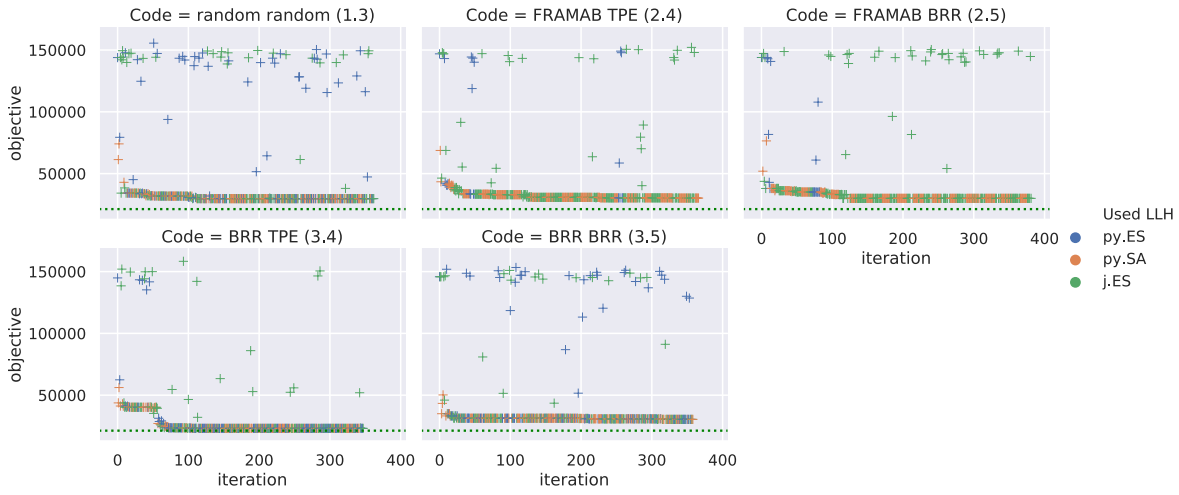


Figure A.10 Intermediate performance of HH-PC on kroA100 (single experiment).

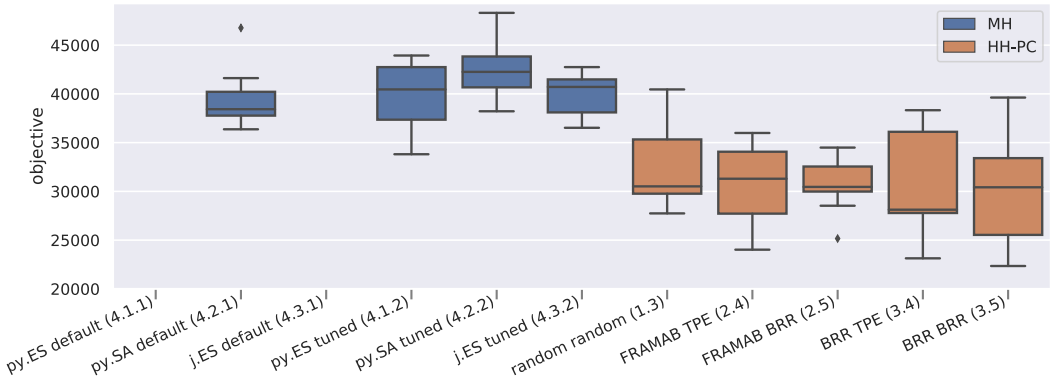


Figure A.11 Final results of HH-PC compared with MH on kroA100 (statistic of 9 runs).

