TECHNISCHE UNIVERSITÄT DRESDEN

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

From Parameter Tuning to Dynamic Heuristic Selection

Yevhenii Semendiak

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

Master Thesis

to achieve the academic degree

Master of Science (M.Sc.)

Supervisors
MSc. Dmytro Pukhkaiev
Dr. Sebastian Götz

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

Submitted on: 11th May 2020

Technische Universität Dresden/Fakultät Informatik

Aufgabenstellung für die Masterarbeit

Name, Vorname: Semendiak, Yevhenii

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

Thema:

From Parameter Tuning to Dynamic Heuristic Selection

Zielstellung :

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

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

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

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

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

Institut: Software- und Multimediatechnik

Beginn am : 01.10.2019

Einzureichen am : 09.03.2020

Unterschrift des verantwortlichen Hochschullehrers

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

Contents

List of Figures

List of Tables

0.1 abstract

Abstract will be available in final versions of thesis.

1 Introduction

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

1.1 Motivation

Structure:

- optimization problem(OP) → exact or approximate (+description to both) → motivation to use **approximate solvers** →
- impact of parameters, their tuning on solvers → motivation of **parameter control** (for on-line solver) →
- but what if we want to solve a class of problems (CoP) → algorithms performance is different →
- user could not determine it [60] → exploration-exploitation balance
- no-free-lunch (NFL) theorem [107] → motivation of the thesis

thesis motivation The most related research field is Hyper-heuristics optimizations [18], that are designed to intelligently choose the right low lewel heuristics (LLH) while solving the problem. But the weak side of hyper-heuristics is the luck of parameter tuning of those LLHs [links]. In the other hand, meta-heuristics often utilize parameter control approaches [links], but they do not select among underlying LLHs. The goal of this thesis is to get the best of both worlds - algorithm selection from the hyper-heuristics and parameter control from the meta-heuristics.

1.2 Research objective

Yevhenii: Rename: Problem definition?

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

Analysis of existing studies of algorithm selection. *(find a problem definition, maybe this will do [60])*

Analysis of existing studies in field of parameter control and algorithm configuration problems *(find a problem definition)* [66]

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

Evaluation of the developed approach with

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

.

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

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

1.3 Solution overview

Yevhenii: Rename: Problem solution?

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

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

2 Background and Related Work Analysis

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

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

2.1 Optimization Problems and their Solvers

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

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

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

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

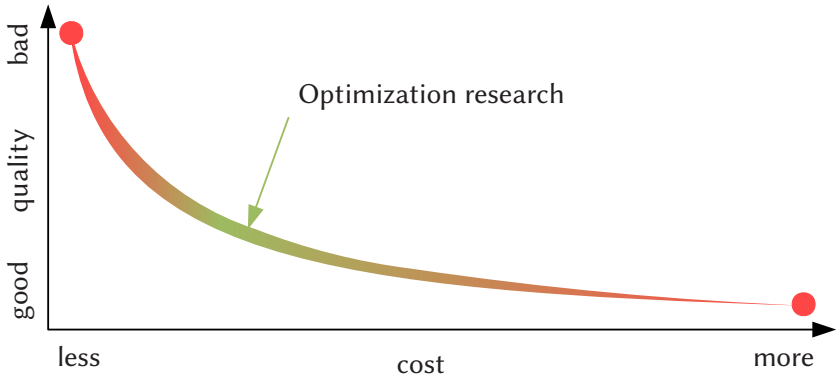


Figure 2.1 Optimization trade-off.

2.1.1 Optimization Problems

While the *search problem* (SP) defines the process of finding a possible solution for the *computation problem*, the *optimization problem* (OP) defined as a special case of the SP, focused on the process of finding the *best possible* solution for computation problem [44].

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 on Figure 2.2. It accepts the input information with its *inputs* X_n, which are sometimes called variables or parameters, processes them performing some *task* and produces the result on its *outputs* Y_m.

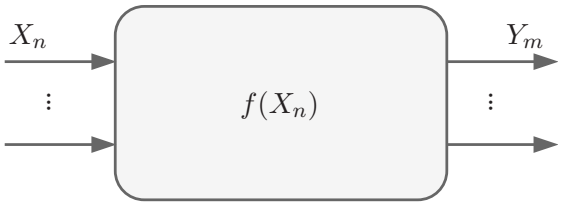


Figure 2.2 Optimization Target System.

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

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

Yevhenii: figure for classification

Next, let us highlight the target system characteristics. In works [2, 11, 26, 39] dedicated to solving the OPs, the authors distinguished OP characteristics that overlap through each of these works. Among them, we found the following properties to be the most important ones:

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

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

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

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

In this thesis we tackle practical combinatorial problems, where the most prominent examples are *bin packing* [73], *job-shop scheduling* [14] or *vehicle routing* [99] optimization problems. All combinatorial problems are *NP-Complete* meaning they are in both *NP* and *NP-Hard* complexity classes[41]. 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 [65] 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[22]. 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 [11].

Evidently, there exist a lot of different facets for optimization problem solvers classification, but they are a subject of many surveying works [11, 36, 55]. 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 [106], 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 [40], the objective of an exact algorithm is to perform much better (in terms of running time) than the exhaustive search. In both works [40, 106] 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, significantly reduces the running time. A simple example from [106] 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* [88] 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 [41]) 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 [105]. 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 [105] 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 [105]:

- **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 [105].

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 [81].
- **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 [9].
- **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 [18].

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 beats 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 the 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 therefore, one should consider them up to own risk.

As we mentioned in the 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 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 imply the following concepts:

- **Neighborhood**, which defines the set of solutions that could be derived performing single step of 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 TS and obtained from the previously found solutions, to control a problem-solving process [81].

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 by the authors in [19]: *constructive* and *perturbative*.

The first type implies the heuristics which construct the solutions from its parts step by step. The prominent example of constructive approach is the *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 *local search*, implies heuristics which operate on the completely created solutions, perturbing them. The simple example of local search is the *hill climbing algorithm*: evaluate the entire neighborhood, move to the best found solution and repeat. 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. Another instance of perturbation algorithm is the *first improvement local search* [104]. This heuristic accepts a better solution as soon as it finds it, during the neighborhood evaluation. The advantage of this methodology over the vanilla hill climbing is the search space traversal velocity.

Indeed, since the optimization result is fully defined be 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 [105].

2.2.2 Meta-Heuristics

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

The research in MHs field had arisen 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 [94] 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 the *higher level* when compared to simple problem dependent heuristics. A typical meta-heuristic structure could be imagined as *n-T—H* (template and hook) framework variation pattern. The template part is stable and problem independent, it forms the core of an algorithm and usually exposes *hyper-parameters*, which could be used for the algorithm tuning. The hook parts are domain dependent and that is why should be adapted for problem in hand. Later, T operates on the set of Hs to perform an optimization. Many MHs contain stochastic components, which provide abilities to escape from local optimum. However, it also means that the output of meta-heuristic is non-deterministic and it could not guarantee the result preciseness [15].

The meta-heuristic optimizer success on a given OP depends 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 local optima. In most cases, it is possible to decompose MH onto simple components and clarify, to which of competing processes contributes each component. The disadvantage of the simple heuristic approaches mentioned above is high exploitation dominance, since they simply do not have the components contributing to exploration.

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 most of them are inspired by real-world processes — physics, biology, ethology, and even evolution.

Meta-Heuristics Classification

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

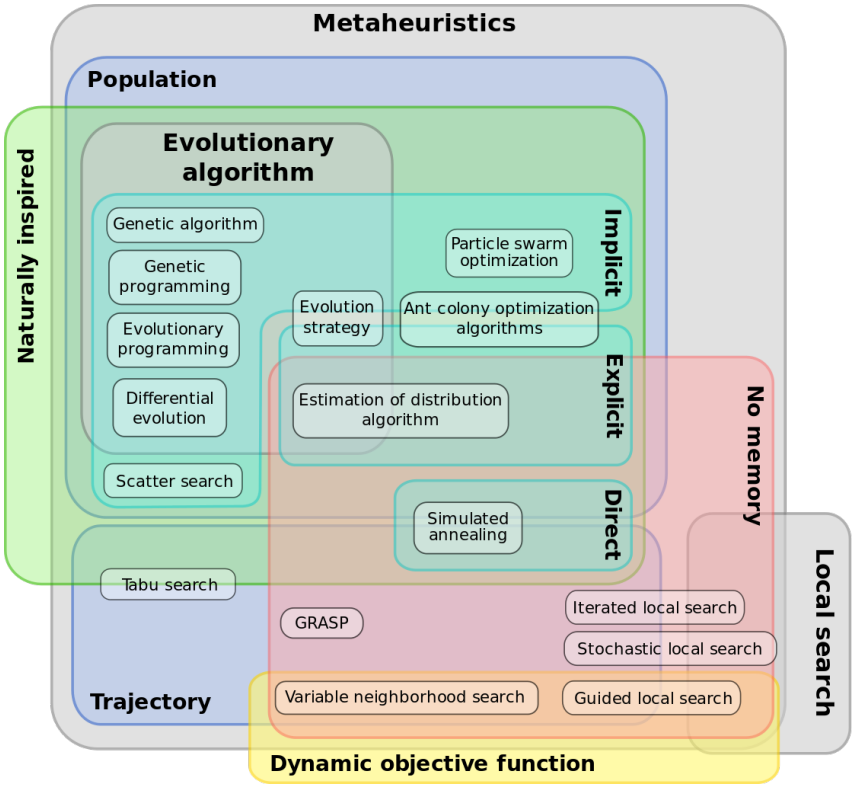


Figure 2.3 Meta-heuristics Classification.

