TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology

Chair of Software Technology

Systematic Parameter Tuning of Genetic Optimization Approaches

Roman Kosovnenko

Born on: 18th January 1994 in Chernihiv, Ukraine
Course: Distributed Systems Engineering
Matriculation number: 4733290
Matriculation year: 2017

Master Thesis

to achieve the academic degree

Master of Science (M.Sc.)

Supervisors
M.Sc. Dmytro Pukhkaiev
Dipl.-Inf. Johannes Mey
Dr.-Ing. Sebastian Götz

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

Submitted on: 9th March 2020

TaskDescription

Acknowledgment

I would like to take this opportunity to express my immense gratitude to all those who given their invaluable support and assistance.

I wish to express my deepest gratitude to my supervisors M.Sc. Dmytro Pukhkaiev, Dipl.-Inf. Johannes Mey and Dr.-Ing. Sebastian Götz, who convincingly guided and encouraged me to do the right thing even when the road got tough. Without their persistent help, the goal of this thesis would not have been achieved.

I am extremely grateful to my parents for their love, support, and faith in my decisions.

Last but not least, I would like to thank my friends and the people around me. They all kept me going.

Contents

List of Figures

List of Tables

1. Introduction

Optimization problems occur in all aspects of our life. While advancing with science, we improved our abilities in solving problems from naïve random trials in try-fail-try approaches to mathematical analysis or even quantum computing [20].

Evolutionary computing (EC) is a research field comprising state-of-the-art optimization approaches, among which evolutionary algorithms (EAs) are one of the most popular [37]. EAs are applicable to various optimization problems: Traveling Salesman Problem [6], VLSI (Very Large Scale Integration) layout [30], Sequencing Problem [15]. They are especially useful when the user does not possess any knowledge about the form of a fitness landscape. Different variants of EAs are built on top of different combinations of primitive operators and relative parameters, meaning that the parameter values could influence performance of an Evolutionary Algorithm.

In this thesis, we describe the process of parameter analysis and tuning for the Evolutionary Algorithm, applied to solve a problem of software variant selection and hardware resource allocation.

This chapter describes the motivation for parameter tuning and analysis of the genetic optimization approach and marks out research questions to be answered in this thesis. It additionally outlines a solution and contains an overview of the thesis structure.

1.1. Motivation

Despite the high popularity of EAs, their development is not trivial. In [1] the author presents an EA, which aims to solve an optimization problem of software variant selection and hardware resource mapping. A results analysis showed that this approach provides worser performance in terms of final result among the other applied approaches such as integer linear programming and even random search.

Researchers and developers know that good parameter values could improve the results of the evolutionary algorithm [13]. However, a search for the optimized parameter values is another optimization problem that needs to be solved. This problem is one of the persisting challenges of the EC field [32]. The main problem is that the specific EA contains unique design choices presented in form of operators or its parameters.

There exist many recommendations about which values of the parameters of EA are preferable [8, 31]. However, the experimental studies [8, 15, 30] show that these recommendations do not always work.

As a result, parameters need to be tuned. There are two main approaches: parameter tuning and parameter control. Parameter control could set the parameter value at runtime, while parameter tuning finds optimized values at design time.

1.2. Objective

The goal of this thesis is to improve a previously developed genetic optimization approach using parameter tuning. The research objective is to identify and/or introduce parameters that can improve the genetic optimization approach in terms of its performance and scalability and the quality of the obtained solution. We need to answer the following research questions in order to reach the research objective:

- RQ1: Does the parameter tuning improve the results, and what effect does it give?
- RQ2: Were there any bad design choices in the genetic optimization approach? Is there any way to improve it?

To answer these questions, we are using several methods that improve the genetic optimization approach. Moreover, we evaluate it on each stage.

1.3. Solution

To improve the results of the genetic approach we performed the following procedure: Firstly, we identified externally changeable parameters. Secondly, we exposed parameters for external tuning because most parameters of the genetic solver were embedded into its source code. The results at this point showed that parameter tuning improves the result of the genetic solver, but for further improvement, it requires more modifications. We introduced fine-grained crossover and mutation points by adding additional probabilities which resulted in a further improvement of the genetic solver.