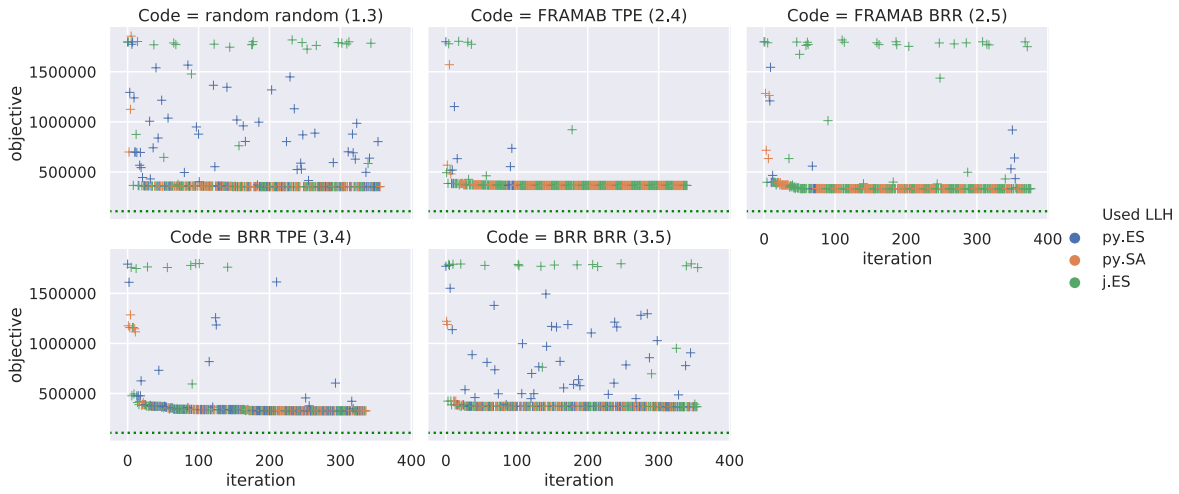


Figure A.12 Intermediate performance of HH-PC on pr439 (single experiment).

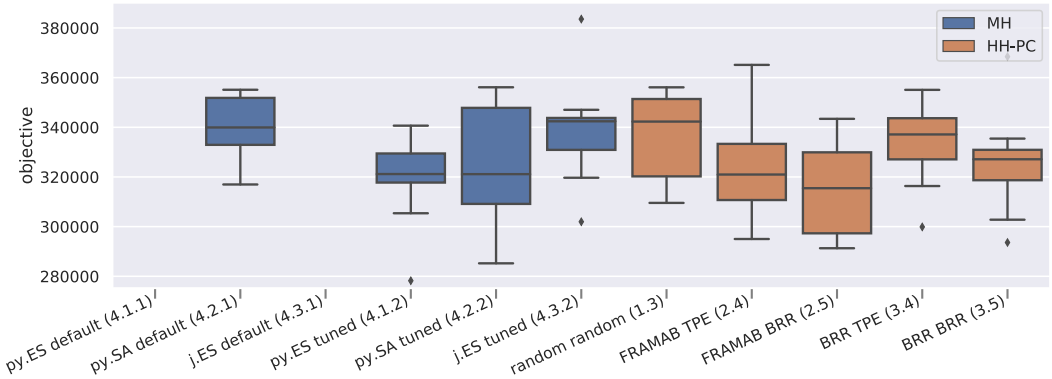


Figure A.13 Final results of HH-PC compared with MH on pr439 (statistic of 9 runs).

A.2 Results in numbers

Table A.1 Found paths distances for kroA100 TSP instance (optimal path length is 21282). The best result in experiment group is highlighted in bold, while the best result found for problem instance is also underscored.

Solver		Results		
Group	**Code**	**Best found**	**Average**	**STD**
Baseline	4.1.1	131075	133596	1249
	4.1.2	**33809**	39572	3738
	4.2.1	36368	**39560**	3191
	4.2.2	38216	42819	3149
	4.3.1	131453	134141	1634
	4.3.2	36522	39905	2242
MH-PC	4.1.3	33160	35219	1760
	4.1.4	27563	35265	4836
	4.1.5	31223	34790	2727
	4.2.3	34902	37986	2548
	4.2.4	32264	37145	3008
	4.2.5	31481	36497	3575
	4.3.3	24415	29768	5202
	4.3.4	**23855**	**27178**	1851
	4.3.5	24567	28793	3209
HH-SP	1.1	35853	40057	2585
	1.2	**33692**	39261	3431
	2.1	33868	39135	3132
	2.2	35312	38920	2711
	3.1	33704	**38019**	2237
	3.2	36818	40461	2565
HH-PC	1.3	27741	32404	4133
	2.4	24019	30652	3958
	2.5	25160	30764	2869
	3.4	23132	**30396**	5407
	3.5	**22345**	30564	5826

Table A.2 Found paths distances for pr439 TSP instance (optimal path length is 107217). The best result in experiment group is highlighted in bold, while the best result found for problem instance is also underscored.