As an example, the authors in [12] highlight 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 on the neighborhood where such prominent examples as the *iterated local search* [72] or the *tabu search* [43] do exist. The second one allows large jumps in the search space, where the examples are such MHs as the *variable neighborhood search* [46] or the *simulated annealing* [61].
- 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. The evolutionary algorithms [32], the ant colony optimization [28] or the particle swarm optimization [59] 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. The tabu search explicitly uses memory in form of tabu lists to guide the search, but the simulated annealing is memory-less.
- The **neighborhood structure** could be either static or dynamic. Most local search algorithms, such as the simulated annealing and tabu search are based on the static neighborhood. The variable neighborhood search is an opposite case, where various structures of neighborhood are defined and interchanged while the algorithm solves the OP.

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

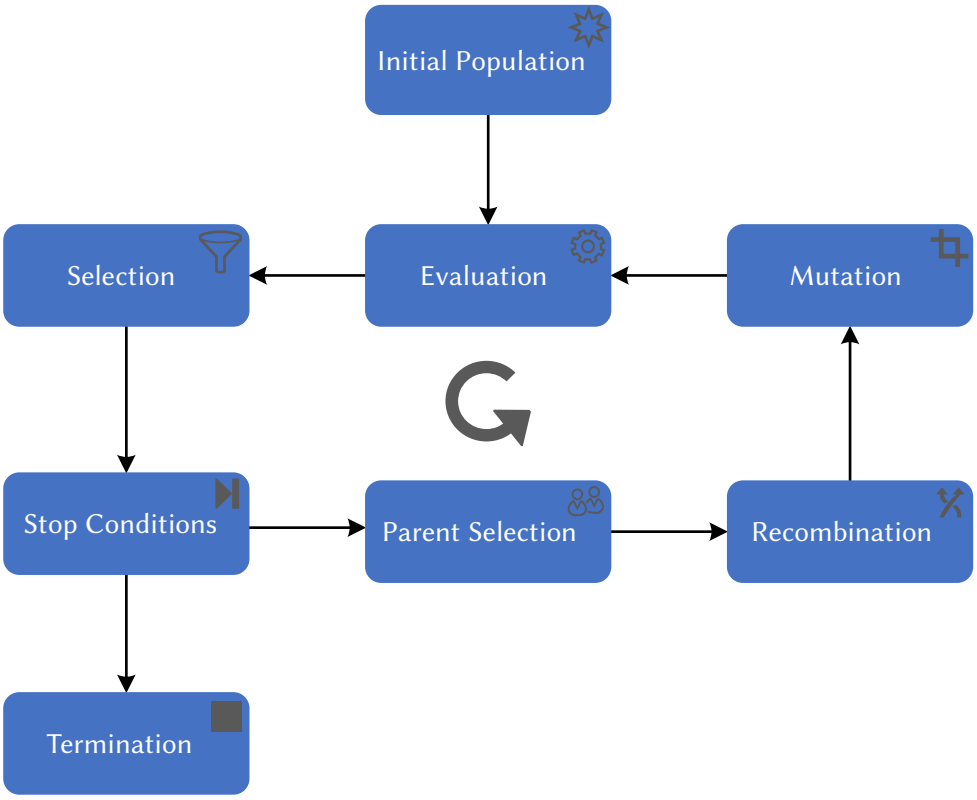


Figure 2.4 Evolutionary Algorithms Workflow.

we did not mention [23].

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 as the LLH in hyper-heuristic, described in the **??**.

Evolutionary Algorithms (EAs). They 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 [32].

Tree basic actions are defined as operators of EAs: the *recombination* operator selects the parent solutions, which later will be combined to produce the new ones (offspring); *mutation* operator, when applied to solution, creates a new and very similar one. Applying both operators, algorithm creates a set of new solutions — the offspring, whose quality is then evaluated on TS. After that, the *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 reached, the number of TS evaluations exceeds, or the solution with required quality is found. The work-flow of EA depicted on the Figure 2.4.

The well-known examples of EAs include the *genetic algorithm* [90], *genetic/evolutionary programming* [64], *evolution strategies* [8], 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 the probability p_c and produces children. Then, newly created Solutions undergo the mutation operator with the independent probability p_m. The resulting offspring perform a tournament within the selection operator and μ survivals replace the current population [31]. Distinguishable characteristic of vanilla GA is the usage of following operators: bit-strings solution representation, one-point crossover recombination, bit-flip mutation and generational selection (only children survive).

Evolution Strategy (ES), comparing to GA, are 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 selecting only among offspring highly enriching the exploration, and $(\mu + \lambda)$ — include also the predecessor solutions into selection, which is often called the *elitist selection* [31]. In many cases, the ES utilizes a very useful feature of *self-adaptation*: changing the mutation step sizes in a runtime, which we will discuss in dedicated to Parameter Control 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 [103]. An annealing technique imposes a globally minimal internal energy state and avoids locally minimal semi-stable structures.

The SA treats the search process as the 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 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 quality loose 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* [15]. 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 parameter control 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 the 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 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. But, 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 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 initial state of the other.

As for the simple heuristics, we introduce some examples of performed hybridization in order to provide the reader a better understanding of useful for the hybridization parts of algorithms and influence of the aforementioned balance on the search process.

Hybrid-Heuristics Examples

Guided Local Search and Fast Local Search. The main focus of 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 local optimum — the higher penalties. A local search procedure is carried out by the FLS algorithm, where the main advantage is a quick neighborhood traversal. It is done by braking it up into a number of small sub-neighborhoods. Afterwards, by performing the depth first search over these sub-neighborhoods, it ignores those, without an improving moves. At some point of the time FLS reaches the local optimum and passes back the control in GLS to update the penalties and repeat the iteration [100].

Direct Global and Local Search. This hybridization consists of two stages: the stochastic global coarse pre-optimization and the deterministic local fine-optimization. In the first stage, authors apply one of the two mentioned earlier meta-heuristics: the Genetic Algorithm or the Simulated Annealing. The 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 made in the last few iterations. Then, the pattern search algorithm [48] 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 local optima [96].

Simulated Annealing and Local Search. After the brief explanation of previous two hybrids, an observant reader hopefully guesses, what is happening in this hybridization. The authors in their work [74] called this method the 'Chained Local Optimization'. Thus, it is a yet another representative of staged hybridization. Iteration starts with the current solution perturbation, called *kick* in [74], referring a dramatic change of current position within the search space. Afterwards, the hill climbing algorithm is applied to intensify the obtained solution. When reached the local optimum, hill climber passes the control flow back to the simulated annealing for acceptance criteria evaluation, which finishes the iteration.

EMILI. Easily Modifiable Iterated Local search Implementation (EMILI) is a framework system for the automatic generation of new hybrid stochastic local search algorithms [80]. EMILI is a solver for permutation flow-shop problems (PFSP), also known as flow shop scheduling problems [86]. In PFSP the search of an optimal sequence of steps to create products within a workshop is performed. In this framework, the authors have implemented both generic algorithmic- and problem-specific building

blocks. They also have defined grammar-based rules for those blocks composition and used an automatic parameter tuning tool IRACE [70] 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 (make-span, 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.

Yevhenii: or we can move it into the hyper-heuristics, stating that we think, it should be there?

Yevhenii: continue proofreading here

2.2.4 No Free Lunch Theorem

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

All search algorithms perform exactly the same, when the results are averaged over all possible Optimization Problems. If an algorithm is gaining the performance in one problems class, it loses in another class. This is a consequence of so-called **No Free Lunch Theorem for Optimization** (NFLT) [107].

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

2.2.5 Hyper-Heuristics

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

The term **Hyper-Heuristic** (HH) was defined to describe an approach of using some *High-Level-Heuristics* (HLH) to select over other *Low-Level-Heuristics* (LLH) and apply them to solve the *class of Optimization Problems* rather than particular instance. Indeed, scientists report that the combination of different HLH produces better results than if hey were applied separately [29] — note previously discussed hybrid-heuristics. This behavior can be explained by the nature of search process and how it evolves in time. When you apply a heuristic, it sooner or later converge to some extreme point, hopefully global optimum. But it is 'blind' to other, not visited regions in the Search Space. Changing the trajectory of investigation by (1) drastically varying the Neighborhood, (2) changing the strategy of

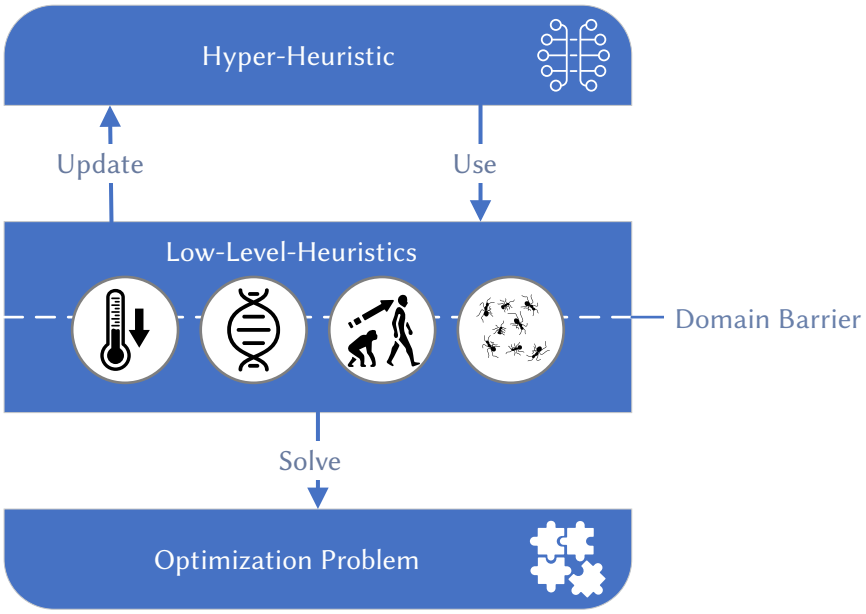


Figure 2.5 Hyper-Heuristics.