Due to the fact that the number of unique elements goes down on each iteration, we developed two methods to solve this issue.

The first approach stops the crossover if the part of unique elements in a population is lower than some threshold. The second approach blocks the possibility of adding a new element to the population if it already contains such an element. This method makes all individuals unique but slows down the solver.

We evaluated our approach on a set of problems of software variant selection and hardware resource allocation which showed that the fine-grained crossover/mutation points and parameter tuning give the best results for the genetic solver. The results are improved not only in terms of solution validity but also scalability.

1.4. Overview

This thesis is organized as follows. In Chapter 2, we provide a background on the evolutionary algorithms, parameter tuning, software selection and hardware resource allocation problem. Chapter 3 describes an iterative approach on improving the genetic approach. The evaluation of the results and its analysis could be found in Chapter 4. Finally, Chapter 5 concludes the thesis and Chapter 6 describes the future work.

2. Background

This thesis is based on several kinds of researches: auto-tuning, evolutionary algorithms, and software optimization. In this chapter, we summarize the important aspects and details of approaches and optimization techniques used within the thesis.

2.1. Evolutionary Algorithms

Evolutionary algorithms (EA) are a subset of evolutionary computation and belong to the set of modern heuristics-based search methods [37]. This domain contains different variants, such as:

Genetic algorithm (GA) is the most widely known type of EA [13]. The research [9] presents the description of the simple GA that represents a solution as a binary string. GA uses proportional selection, and a low probability of mutation. It has a fixed flow. The entire population changes in each generation to a new population. Currently, more preferable is to use the rank-based selection, one-point crossover changed to the alternatives. And most important, that developers use not a binary but problem dependent representation that increases an understanding of the problem that GA is solving.

Evolution strategy (ES) works with vector representation of a solution. New offspring is generated by adding random values to the elements in the vector. It uses self-adaptation for own parameters. ES self-adapt the mutation step sizes.

Evolutionary programming (EP) was developed with the aim to get artificial intelligence [13]. It has the same Solution representation as ES, but do not use the recombination. The selector algorithm selects parents from the union of the population and new offspring.

Genetic programming (GP) use as a representation of solutions tree-shaped structures. Recombination works by sub-tree exchanging. A particular difference is in the workflow. A GP performs either a crossover or a mutation, and a GA performs two operations in sequence.

The main difference between them is a solution representation called the **Genotype** or the **chromosome**. Each genotype consists of **genes** which represents the minimal

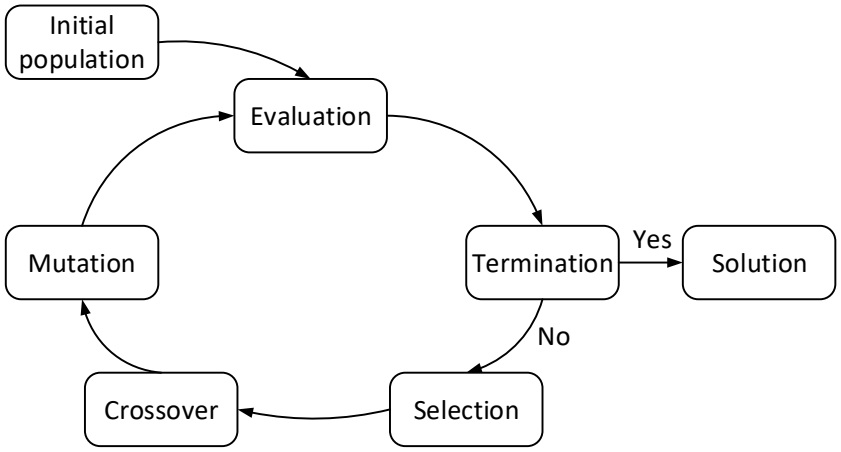


Figure 2.1. The main loop of the evolutionary algorithm

element of the solution. A genotype in a form that could be evaluated is **Phenotype**. A set of genotypes (chromosomes) is a population.

Any EA is based on different components and operators that construct it flow. Figure 2.1 shows the main principle of EA. It looks like a loop with iteratively repeated steps. The first step is the creation of an initial population. It is a set of randomly created genotypes; The second step is Evaluation. Calculate the fitness or objective function of the current population. If the solution is founded and all requirements for the termination are fulfilled, then we get the final solution. Otherwise, select the best candidates to create a new generation. In step 4, using Recombination and Mutation on selected candidates, GA creates a new generation of the population. And after that, evaluate it.