Solver		Results		
Group	**Code**	**Best found**	**Average**	**STD**
Baseline	4.1.1	1625689	1663522	22427
	4.1.2	**278214**	**319803**	18705
	4.2.1	316973	340823	12546
	4.2.2	285220	325721	23970
	4.3.1	1685919	1713684	14766
	4.3.2	301964	338969	22113
MH-PC	4.1.3	423043	450570	20626
	4.1.4	327995	354661	18535
	4.1.5	425453	445088	20507
	4.2.3	316908	337100	15007
	4.2.4	308850	351326	55555
	4.2.5	324957	343358	12607
	4.3.3	**264713**	316496	27886
	4.3.4	289704	317360	15600
	4.3.5	273744	**311078**	19442
HH-SP	1.1	332095	351533	13784
	1.2	308913	332514	24001
	2.1	331487	360533	25971
	2.2	**288830**	**313859**	19735
	3.1	311906	342915	19211
	3.2	306461	334998	15175
HH-PC	1.3	309552	334987	18995
	2.4	295015	324616	21519
	2.5	**291299**	**315932**	18866
	3.4	299884	333992	17465
	3.5	293607	325887	21150

Table A.3 Found paths distances for rat783 TSP instance (optimal path length is 8806). The best result in experiment group is highlighted in bold, while the best result found for problem instance is also underscored.

Solver		Results		
Group	**Code**	**Best found**	**Average**	**STD**
Baseline	4.1.1	150799	156314	4157
	4.1.2	30350	39203	12357
	4.2.1	31873	35157	2825
	4.2.2	27215	30094	1393
	4.3.1	164349	165617	644
	4.3.2	**23015**	**<u>23964</u>**	724
MH-PC	4.1.3	57003	94429	49422
	4.1.4	38223	78434	59507
	4.1.5	62964	96369	48767
	4.2.3	32459	34229	1305
	4.2.4	30335	31545	749
	4.2.5	31475	33691	1784
	4.3.3	**23607**	26183	1353
	4.3.4	24517	**25916**	952
	4.3.5	25202	26450	796
HH-SP	1.1	34490	36637	1132
	1.2	23341	24233	741
	2.1	33029	34794	1444
	2.2	**<u>22816</u>**	24324	1007
	3.1	32006	34693	1507
	3.2	22927	**24149**	797
HH-PC	1.3	**23292**	26921	2430
	2.4	23512	**25500**	1334
	2.5	25388	26721	908
	3.4	23488	26129	1801
	3.5	25105	27114	1233

Table A.4 Found paths distances for pla7397 TSP instance (optimal path length is 23260728). The best result in experiment group is highlighted in bold, while the best result found for problem instance is also underscored.

Solver		Results		
Group	**Code**	**Best found**	**Average**	**STD**
Baseline	4.1.1	2730074917	2737831472	4225616
	4.1.2	2421486544	2432905011	7405688
	4.2.1	1360505168	1380256720	13657387
	4.2.2	1162583357	1171728906	5239671
	4.3.1	2633785765	2648540705	8181262
	4.3.2	**761509264**	**775957737**	10846169
MH-PC	4.1.3	2729415229	2738113236	7383459
	4.1.4	2712114799	2734298124	9381522
	4.1.5	2723433962	2732529772	5694368
	4.2.3	1354022277	1369131424	11698311
	4.2.4	1186230750	1407196180	292371422
	4.2.5	1255612025	1271166341	14802252
	4.3.3	933923318	1109347363	85835240
	4.3.4	**663489188**	**764140101**	108747262
	4.3.5	903993065	1048196489	109819327
HH-SP	1.1	1777658580	2203819260	437999367
	1.2	794461798	832754319	22848007
	2.1	1569150086	1728591281	364476875
	2.2	**775887027**	794478412	13501812
	3.1	1475527859	1934187052	449660041
	3.2	777931654	**791768317**	14578963
HH-PC	1.3	1687855495	1815721794	87307521
	2.4	**912264049**	**1199738016**	205533985
	2.5	1354196166	1460681037	57770764
	3.4	1026880309	1579805637	375451837
	3.5	1629147332	1736442893	89413787

Confirmation

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

Dresden, 10th May 2020