Neighborhood exploration and exploitation could (1) bring you to those previously unreachable zones (2) in more rapid ways. However, usually it is hard to predict how one LLH will behave in every stage of the search process in comparison to another. In Hyper-Heuristics, this job was encapsulated into the HLH and performed automatically.

In [77] authors made infer that Hyper-Heuristics can be viewed as a form of Reinforcement Learning, which is a logical conclusion especially if we rephrase it to *Hyper-Heuristics utilize Reinforcement Learning methodologies*.

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

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

Classification

Although, the research in Hyper-Heuristics field is actively ongoing, many algorithm instances were already created and some trials to organize approaches were conducted in [19, 29, 89].

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

In this section we present the union of those important Hyper-Heuristics classification facets to better justify the goal of this thesis.

We begin with the two broadest classes, which differentiate HH **routine**, also called as **nature of High-Level-Heuristic Search Space** [19, 20, 29]. The first class are Hyper-Heuristics to *select* Low-Level-Heuristic, in other words *Selection Hyper-Heuristic*. All our previous references to Hyper-

[1] Icons from thenounproject.com

Heuristics were made this concrete type. These algorithms operate in the Search Space, defined by complete and rather simple Low-Level-Heuristics that solve Optimization Problem. The task of HLH here is to pick the best suited LLH (or sequence of LLHs) based on available prior knowledge and apply it to the OP underway. Note, that staging Hybrid-Heuristics could be viewed as Solutions of Selection HHs. Hyper-Heuristics of the second class seek to *construct* LLH following some predefined receipt and using the atomic components of other heuristics as Lego bricks. The other commonly used name here is *Construction Hyper-Heuristics*. These approaches often lead to creation of new and unforeseen heuristics that are expected to reveal good performance while solving the problem in hand.

Next, the distinction in **nature of Low-Level-Heuristics Search Space** arises. In other words: "How does the LLH derive Solutions for the OP?" Authors in [19, 20, 29] distinguished *construction* LLHs where Solution creation happens each time from scratch and *perturbation* LLHs where new Solutions created from parts of already existing ones.

The other broadly used characteristic is the **memory usage method**. From this perspective we distinguish Hyper-Heuristics in which the learning happens *on-line*, *off-line* or learning mechanisms are *not present* at all [19, 89].

- In **on-line** case, the HH derives an information, used to select among LLH, while those LLH are solving the problem.
- In **off-line** case, the learning happens before solving concrete Optimization Problem. Here one should first train an HH solving other homogeneous problem instances by underlying LLHs (off-line learning phase). After that, the HLH will be able to choose among LLHs, thus be applicable to problem in hand (on-line use phase). Note, that this approach also requires creation of meta-features extraction mechanism and its application to every Optimization Problem.
- There exist also **mixed** cases, where learning happens first in off-line and later also in on-line phase. Definitely it is a promising (in terms of results quality) research direction, despite it high complexity.
- In the last case **no learning** mechanisms present, therefore HLH here performs some sort of Random Search over LLH Search Space. At first sight, it may look like weak approach, but looking onto Variable Neighborhood Search Meta-Heuristic we would doubt it.

For more detailed analysis, description, other classification facets and respective Hyper-Heuristic examples we encourage reader to look into resent classification and surveying researches [18, 19, 29, 89].

Hyper-Heuristics Instance Examples

Hyper-Heuristic for Integration and Test Order Problem [45]. *HITO* is an example of generational 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 [30] using interchangeably Choice Function (in form of weighted linear equation) and Multi Armed Bandit based Heuristic to balance exploitation of good components and exploration of new promising ones.

Markov Chain Hyper-Heuristic [75]. *MCHH* is an on-line selective Hyper-Heuristic for multi-objective continuous problems. It utilizes reinforcement learning techniques and Markov Chain approximations to provide adaptive heuristic selection method. While solving the OP, *MCHH* updates prior knowledge about the probability of producing Pareto dominating Solutions by each underlying LLH

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

Hyper-Heuristics Frameworks Examples

Hyper-Heuristics Flexible Framework [78]. *HyFlex* 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, Traveling Salesman and Vehicle Routing problems), such as problem and solution descriptions, evaluation functions and adaptations for set of Low-Level-Heuristics. The set benchmarks and comparison techniques to other built HHs on top of *HyFlex* are included to framework as well.

The intent of *HyFlex* creators was to provide Low-Level features that enable others to focus directly on High-Level-Heuristics implementation without need to challenge other minor needs. 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 [29, 76, 89] or on the CHeSC 2011 challenge website[1] (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** [95] is a construction hyper-heuristic framework aiming to extract information from the OP search domain for identification of promising components in form of object-oriented analysis.
- **hMod** [102] 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 pre-built heuristics.
- **EvoHyp** [83] framework focuses on hyper-heuristics created from evolutionary algorithms and their components. Here 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 remind you pros and cons of each heuristic level.

On the basis remain Simple Heuristics with all their domain-specific knowledge usage and particular tricks for solving problems. Usually, they are created to tackle a concrete problem instance in hand applying simple algorithmic approach. The simplicity of application and usually fast runtime paid back by 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 concrete problem and parameter tuning for better performance still should be performed.

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

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

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

2.3 Setting Algorithm Parameters

The most of 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 possible result quality.

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

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

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

3. **Configuration** is the unique combination of parameter values, sufficient to run TS.

4. **Search Space** is the set of all possible Configurations for defined Parameters.

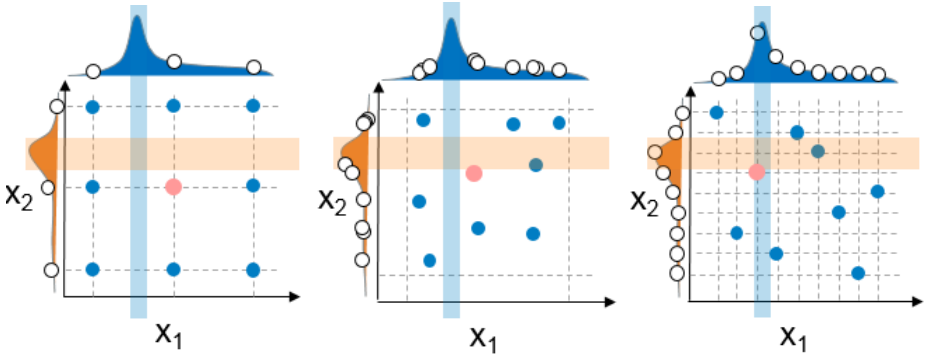
In this thesis we use notions of *Parameter* and *Hyper-Parameter* (HP) interchangeably, since the approaches discussed in this section are generally applicable also in Machine Learning cases. For instance, consider Neuron Network (NN). Hyper-parameters in this case will specify the structure (number of hidden layers, units, etc.) and learning process (learning rate, regularization parameters values, etc.) of network and changing them dramatically affect performance and results.

One frequent incumbent of Optimization Problems is **Parameter Settings Problem** (PSP) — searching of hyper-parameter values that optimize some characteristic of TS. When talking about NN example, PSP could be defined as task of maximization network accuracy in given dataset (Single Objective PSP). Taking into account a number of TS characteristics simultaneously, such as training time and prediction accuracy PSP transforms into Multi-Objective PSP.

The same applies to heuristics: proper assignment of hyper-parameters has a great impact on exploration-exploitation balance and thus on overall algorithm performance.

Up until now there were formalized many approaches for solving task of settings hyper-parameters. One of the simplest and error-prone ways is just trusting your (or someones else) intuition and using those parameters that seems more or less logical for particular system under the problem instance. People quickly abandoned it in favor of automatic approaches, fortunately novel computational capacities easily provide a possibility for it. These automatic methods later could be split onto *off-line or parameter tuning* and *on-line or parameter control* techniques.

Yevhenii: interesting visualization scheme on page 366 *https* : *//link.springer.com/chapter*/10.1007/978 − 3 − 642 − 29178 − $4_3$7



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

Figure 2.6 Automated Parameter Tuning Approaches.

2.3.1 Parameter Tuning

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

In this part of thesis we briefly outline existing automated approaches for parameter tuning illustrating them on the Figure 2.6[1]. In this example, the TS exposes two parameters: X_1 and X_2. Each graphic shows dependencies between X_1 (horizontal axis) and X_2 (vertical axis) values and the subject of optimization along those axes (here depicted the maximization case). The best found configuration by each approach is 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 brute-force algorithm. The set of all possible configurations (parameter sets) for relaxed problem is derived by specifying a finite number of possible values for each hyper-parameter under consideration. After evaluating all configurations on TS, the best found solution is reported. As you can see, this approach could skip promising parts of search space (Figure 2.6a).

Random Search Parameter Tuning. This methodology relies on random (often uniform) sampling of hyper-parameters and their evaluation on each iteration. At first sight, it might look unreliable to chaotically traverse the search space. But empirical studies show that with growing number of evaluations this technique starts to outperform grid search [6]: compare the best configurations (highlighted in pink) found by grid (Figure 2.6a) and random search (Figure 2.6b) techniques.