2.1.1. Selector

The Selector is one of the most important components of the EA. It selects ***mu*** chromosomes from the population, that are used to create new ***lambda*** chromosomes. This process is called *Selection algorithm*. There are many Selection algorithms.

NSGA-II (Non-dominated sorting based genetic algorithm) [11] is a multi-objective selection algorithm. It is searching for non-dominated solutions. Figure 2.2 demonstrates how it works. There is a population R_t that consist of two sets. First set is a parent population. It is marked as P_t. The second set is new offspring and denoted as Q_t. Combined population R_t is sorted using non-dominated sorting. After that, it generates a Pareto front. After sorting, selector selects the new population P_{t+1} of size represented as a parameter by binary tournament selection and use it for the next round of reproduction.

SPEA2 (Strength Pareto Evolutionary Algorithm) [42] a multi-objective selection algorithm, that used to find the Pareto optimal solution for multi-objective problems[41]. Figure 2.3 demonstrate how it works. There are two sets. First one is a current population that denoted as R_t. The second is an archive set. It is marked as A_t. SPEA2 takes the union of all solutions in R_t and in A_t. The union set is marked as U_t. After that it compares the size of U_t and archive size. If U_t is grater than archive size then SPEA2 reduce the union population. If U_t is less size than SPEA2 extend the population from

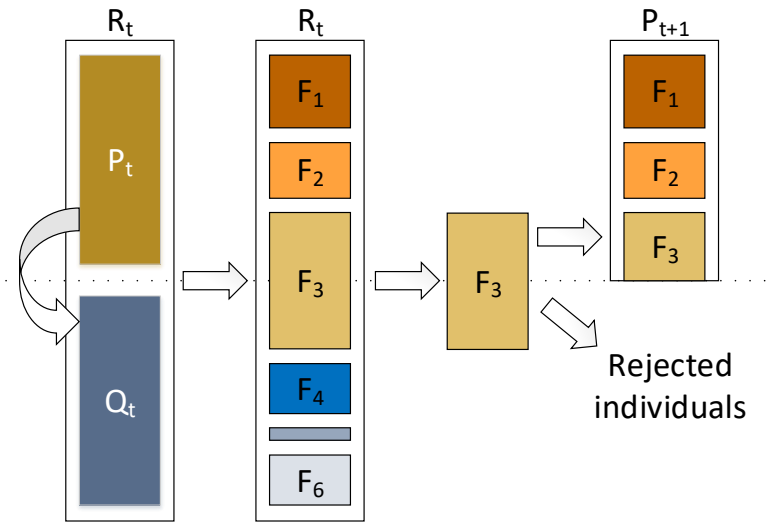


Figure 2.2. NSGA-II selection algorithm

the dominated solutions. After that, it selects best non-dominated solutions that used in next iteration.

NSGA3 [10] extends NSGA-II to using reference points to handle multi-objective problems, and its usage on one- or two-objective problems is discouraged.

SPEA3 presented in [29]. It is a generalization of SPEA2 that consists of the exchange of the selection procedure that aims to determine the non-dominated solutions with a high spread and balanced distribution. As described in researches, it works well with two- and three-objective problems.

In this thesis, we focus on a Selection algorithm only as a parameter of a genetic algorithm and use NSGA-II and SPEA2 due to software constraints of the used framework.

2.1.2. Crossover

Crossover is an operator of the EA that allows the recombination of two genotypes by swapping some genes between them. In general, the crossover has several parameters such as:

- Crossover rate is a parameter that describes the probability of two chromosomes to exchange their genes;
- Crossover point is a point in which the exchange could be done.

The principle of the crossover is next. Firstly, select the crossover point. For example, a chromosome could be described as a vector of bits. Then the crossover point is the start index of bits, which replaced by another chromosome. Secondly, it swaps genes between chromosomes.

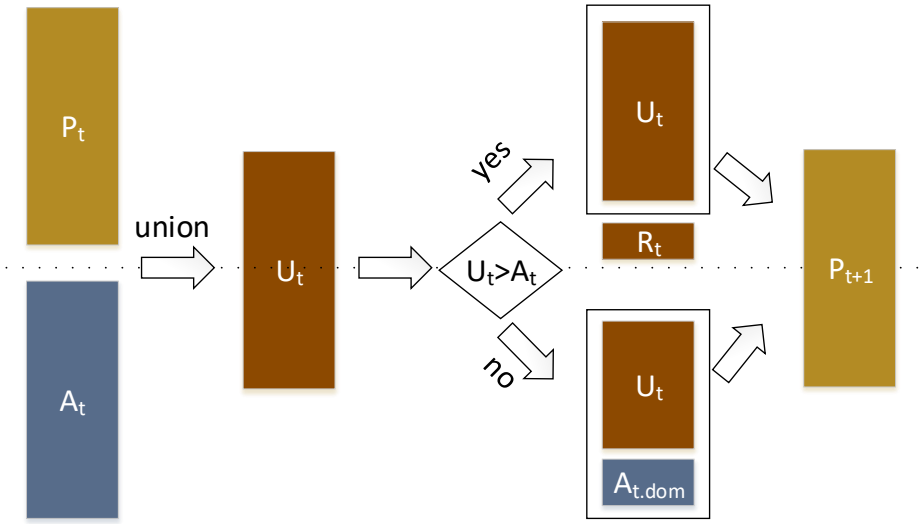


Figure 2.3. SPEA2 selection algorithm

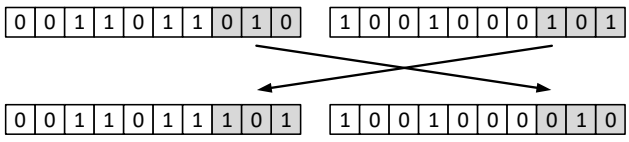


Figure 2.4. Example of the Crossover between chromosomes that described as a vector of bits

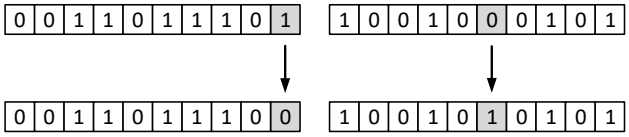


Figure 2.5. Example of the Mutation in a chromosome that described as a vector of bits

2.1.3. Mutation

The mutation is an operator of the EA that changes a single gene in a chromosome. Similarly to the crossover, a mutation has parameter `Mutation rate`. It describes the probability of mutation.

To perform a mutation on chromosome it is needed to:

1. Randomly select gene which mutates;
2. Change selected gene to another.

Figure 2.5 shows a simple mutation on the chromosome that is described as a vector of bits.

2.2. Parameter Tuning and Parameter Control of the Evolutionary Algorithms

There are many parameters used by EA to solve the optimization problem. And we believe that "good" parameter values could improve EA. There exist different approaches. The first approach of parameter tuning is manual setting. It adheres to select the values of parameters by some conventions, ad-hoc choices, or by experimental comparison on a limited scale. Such a tuning strategy is based on assumptions about user-preferences and the influence of parameter values [12, 13]. For example, the developer tries to tune 4 parameters and 10 values for each of them. The drawbacks of this approach are:

- Parameters could depend on each other. As a result, they can not be optimized one by one;
- Trying all possible combinations is difficult and requires a huge amount of time. To test 4 parameters with 10 possible values, need more than 1 million runs. If we run it at least 3 times to achieve some level of accuracy, then we need to run EA more than 3 million times. If a single run of the EA needs 1 minute, then parameter tuning will be finished after 5 years;
- If parameters are numerical or continuous. The optimal value may not be in the tested values.

We can conclude that parameter tuning without automation and without the understanding of the process is not working. Tuning an algorithm requires a lot of computer power, while some people argue that this is a waste of time [33].

As a result, there are 2 approaches to choosing parameters: Parameter tuning and parameter control [13, 34]. *Parameter tuning* finds optimized values of parameters before the run. So all parameters are fixed and do not change during the run. However, *parameter control* sets the values of parameters while EA is running. So the parameter is initially given, but during the run, the parameter value is changing.