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

Naturally, this optimization problem could be tackled by almost every discussed here approach, however taking into account the facts that (1) TS here in most cases is a *black-box* we eliminate exact

[1] Original graphics are taken from [62]

and approximate solvers, (2) the evaluation cost is huge, thus it is not desirable to apply any of described above heuristics directly.

With this idea in mind, researchers started to (1) create Bayesian optimization algorithms that traverse the search space more efficiently and (2) build models that could recreate the dependencies between parameters and objective values, a so-called *surrogate models*. While the former direction is nothing else but an enhancement to already existing optimization techniques, the later one is crucial for problems where objective is expensive to evaluate. The later is some sort of enhancement used in combination with former enabling you to simulate evaluation of real system instead of expensive direct evaluations. Still, it is a common approach to combine previously reviewed search space traversal techniques, such as evolutionary algorithms, simulated annealing, tabu search with surrogate models for optimization.

2.3.2 Systems for Model-Based Parameter Tuning

The parameter tuning is an obligatory task when the maximum system performance is a mast-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 system under estimation.

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

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

- **Conditional parameters support** is (1) a provided for user and (2) handled by tuning system ability to describe conditional dependencies between hyper-parameters. As an example, imagine *crossover type* parameter of Genetic Algorithm that can take only some specific values: *Partially Mapped Crossover* (PMX), *Cycle Crossover* (CX), etc. Binding concrete crossover type, one will be required provide parameters for this crossover type, but eliminate respective parameters for other crossover types. This type of dependency could be described in form of parent-child relationship, however other types of dependencies also exists.

- **Parameter types support** is one of the basic usable tuning systems requirements. More concretely, Target System parameters could be not only numerical (integer or fractional), but also categorical in form of strings, boolean values, etc. Considering categorical data types, they could be either *nominal* (depict only possible atomic values) or *ordinal* (implies also value comparison, but no distance notion). For instance lets again analyze parameters of Genetic Algorithm: population size — numerical integer in range $[1...\inf)$, mutation probability — numerical fractional $[0...1]$, crossover type — categorical nominal PMX, CX. Indeed, we could treat population size as a set of finite values $10, 100, 1000$ — categorical ordinal parameter type.

- **Extensibility** is crucial when someone would like to try a new promising and intriguing learning algorithm for guiding a search, that was not available in parameter tuning system yet. Practically, one may need not only new learning algorithm, but some other features like non-trivial stopping criterion, tools for handling stochastic behaviors, or different strategies for random sampling (which are utilized while tuning system is learning before making a prediction).

- **Parallel Evaluations** required for utilizing available computing resources that could scale horizontally, thus providing simultaneous evaluation of multiple Configurations which speeds-up the learning process.

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

SMACv3 [50]

Sequential Model-based Algorithm Configuration (SMAC [1]) is a system for parameters tuning, developed by the AutoML research group (here we review a $3^r d$ 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 choice next configurations for evaluation. We found this term naturally formalizes the most of existing (to the best of our knowledge) parameter tuning approaches and may be used as a distinguishing characteristic of optimization algorithms (since they naturally could be applied not only to parameter tuning problems).

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

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

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

The evaluation (3) here carried out by the original ROAR mechanism, where the evaluation of each new candidate solution continues until enough data (from benchmark set of problem instances) obtained to either replace the current solution or reject the candidate. In a contrary to model-less ROAR, SMAC at step (1) builds the regression random forest — an instance of machine learning algorithm [16]. 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) the iterative local search (ILS) heuristic applies in combination with *Expected Improvement* (EI) evaluation (form of the Bayesian optimization) [92]. ILS starts at the best previously found configuration and traverses its neighborhood distinguishing between configurations using EI and regression model built at step (1). EI is large for those configurations, which has low predicted cost and for those, with high uncertainty in results therefore, naturally providing the exploration-exploitation balance [54].

IRACE

IRACE [2] is hyper-parameter tuning package [70] as the implementation of Iterated Racing Algorithm [13].

The underlying methodology is somehow similar to one implemented in SMACv3 and comprise three main steps: (1) sampling new configurations using a prior knowledge, (2) empirically finding the best ones among sampled using the racing algorithm and (3) updating the prior knowledge to bias future samples in towards better configurations. The prior knowledge here is represented as the probability

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

[2] IRACE GitHub repository `github.com/MLopez-Ibanez/irace`

distributions of values for each parameter independently (truncated normal and discrete distributions for numeric and categorical hyper-parameters respectively). During update step (3), the probability distributions are build using the best found in (2) step configurations therefore, increasing the sampling possibilities for the promising values.

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 reaching the required number of survivals, or after solving a requiring 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 solved by different underlying distributions, the conditional relationships are handled by the dependency graphs. During sampling (1), firstly the non-conditional parameters are sampled and only afterwards, if respective conditions are satisfied, the dependent parameters are sampled.

HpBandSter

A distributed Hyperband implementation on Steroids (HpBandSter [1]) is the realization of BOHB algorithm [34] in the software framework. While SMAC outperforms and partially reuses the decisions made in ParamILS, 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 name, the SMBO routines in this framework are carried out with mainly two algorithms: (1) learning and configurations sampling done by the Tree Parzen Estimator (TPE, Bayesian Optimization technique), while (2) evaluation of sampled configurations and their comparison are carried out by the Hyperband (HB) algorithm.

The TPE usage instead of naïve Gaussian Processes-based (GP BO) Bayesian Optimization + Expected Improvement evaluation was motivated by a better dimensional scaling abilities and internal support of both numerical and categorical data 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 parameters distributions. It splits the configurations into two sets according to their 'goodness'. During the sampling, it proposes those parameters, which have 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 [7].

The central part of BOHB, namely the HyperBand is a promising bandit-based strategy for hyper-parameter optimization [68] in which the *budget* for parameter tuning is defined beforehand and divided between iterations. The role of budget could play any control parameter that denotes the accuracy of configuration evaluation by TS, where estimation with the maximum budget gives you the most precise configuration evaluation, while the minimum amount of budget results in the least accurate approximation of configuration evaluation result. The running examples of budget could be a number of iterations in iterative algorithm, a number of epochs to train the neuron network, or a number of problem instances from benchmark set to evaluate. As the result, requirements arise for TS to expose and support budget usage as expected in BOHB.

At each iteration, HB samples uniformly at random a number of configurations. 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

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

produce a higher number of more precise measurements. Each iteration of HB is split to the number of *Successful Halving* (SH) procedure executions which drop poorly performing configurations (usually 2/3) at each execution. As one could 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 Tree Parzen Estimator is made in several places. Firstly, the learning models are updated each time, when new results are available for every budget value. Next, at each HB iteration instead of random sampling, the TPE model is used to pick next configurations. Note that *BOHB* uses only those models, which built on configurations obtained for the largest budget. This decision results in more precise surrogate models (parameter distributions) 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 in hand, but also on the general usability and adaptivity to unforeseen tasks. Here we review a 2^{nd} version of BRISE[1] [84], since the very early BRISE versions (major version 1 [85]) were more monolithic and hard to apply for parameter tuning problem in hand.

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

Before reviewing each perspective, it is worth to justify the central terms used in system. Note, that some of them are pretty the same as we defined above, but here most of them also were implemented in form of classes.

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

2. *Configuration* is a combination of input parameter values for target system, or algorithm under tuning. It could be run several times to obtain a statistical data therefore, contain number of *Tasks*. Naturally, the configurations are comparable in terms of their runs results.

3. *Parameter* is a meta-description of concrete configuration part and a building block for search space. It defines a set or range of possible values.

4. *Search Space* comprises all parameters and their dependencies and could verify the validity of configuration.

5. *Task* is an evaluation of TS under provided configuration for specified in description problem scenario.

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, *worker-service* to distribute tasks between set of worker-nodes, *front-end-node* to control and report optimization process on a

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

web-page, and non-obligatory *benchmark-node* that could be handy for executing and analyzing number of experiments.

The main-node is a combination of objects, which interact in terms of queue callbacks. Therefore, as soon the new configuration is evaluated, the new model is built and used for the prediction using of all available information. The intent of introducing mentioned above terms is to use them as a core of 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 system.

2.3.3 Parameter Control

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

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

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

The only one broad classification facet we able to distinguish is the *type of control mechanism*, where the deterministic and adaptive strategies are distinguishable. The first type suggests changing the parameters in a predefined schedule, while the last approach assigns the parameter values upon received feedback. To the best of our knowledge, the adaptive approaches are mostly depended 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 [49].

Parameter Control in Simulated Annealing

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

The cooling rate parameter control 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 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 [56] the authors proposed an adaptive strategy to change the cooling rate, based on the statistical information, evaluated on each optimization step. Concretely, if the statistical analysis, named in research as a *heat capacity*, shows that the system is unlikely to be trapped in local optima — cooling rate is increased. In a contrary, it is decreased if the possibility of being trapped is high.

The study in [42] proposes the adaptation of other hyper-parameter — an acceptance criteria. The proposed mechanism is based on thermodynamics fundamentals, such as an *entropy* and a *kinetic energy*. Authors suggest replacing the standard acceptance criteria (based on the current temperature and the solution quality) with the one based on evaluation of the solutions entropy changes.

Many researches were made to investigate, which is the best between deterministic and adaptive strategies [5, 25, 52, 71, 97]. In many cases, authors conclude the adaptive methods provide more robust and promising results.

Parameter Control in Evolutionary Algorithms

While searching the parameter control examples in heuristics, one will find dozens of proposed methodologies for the evolutionary algorithms. It is arising from the fact that an idea of changing the algorithm parameters dynamically came from in EAs [58]. The motivation for such number of performed studies lays in the strong dependence of an algorithm performance on parameter values.