2.2.1. Parameter Control

As mentioned before, parameter control could adjust EA parameters on-the-fly. A run of the EA is a dynamic process, and using static parameters could be optimal only for some state, problem, or stage of EA. The simplest example of such technique is Rechenberg's 1/5 success rule [28]. It describes how to change mutation parameters using information about the number of successful mutations. Commonly such features based on the developed feedback mechanism that contains information about the search process.

Parameter control consist of 3 categories:

- Deterministic parameter control in which a parameter value adjusted by the deterministic rule without any feedback;
- Adaptive parameter control uses some feedback from the EA as input for parameter changes. The value of the parameter may depend on the quality of the solution. Parameter changing is not a part of EA that solves the optimization problem;
- Self-adaptive parameter control in which parameters are part of the chromosome. As a result, better chromosomes lead to better parameter values.

In this thesis, we focus on parameter tuning of EA, but we understand that parameter control is a very important approach that could significantly improve the results.

2.2.2. Parameter Tuning

All parameter tuning methods are divided into 2 categories: model-free and model-based [22]. The first category trying to find "good" parameter values that give the best result of EA to be tuned. Examples of such tuners are F-Race [5] and ParamILS [21].

F-Race is a racing method that is adapted to fine-tune stochastic search methods [23]. To compare sets of parameter values, it uses Friedman two-way analysis [36] of variance by ranks. This method does not require any additional hypothesis about the search space. It works until complete all search space, or when a defined number of configuration is analyzed. The F-Race has the following parameters:

- initial number of runs without calibrations;
- the confidence level;
- the max budget;
- the ranges of all parameters.

ParamILS (Parameter Iterated Local Search) is performing parameter tuning as an iterated local search algorithm. It starts the optimization process with default parameter values and goes further. It moves from the default set to the neighborhood. On each iteration, it takes a few sets of parameter values and performs a local search to find out the best configuration. The ParamILS has the following parameters:

- the number of configurations for the first iteration;
- the number of tested configuration per iteration;
- a restart probability;
- the max budget.

The second type is a more complex solution. Such tuners build a model that predicts the results of EA for given parameter values. As a result, different models or heuristics

are used to reduce the number of tests. The model builds based on data about parameters and results that parameters give. The examples of model-based tuners are: Sequential Parameter Optimization (SPO) [4], Bonesa [3, 34] REVAC [24] and BRISE [26, 27].

REVAC (Relevance Estimation and Value Calibration of EA. Works as an estimation of distribution algorithm [23, 25]. It represents each parameter as a set of values. The REVAC starts for each parameter the search process that uses the uniform distribution of values. It also performs a transformation of parameter sets to reduce the range of values and perform a predefined number of runs. This tuner appropriate for ordered parameters. The REVAC has parameters:

- the number of elements in a set;
- step sizes of set modifications;
- the max number of iterations.

BRISE - the software product line (SPL) for parameter tuning [27]. The work concept consists of a combination of local search and model prediction. It starts with the selection algorithm, which selects a new set of parameters. After that, the model rebuilds on each iteration and predicts the best parameter values. The BRISE is a more generic approach and used not only for EA. It has many parameters that describe its components.

The experimental comparison in [33] shows that no matter what tuner is used, it gives a better EA than relying on intuition and the usual parameter setting conventions.

In conclusion, many tuners help to get optimized parameter values for EA. They have different methodologies to tune parameters such as models or selectors to select parameter values from several places of the search space. This overview shows that we could use all types of tuners.

Since a genetic solver requires a big amount of time to solve the task, we decide to use one of the model-based tuners, and BRISE is more generic than others. It is preferable because BRISE uses its core entities such as Configuration and Experiment. It does not know how the target system works and analyze hyperparameters and results of runs. More details about BRISE are in the next section.

2.3. BRISE

BRISE is a software product line (SPL) for parameter tuning [27]. This SPL has two conceptual parts:

- Static part describes the main flow of parameter tuning, task management, and reporting;
- Configurable parts are configured for each experiment.

Figure 2.6 shows the high-level architecture of BRISE. Configurable part consists of:

Experiment description describes the Experiment, parameters to be tuned, the objective of the experiment, and expected values to detect outliers.