The deterministic and adaptive mechanisms in EAs are extended by the 3^{rd} type — the *self-adaptive* approach. It implies an encoding of parameter values in the solution genomes therefore, allowing them to co-evolve with the solutions in runtime [27]. All the proposed strategies could be split into two families: one includes the algorithms proposing to adapt a concrete parameter solely and the other, which includes the approaches to control a group of parameters. In EAs the commonly implicated hyper-parameters are the *population size*, the *selection strategies* and the *variation aspects* (namely the crossover and mutation operators). In case of interest, we propose the reader to analyze recently conducted reviews and researches dedicated to the parameter control in evolutionary algorithms [1, 27, 58, 93].

There is one rather intriguing parameter control approach proposed for the evolutionary algorithms. In [57] the authors introduce a Reinforcement Learning (RL) parameter controller, which goal is to select the EA parameters on-line, evaluating a set of simple observables. They include: a genotype diversity, a phenotype diversity, a fitness standard deviation, a fitness improvement and a stagnation counter. The RL in this work is executed following MAPE-K methodology [17]. On each iteration the observables are **m**onitored, their values are **a**nalyzed to build a parameter control **p**lan, which is **e**xecuted in the next iteration. The letter **k** denotes the central part of thus methodology — the knowledge. This set of proposed observables could be split onto two logical groups. One is the algorithm specific with the genotype/phenotype diversities and the fitness standard deviation, while the other is algorithm independent and includes the fitness improvement and the stagnation counter. We believe the proposed RL approach could be applied to other algorithms, with the only requirement in exposing the observable knowledge. The proposed in [57] parameter control methodology may one of the first to generalize a parameter control techniques and later in Section 3.1 we use it as a part of our approach.

2.3.4 Conclusion on Parameter Setting

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

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

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

3. The decision on concrete technique usage is the 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 in hand.

In this section we review one particular representative of automatic machine learning systems. Based on ML framework Scikit-learn [82], Auto-sklearn system [37] operates over number of classifiers, data and features preprocessing methods **including their hyper-parameters** to construct, for given dataset, the best performing (in terms of classification accuracy) machine learning pipeline. This problem was formalized as *Combined Algorithm Selection and Hyper-parameter tuning problem* (CASH) and presented previously in Auto-WEKA [98] system. Intuitively it could be rephrased as follows: "For given optimization problem, find the best performing algorithm and its hyper-parameters, among available, and solve the problem".

Note, how Auto-sklearn is in some sort similar to Hyper-Heuristics, which use LLHs for traversing a Search Space and solving the OP. CASH problem seems to us as union of problems solved by Hyper-Heuristics and Parameter Tuning approaches. We also found that the Architecture Search problems [33] (related to Neural Networks) are nothing else, but particular case of CASH.

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

During the off-line phase, for each of available dataset, published by OpenML community [38], the search of best performing machine-learning pipeline was done using Bayesian Optimization technique implemented in discussed previously SMAC [50] framework. After that, the *meta-learning* executed to conduct the meta-features for each dataset. The entropy of results, data skewness, number of features and their classes is a sparse set of meta-features used to characterize a dataset (overall number is 38).

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

During the on-line phase, the other crucial improvement was introduced. Usually, while searching the best-performing pipeline, a lot of effort spent to built, train and evaluate intermediate ones. After each evaluation, only the results and pipeline description are stored, but the pipeline itself is discarded. In Auto-sklearn, however, the idea lays in preserving previously instantiated and trained pipelines, obtained while solving the CASH problem. Later they are used to form an ensemble of models and tackle final problem together. This mean, that the results of this architecture search is a set of models

with different hyper-parameters and preprocessing techniques, rather than one model. This ensemble starts from the worst performing ones (obtained at the search beginning) and ending by the best suited for dataset in hand. Naturally, their influence on final results are weighted.

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

In general, the empirical investigation of proposed approach universality would be rather intriguing, since the only cases of Auto-sklearn application we found are the classification tasks, but not a regression problems [10, 37].

The field of automated machine learning is one of trending research directions, that is why there exist dozens open-source systems, such as *Auto-Weka* [98], *Hyperopt-Sklearn* [63], *Auto-Sklearn* [37], *TPOT* [79], *Auto-Keras* [53], etc. Among open-source, there are many commercial auto-ml systems, such as *RappidMiner.com*, *DataRobot.com*, Microsoft's *Azure Machine Learning*, Google's *Cloud AutoML*, and Amazon's *Machine Learning on AWS*.

2.5 Conclusion on Background and Related Work Analysis

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

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

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

The outcome of No Free Lunch theorem [107] can not be ignored, according to which no single algorithm can tolerate broad range of problems equally outperforming other solvers. That is why we can not set aside hyper-heuristics, which are designed to select the best suited, in particular case, problem solving algorithm.

The research in automatic machine learning made step further and are tended to combine both algorithm selection and parameter tuning problems into single Combined Algorithm Selection and Hyper-parameter tuning problem (CASH), formalized in [98]. The search space in CASH problem is formed of algorithm variants and their respective hyper-parameters. However, one solver can not use the parameters of another, thus the resulting search space happen to be 'sparse'. In general, the structure of CASH problem is almost the same as regular parameter tuning case. That is why the commonly used solver for CASH problem are systems for parameter tuning: SMAC in Auto-sklearn and Auto-Weka, Hyperopt in Hyperopt-Sklearn and so forth. Not many surrogate models are able to handle such search spaces: random forest machine learning model and Bayesian Optimization approaches with exotic kernel density estimators [67]. Even fewer optimizers are able to perform well in such 'sparse' spaces.

The other drawback, is that the CASH problem definition is limited to searching the algorithm and its parameters in the off-line manner.

Scope of thesis defined. At this point, we would like to define the scope of our thesis.

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

As CASH merge Algorithm Selection and Parameter Tuning techniques to get the outstanding performance, here we found a merge of Algorithm Selection and Parameter Control problems an intriguing and worth-to-try idea. Thus, in this thesis we are trying to achieve the best of both worlds, namely on-line Hyper-Heuristics and Parameter Control techniques. Doing so, the resulting approach should be able to solve the problem in hand, applying the best suited Low-Level-Heuristic and setting its parameters in run-time. With this idea in mind, we investigate a possibility of turning existing parameter tuning system into so called on-line selective Hyper-Heuristic with Low Level Heuristics Parameter Tuning.

3 Concept Description

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

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

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

3.1 Combined Parameter Control and Algorithm Selection Problem

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

In our work, we solve the parameter control problem utilizing a similar to proposed in [57] reinforcement learning (RL) approach for evolutionary algorithms. The underlying idea of RL could be described as a process of performing actions in some environment in order to maximize the reward, obtained after each performed action. To apply this technique onto the parameter control problem, we must define what are those *actions* and how to estimate the *reward*. Thus, for making the parameter control applicable to broad range of algorithms, we analyze not the solver state itself, but the optimization process (in [57], the authors use both algorithm-dependent and generic metrics). To realize the MAPE-K control loop, we must interrupt the solver, analyze the intermediate results, learn the current trend among parameters, configure the solver with the most promising parameter values and continue solving. The number of MAPE-K loop iterations i define the granularity of learning, where one should balance between *time to control* (TTC) the parameters vs *time to solve* (TTS) the problem. Naturally, the limitation of proposed approach is the use-cases, where $TTS >> TTC$.

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

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

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

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

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

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

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

During the review of model-based parameter tuning approaches (Section 2.3.1), we made a conclusion that all reviewed systems follow strictly the first idea. For instance, as the surrogate models, BOHB [34] and BRISE [84] use the Bayesian probability density models. Those surrogates could naturally fit to the described search space shape, but the proposed approaches are not able to make the predictions effectively, since the most of predicted configurations will violate the dependencies. As the illustration, imagine after i^{th} iteration, the surrogate models learn about two superior parameters: one indicates a well-performing heuristic type (the Genetic Algorithm), the other — an effective configuration for another algorithm type (an exponential cooling rate for the Simulated Annealing). In this case, the reviewed systems sampling methods will tend to predict the invalid configurations with those two parameter values.

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

3.2 Search Space Structure

When the time comes to selecting not only the solver parameters, but also the solver itself, the united search space no longer could be presented as 'flat' set of parameters since it tends to appearance vast amount of invalid parameter combinations. Let us estimate the number of all possible configurations vs the amount of meaningful ones. Suppose, we have the N_s number solver types, each exposing the $N_{s,p}$ number of hyper-parameters with the $N_{s,p,v}$ number possible values. The aggregated quantity of configurations N_c in the disjoint search spaces is calculated as the number of possible combinations using the 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 the Equation (3.3).

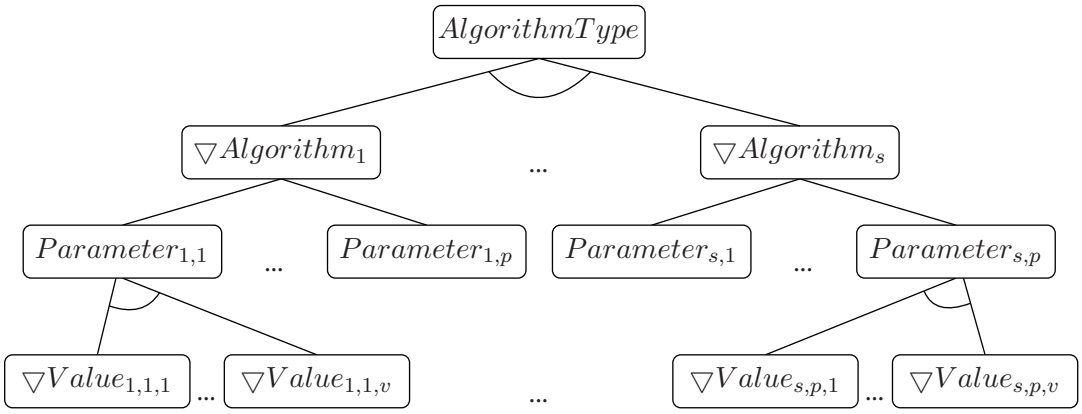


Figure 3.1 Search space representation.

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

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

To overcome this, we utilize similar to the utilized in IRACE [70] framework idea: *explicitly indicate the dependencies as a parent-child relationship among the search space entities p, firstly predict the parent parameter, afterwards — the children.* This give us an opportunity to threat the algorithm type as the regular categorical parameter, makes the search space structure uniform and simplifies the prediction process.

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

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

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

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

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

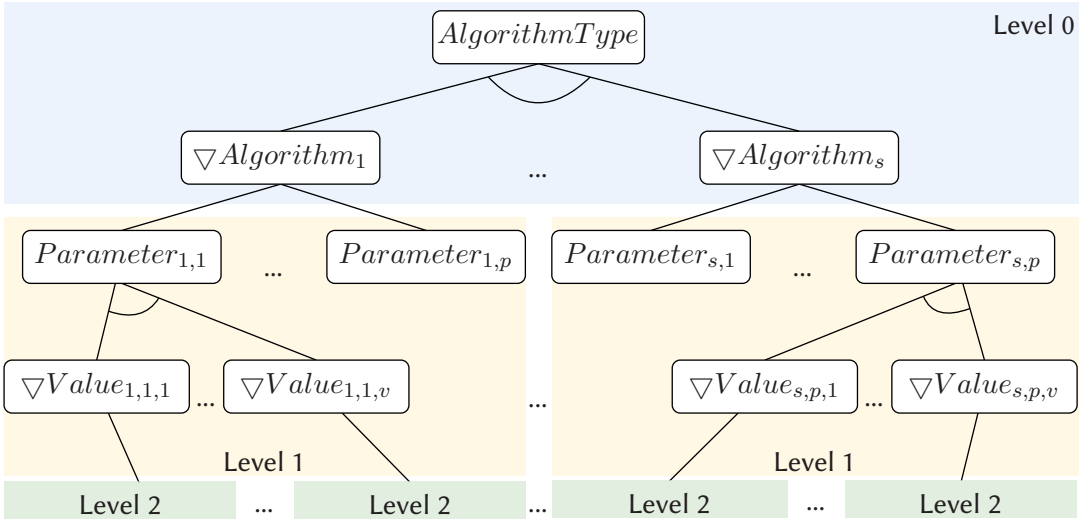


Figure 3.2 Level-wise prediction process.

3.3 Parameter Prediction Process

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

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

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

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

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

At this point, let us summarize the functional requirements.

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

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

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

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

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

- **In the prediction models:**

P.F.R.1 The **model encapsulation mechanism**, which should aggregate and hide the level-wise approach of the search space traversal and the feature ignorance as well. In 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.

3.4 Low Level Heuristics

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

The proposed idea of the MAPE-K reinforcement learning application, implies the usage of anytime algorithms (see classification of solvers in the 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 the Section 4.4, dedicated to LLH implementation.

3.5 Conclusion of concept

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

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

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

It is worth to mention that the proposed structure of search space representation is similar to the *Feature Model*, used to describe the Software Product Lines (SPL) [91]. On the Figure 3.1 and Figure 3.2 we used the notions from SPL feature models to denote an *alternative* parameter values. The process of configuration construction within a search space can be referred as the *Staged Configuration* in SPL.

4 Implementation Details

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

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

In the Section 4.1 we analyze the parameter tuning frameworks from a perspective of needed effort to fulfill the HLH requirements, listed in the Section 3.3. In a conclusion (Section 4.1.2) we outline the adaptations, needed to be done in the selected code basis. Afterwards, we split the former HLH implementation description into two logical parts, as we have done for the concept description. In the Section 4.2 we discuss the search space development, while the 4.3 is dedicated to the prediction process.

Finally, in the Section 4.4 we choose the LLH code basis for, present a set of reused meta-heuristics and their adaptations to fulfill our requirements listed in the Section 3.4 as well.

4.1 Hyper-Heuristics Code Base Selection

To start with the framework analysis, let us firstly outline the important characteristics from the implementation perspective.

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

The next is a *support for on-line optimization*. This is a bit complex characteristic of system that we are willing to distinguish. As it turns out, many parameter tuning systems require full evaluation of target system for configuration comparison. However, in case of desired hyper-heuristic, we treat the configuration evaluation as a trial to solve the problem in hand using particular LLH with tuned parameter. It implies an important ability of the optimization problem (OP) solving in a set-wise manner (Section 3.1).

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

4.1.1 Parameter Tuning Frameworks Analysis

SMACv3 The implementation of Sequential Model-based Algorithm Configuration framework is available on-line under the BSD-3 license.

The idea of SMACv3 lays in ROAR mechanism enhancement with the model-based sampling algorithm. As we mention in Section 2.3.2, underlying surrogate model are one of the random forest or Gaussian process kernel density estimator. These models could fit complex dependencies among parameters in

the search spaces. To choose a next configuration the *one-exchange* neighborhood of best found so far is traversed, using the surrogate models and the expected improvement estimations. The learning capabilities in surrogates are great, and the usage of expected improvement guarantees converging to the global optimum given enough time as well. However, the major drawback in this system is lack of ability to include the conditional dependencies between parameters into the sampling process. In fact, used here search space representation framework ConfigSpace [69] is able to specify the dependencies among hyper-parameters. However, to the best of our knowledge, the one-exchange neighborhood used as sample mechanism in SMACv3, is unaware of the parameter dependencies therefore, violates them during the configuration sampling resulting in illegal parameter combination. Those cases are rejected by the ConfigSpace, but we believe that in case of 'sparse' search spaces this approach could lead to ineffective sampling and struggling in system predictive abilities. Unfortunately, we did not find any officially published empirical studies of such cases and can only make guesses based on own intuition but, the SMACv3 developers advises for operation in such cases [1] could serve as an evidence to correctness of our assumptions. One of the possible solutions here could be the usage of a conditional-aware one-exchange neighborhood definition for sampling process.

The ROAR mechanism is a derivative from the FocusedILS algorithm (solver in the parameter tuning framework ParamILS [51]) where each evaluation of a new candidate solution on problem instance performed sequentially. Since the ROAR evaluation strategy is also used in SMACv3, we expect it to require much effort to enable system solving the problem on-line.

IRACE The IRACE framework implements the iterated racing algorithm to evaluate the set of configurations during the parameter tuning session(Section 2.3.2). It is distributed under the GNU General Public License and the source code is available on-line.

As the surrogate models, IRACE uses the probability distributions of those parameter values, which shown to be good during racing step. The prediction process is defined as the step-wise sampling in previously built distributions. It elegantly handles the conditions among parameters and illuminates the possibility of invalid configuration appearance. The framework completely relies on the racing algorithm for parallel evaluations and on *Friedman test* [24] or alternatively *paired t-test* for statistical analysis of racing configurations. Thus, we found it to be static in terms of variability and extensibility on the learning mechanisms. In terms of parallel evaluations, the algorithm utilizes all available resources at the beginning of each racing step, but as the process continues, fewer evaluations are executed in parallel those available resources are idling and not utilized optimally at all steps of IRACE execution.

As for the on-line problem solving, let us discuss the racing algorithm. As we described in Section 2.3.2, this step is executed with a (1) set of TS configurations under evaluation and (2) a benchmark set of optimization problems. Then, the TS starts to solve the problem set under each configuration, while racing algorithm drops the worst-performing configurations. It is possible to use a single problem instance however, divided into parts (from the perspective of TS allowed running time) instead of using a benchmark set. Doing so, it will be possible to adapt system for on-line problem solving cheaply however, the granularity of parameter (and LLH type as well) control will be reduced. The reason for such reduce is the amount of information obtained from race: only the best configurations are returned, leaving the performance evidences of others behind, which may used to create a more precise surrogate models. The only possible way to deal with this is to leave a racing algorithm and use the reinforcement learning.

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

[1] Visit GitHub repository of SMACv3 for more info `github.com/automl/SMAC3/issues/403`

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

Let us discuss the process of conditions between hyper-parameters handling. SMACv3 and HpBandSter as well, uses ConfigSpace framework for search space representation. As we discussed in SMACv3 description above, ConfigSpace naturally allows to encode the dependencies and conditions among parameters. The TPE learning models are also able to somehow fit these dependencies by using the *impuration* mechanism[67]. In short, when fitting the surrogate models, inactive parameters (turned off by means of dependencies) are treated with their default values. Later, while building the surrogate models those default parameter values are ignored therefore, the probability densities still represent a proper parameter values distributions. However, consider an appearance of two configurations sets: C_1 and C_2, such that some parameter P_i is forbidden in C_1, but required in C_2 and in a contrary the other parameter P_j is required in C_1, but forbidden in C_2. If these configuration families are turn to be superior, this will bias the densities towards P_i and P_j values. As a consequence, the proposed prediction mechanism will sample non-default parameter values for both P_i and P_j, which results in the configurations with violated parameter dependencies. The more 'sparse' search space, the more harming an effect will be from a prediction performance perspective. A possible treatment here is to change the sampling process, introducing an intermediate layer to perform a prediction in level-wise approach, suggested in the Section 3.3.

BRISEv2 BRISEv2 is a software product line (SPL), created with aim in 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 modular design of *main-node*. 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-box for all variable entities, but we focus our attention to implemented sampling process. The reason of such greedy review is that the underlying search space representation is carried out by the same ConfigSpace, while the provided surrogate models are ridge regression model with polynomial features and Bayesian Tree Parzen Estimator (TPE). We are not going to repeat ourselves reviewing the ConfigSpace + TPE combination, but we have to put a few words about the ridge regression.

Ridge is the machine learning linear regression model with parameters regularization [47]. Since, it is a linear model, its ability to fit a 'sparce' search spaces is poor therefore, the machine learning community are suggested to treat such cases with *conditional linear regression* [21]. The underlying idea is to split the search space into sub-search spaces and build a separate regression model, but to the best of our knowledge, this approach is not built in into the underlying ridge regression model.

As for the support of on-line problem solving, the routine of optimization process, implemented in BRISEv2 is nothing else, but the reinforcement learning approach. After each new obtained evidence, a 'fresh' surrogate model is built to react on the learning process by predicting of new configuration.

Which makes it easy to embed the on-line parameter tuning approach, presented in Section 3.1.

4.1.2 Conclusion on Code Base

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

To sum um our review, we use a term *quality* to aggregate both the provided out-of-the-box desired characteristic support and the required effort to adapt it, if necessary. We aggregate the reviewed characteristics quality in each software framework into Table 4.1 for the visual representation. The quality estimates are quantized into three ordinal values:

1. **Poor** quality, which implies the weak characteristic support and much effort required to provide it.

2. **Average** quality, which implies the weak support, but requires small amount of effort to provide it.

3. **Good** quality, which implies a great out-of-the-box characteristic support, and requires minor or no changes at all.

Table 4.1 Code basis candidate systems characteristics.

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

Among the reviewed software systems, the majority were created as an implementation of some concrete algorithm (or a combination of algorithms). It results in the reduction of system flexibility. It turns out that there are two most promising candidates for hyper-heuristic with parameter control creation: IRACE and BRISEv2. They both require much adaptation and preparation steps, which we will discuss in upcoming sections, but still less in comparison to SMACv3 and HpBandSter. Among such features as proper support of conditional parameters vs variability and extensibility, the former plays a settle role therefore, we choose BRISEv2 as the code basis.

4.2 Search Space

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

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

4.2.1 Base Version Description

From the structural point of view, in ConfigSpace[1] the parameters coupling is made implying parent-child relationship, which fit into our requirements. The parameter types suite the most of use-cases and the value-specific dependencies are supported as well. Therefore, from the structural requirements S.R.1, S.R.2 and S.R.3 (Section 3.2) are perfectly met.

When it comes to the functional requirements (Section 3.3), ConfigSpace samples the random configurations in a completion approach. In other words, there is no step-wise configuration creation, only a final and valid ones are produced. Thus, there is no way to expose the underlying dependencies among parameters for the prediction models, except of carrying them on a side and evaluating manually. As a consequence the data filtering mechanism should be implemented on a side and the sampling propagation as well. The framework exposes an ability to validate a created configuration, which turn to be also a proprietary class. As for the parameter description, the amount of exposed knowledge is satisfying. Here we conclude that all functional requirements, except S.F.R.4 are not met.

As the conclusion, we decided to set aside the $3rd$ party ConfigSpace framework because: (1) it still requires much adaptation and implies its usage as a core entity in BRISEv2, (2) it implies replacement of the other core entity — Configuration, which turns to be costly and (3) it obligates us to use a third proprietary entity — Hyperparameter.

4.2.2 Search Space Implementation

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

Yevhenii: add class diagram in appendix

In the code snippets provided through the explanation we highlight an implemented method signature, which fulfills one of our requirements, specified in Chapter 3.

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

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

Listing 4.1 S.R.1 implementation.

Note, currently we support a composite construct only by means of categorical parameter type therefore, requiring a list of activation categories and postponing composition on numerical ranges enhancement for the future work, since our current needs do not include the composition on numerical ranges.

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

Search space role in prediction. Imagine a number of configurations were already evaluated. For making the prediction in tiered approach, the parameter values on current level should be selected. Thus, first we filter data which fits to this level by means of search space S.F.R.1. Thereby, filter accepts the already chosen parameter values and iterates over the available configurations. It is implemented in form of hyper-parameter instance method in Listing 4.2. An intent of this method is to derive whether the already chosen parameter values ($base\_values$) form a sub-feature-tree of the parameter values under comparison ($target\_values$). The outcome of this method is a decision, should this particular configuration be included into the dataset or not. For instance, if the selected LLH type in $base\_values$ is not the same as one in $target\_values$, the result will be false.

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

Listing 4.2 S.F.R.1 implementation.

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

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

Listing 4.3 S.F.R.2 implementation.

Randomly sampled for current level values are then used for getting a description and (1) cutting-off the data from levels above and below, (2) building the surrogate models and (3) making the prediction.

To build the surrogate models we require an available data (parameters) description. Thus, S.F.R.3 implementation is performed in method $describe$ (Listing 4.4). This is once again a recursive method, which terminates when parameter can not find activated children or himself in the provided $values$. This description contains a mapping from parameter name to its type and range of possible values.

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

Listing 4.4 S.F.R.3 implementation.

Later, this description is used by prediction models for building surrogates and making the predictions, which will replace a randomly sampled parameter values in $generate$ method.

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

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

Listing 4.5 S.F.R.4 implementation.

4.3 Prediction Process

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

P.F.R.1 implies the addition of entity, which will encapsulate the prediction process, described in the Section 4.2.2. According to P.F.R.3, this entity may also be responsible for the forgetting strategy. Both requirements are not available in BRISEv2 yet therefore, we will implement them from scratch.

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

The following parts of prediction logic description are dedicated to (1) P.F.R.1 + P.F.R.3 implementation in form of $Predictor$ entity and P.F.R.2 in form of decoupled data preprocessing mechanism and the prediction models. Note, the current implementation does not solve the problem of binding surrogate models creation and optimization over them. We postpone this to the future work implementing a simple random search over the surrogate models.

4.3.1 Predictor Entity

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

To implement the information forgetting mechanism, we refer the idea of sliding window, mentioned in [35]. According to this, predictor should use specified in a settings number of the latest configurations as information for surrogate models creation. We modify this logic, allowing user to specify not only a static number, but also a percentage of the latest configurations (line 3). This fulfills the P.F.R.3 however, more exotic approaches may arise such as the statistical analysis to estimate a required configuration number or the other type of meta-learning.

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

```
class Predictor:
  def predict(measured_configurations):
    level_configurations = trim_in_window(measured_configurations)
    prediction = Mapping()
    while not search_space.validate(prediction, recursive=True):
      level_configurations = filter(search_space.are_siblings(prediction, x), level_configurations)
      randomly_generated = search_space.generate(prediction)
      full_description = search_space.describe(randomly_generated)
      level_description = trim_previous_levels(description, prediction)

      data = trim_accodring_to_description(level_configurations, level_description)
      model = get_current_level_model()
      model.build(data, level_description)

      if model.is_built():
        level_prediction = model.predict()
        if not search_space.validate(level_prediction, recursive=False):
          level_prediction = randomly_generated
      else:
        level_prediction = randomly_generated

    return Configuration(prediction)
```

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

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

4.3.2 Data Preprocessing

Data preprocessing may be split into two concepts: an obligatory data encoding and optionally data transformation. The first is required to make the underlying model capable with provided data. Imagine the parameters values to be a simple string. Having a surrogate model, which is constructed as probability densities of parameter values, one should first derive a numeric data for those string parameter values. The second concept is applied on a data, which is already suitable. This is usually done to improve an available surrogate model performance. An example of former could be simple indexing of all possible string values, which results in replacement of strings by their indexes during the data preprocessing. On a contrary, the later case may be presented by an addition of polynomial features to already available data with aim to improve surrogate models preciseness by learning more complex dependencies.

The decision of which encoding to use is strictly defined by the learning model, while the decision on data transformation is carried out by user and depends on the concrete use-case.

In all reviewed parameter tuning systems, the data preprocessing is implemented only by means of an obligatory for underlying learning models encoding and omitting the possible data transformation. In most cases, the encoding is performed in simple label into integer numbers. Being the easiest approach, this encoding may introduce a non-existing patterns in nominal data. For instance, having 3 possible LLH types genetic algorithm, simulated annealing and evolution strategy, the label encoding will encode such parameter values to numbers 0, 1 and 2 respectively. When the such data is passed to surrogate for learning, the model may interpret that GA is closer to SA than to ES in the search space. To prevent this, the other preprocessing type should be used for instance, binary encoding.

In any case, the intent of this discussion is to provide the reader an insight of data preprocessing importance, but the discussion of possible cases and their influence are out of this thesis scope. Here instead we decided to gain a certain level of flexibility by providing a uniformed wrapper for the

preprocessing routines, implemented in Scikit-learn machine learning framework [82].

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

4.3.3 Prediction Models

As a derivative from predictor implementation, the underlying prediction models should implement unified interface and behavior. Due to predictor, the models are acting on a 'flat' search space levels. This enables us to use a vast range of possible surrogates for instance, linear regression models. In fact, the previously used in BRISEv2 ridge regression with polynomial features is nothing else, but a combination of data preprocessing from the Section above and the ridge regression model from scikit-learn. Below 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) selection strategy proposed in [4]. This is motivated by a great reported performance of the selective hyper-heuristics, with MAB as HLH. However, it is worth to mention that MAB is applicable only to categorical parameters types.

The previously available in BRISEv2 Bayesian Tree Parzen Estimator should be decoupled from the data preprocessing logic however, no other major changes except refactoring are required. Thus, there is no reasons to present TPE implementation here.

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

All available in scikit-learn linear regressors implement the same interface and usage routines. Firstly, before usage a regression model should be trained on a data providing separately *features* and *labels*. Afterwards, one may request a prediction for unforeseen features and surrogate will produce a corresponding label according to learned dependencies. This implies that to obtain a prediction of the best parameter combination, one should solve the same optimization problem. However, the crucial advantage of surrogate models is the reduced estimation cost.

To reuse an available in scikit-learn surrogate models we create wrapper as an implementation of unified $Model$ interface, which is required in $Predictor$. The pseudo-code of this wrapper is presented on the Listing 4.7.

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

```
1  class SklearnModelWrapper(Model):
2    def build_model(features, labels, features_description):
3      features_preprocessors, labels_preprocessors = build_preprocessors()
4      transformed_features = features_preprocessors.transform(features)
5      transformed_labels = labels_preprocessors.transform(labels)
6
7      score = cross_validation(model, transformed_features, transformed_labels)
8      if score > threshold:
9        model.fit(transformed_features, transformed_labels)
10       model_is_built = True
11     else:
12       model_is_built = False
13     return model_is_built
14
15   def predict():
16     features = random_sample(features_description)
17     features_transformed = features_preprocessors.transform(features)
18
19     labels_predicted_transfored = model.predict(features_transformed)
20     labels_predicted = labels_preprocessors.inverse_transform(labels_predicted_transfored)
21
22     features_transformed_chosen = select_by_labels(features_transformed, labels_predicted)
23     prediction = features_preprocessors.inverse_transform(features_transformed_chosen)
24
25     return prediction
```

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

During the model creation, we firstly instantiate features and labels preprocessors, and transform the input data (lines 3-5). The underlying process of model building includes also a verification step, which is performed by means of cross-validation: splitting the set of data onto *k* folds, training *k* times model each time excluding one fold for validation and using the others for training (line 7). If the obtained score less than predefined threshold — the model is considered to be not accurate enough therefore, we reject it (line 12), forcing the predictor to use the random parameter values sampling. However, if the model is able to perform well, we train it on an entire dataset and store for further usage (lines 8-10).

Later, during the prediction (if the model was built successfully) we firstly sample features from this level and transform them to fit into created previously model (lines 16-17). Afterwards, we use a regression model to make a prediction for sampled features and transform those predictions back into original labels (lines 19-20). Finally, we select the best feature by means of predicted labels, transform it and return (lines 22-23).

Multi Armed Bandit

Originally, the Multi Armed Bandit (MAB) problem were introduced by [87] and defined as: for 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 one among available choices c_i. The problem derived its name as an analogy to one-hand slot machines in casino and comprises well-known exploration vs exploitation dilemma.

In most of the times, MAB is solved by RL approaches, where before performing each new step, the already available evidences are analyzed. The authors in [4] propose the Upper Confidence Bound algorithm, which proposes an intuitive solution: in iteration k, among available choices select one with a maximal UCB value. The category UCB values is calculated according to the Equation (4.1), where first component Q is a quality of category under evaluation and represents the exploitation portion of

UCB, while the second component estimates the exploration portion with C as 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 Fitness-Rate-Average based MAB (FRAMAB) with two reasons: (1) it is an intuitive and robust approach, (2) according to benchmarks in [35], it outperforms the other. 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 here is the average improvement, obtained by the underlying category.

As for the balancing coefficient C, the authors in [35] were evaluating a range of values $10^(-4)...10^(-1)$ and the dominance for various problem types were different. In addition to the statically used C value, we add a mechanism for proper C estimation by means of standard deviation among improvement. The motivation is as follows: if the deviation is high, there exist an uncertainty in category domination, therefore we encourage the exploration portion of UCB values. Since the implementation straightly repeats provided above algorithm description, we do not provide a pseudo-code.

4.4 Low Level Heuristics

When the HLH is ready to solve the problem, time comes to provide the tools for solving. In this section we present a review of several toolboxes (meta-heuristic frameworks) with an intent to select the best suited one.

Nevertheless, before diving into description of available frameworks we briefly refresh our LLHs requirements, in the interest of which we analyze each framework.

4.4.1 Low Level Heuristics Requirements

4.4.2 Code Base Selection

Available Meta-heuristics with description of their current state With the aim of effort reuse, the code base should be selected for implementation of the designed hyper-heuristic approach.

SOLID

MLRose

OR-tools

pyTSP

LocalSolver

jMetalPy

4.4.3 Scope of work analysis

opened PR

4.5 Conclusion

5 Evaluation

5.1 Evaluation Plan

5.1.1 Optimization Problems Definition

Traveling Salesman Problem

tsplib95 benchmark set which problem I want to solve with hyper-heuristic $http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html$

5.1.2 Hyper-Heuristic Settings

To evaluate the performance of developed system we first need to compare it with the base line. In our case it is the simple meta-heuristic that is solving the problem with static hyper-parameters.

In order to organize the evaluation plan, we distinguish two stages of setup, where different approaches could be applied. At the first stage we select Low Level Heuristic, while at the second one we select hyper-parameters for LLH. The approaches for each step are represented in table 5.1.

Table 5.1 System settings for benchmark

Low Level Heuristics selection	LLH Hyper-parameters selection
1. Random	1. Default
2. Multi Armed Bandit	2. Tuned beforehand
3. Sklearn Bayesian Optimization	3. Random
4. Static selection of SA, GA, ES	4. Tree Parzen Estimator
	5. Sklearn Bayesian Optimization

For instance, mentioned above baseline could be described as $Settings4.1.$ for meta-heuristics with default hyper-parameters and as $Settings4.2.$ for meta-heuristics with tuned beforehand hyper-parameters.

For our benchmark we selected following settings sets:

- *Baseline:* $4.1, 4.2$;
- *Random Hyper-heuristic:* $1.1, 1.2, 1.3, 4.3$;
- *Parameter control:* $4.4, 4.5$;
- *Selection Hyper-Heuristic:* $2.1, 3.1, 2.2, 3.2$;
- *Selection Hyper-Heuristic with Parameter Control:* $2.4, 2.5, 3.4, 3.5$;

The goal of parameter control is to reach the performance of algorithms with the tunned hyper-parameters. The goal of hyper-heuristic is to reach the performance of the best underlying algorithm.

Each of this settings will be discussed in details in following section.

5.1.3 Selected for Evaluation Hyper-Heuristic Settings

Baseline

Hyper-heuristic With Random Switching of Low Level Heuristics

Parameter control

Selection Only Hyper-Heuristic

Selection Hyper-Heuristic with Parameter Control

5.2 Results Discussion

5.2.1 Baseline Evaluation

Meta-Heuristics With Default Hyper-Parameters

Meta-Heuristics With Tuned Hyper-Parameters

Results Description and Explanation

5.2.2 Hyper-Heuristic With Random Switching of Low Level Heuristics

Results Description and Explanation

5.2.3 Parameter Control

Results Description and Explanation

5.2.4 Selection Only Hyper-Heuristic

auto-sklearn paper, p.2 - comparison of GP and TPE BOs.

Results Description and Explanation

5.2.5 Selection Hyper-Heuristic with Parameter Control

Results Description and Explanation

5.3 Conclusion

6 Conclusion

Reviewer: answer research questions

comparison to HITO [45]

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

Hyper-Heuristics and Automatic Machine Learning are the same.

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

7 Future work

Numerical ranges for parameter composition

add more sophisticated models

dependencies / constraints in search space

add new class of problem (jmetalpy easly allows it)

evaluation on different types and classes

adaptive time for tasks

bounding LLH by number of evaluations, not time

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

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

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

influence for warm-start onto this kind of HH (by off-line learning) Although, the influence of meta-learning, applied in Auto-Sklearn system [37] to warm-start the learning mechanism proved to worth the effort spent, as well as it was reported by developers of Selective Hyper-Heuristics with mixed type of learning [101], it is intriguing to check an influence of metal-learning onto Selective Hyper-Heuristics with Parameter Control. Also, $https://ml.informatik.uni-freiburg.de/papers/20-ECAI-DAC.pdf$

Random Forest HLH surrogate model

add other learning metrics Inspiration could be found at: - EAs in [57]: progress stagnation,

Reviewer: consider merging with conclusion, if too short

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

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

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

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

[20] 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.

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

[22] 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.

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

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

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

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

[27] 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.

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

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

[30] 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.

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

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

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

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

[35] 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.

[36] 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.

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

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

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

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

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

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

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

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

[45] 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.

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

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

[48] 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.

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

[50] 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.

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

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

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

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

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

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

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

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

[65] 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.

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

[67] 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.

[68] 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.

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

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

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

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

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

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

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

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

[83] 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.

[84] Dmytro Pukhkaiev and Uwe Aßmann. “Parameter Tuning for Self-optimizing Software at Scale”. In: *arXiv preprint arXiv:1909.03814* (2019).

[85] 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.

[86] 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.

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

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

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

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

[91] 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.

[92] 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.

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

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

[95] 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.

[96] 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.

[97] 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.

[98] 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.

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

[100] 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.

[101] 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.

[102] 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.

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

[104] 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.

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

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

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

Confirmation

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

Dresden, 21st April 2020