Selector is a Selection Algorithm that is used to get the combination of parameters from the Search Space of all possible parameter combinations. BRISE has two selection algorithms out of the box: Sobol sampling [35] and the uniform distribution.

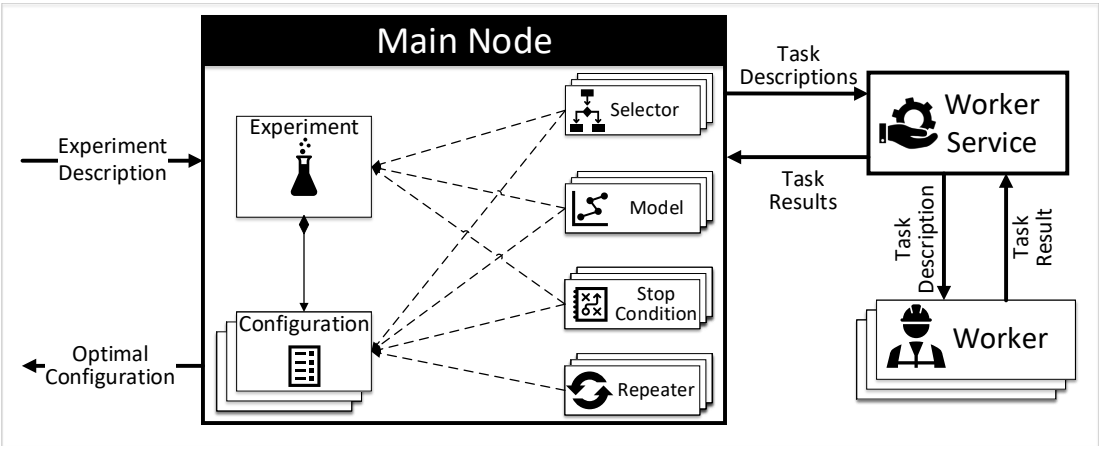


Figure 2.6. The high-level architecture of BRISE

Model predicts parameter values. New Model builds after each iteration of measurement of configuration, and if the model predicts a valid result, this configuration is measured in the next iteration. BRISE has several models out of the box.

Stop condition is a component that analyzes the results of the experiment, validates the result, and makes a decision to stop the experiment or continue. There are many stop conditions that work as a combination of criteria.

Repeater is a component of BRISE that decides about the number of measurements for each configuration to get the desired accuracy of the results.

Worker contains the process that BRISE needs to tune. Run this process with different configurations and returns the metric of this configuration to use it in the next model build. All components from the configurable part could be extended by users to achieve their goals.

BRISE has two modes: *Search space exploration* and *Hyperparameter tuning*. The first mode is used to get information about the search space of hyperparameters. In this mode, there are no predictions because BRISE uses the selector to cover the search space. As a result of using BRISE without predictions, the model component is disabled.

The second mode is used to find the optimized values for hyperparameters. All components of the main node are used in this mode.

The main principle of how BRISE works in both modes is the same, but Search space exploration mode has one exception.

The user prepares a target system. Workers call it with specified hyperparameter values and return the results to the BRISE. After that user describes the experiment description. It is a JSON file that contains all information about hyperparameters, their ranges and default values, result structure, expected ranges for result values, and the max time needed to run one task (see Appendix A).

The user input of BRISE is an experiment description that in BRISE transforms into the experiment object after that BRISE starts working.

It starts with the measurement of the default configuration. When the default configuration has been measured, the selector gives a set of new configurations to be measured. After measurements, workers return the results to the main node. The

repeater checks the result and decides to measure this configuration one more time, or results have needed accuracy.

On the next step model component tries to build the model and validate it. If a model is valid, then it predicts a new configuration. If the model is not valid, BRISE gets a new configuration from the selector. BRISE sends it to worker service to measure and get the result. If SPL is working in Search space exploration mode, the model build step is skipped, and the new configuration is always received from the selector.

These measurements are being performed until stop condition criteria in stop condition components are met.

We already discussed how to tune parameters and decided what tuner to use. Now let us consider the optimization problem.

2.4. MQuAT Problem

Multi-Quality Auto-Tuning (MQuAT) is an approach to self-adaptive software, which provides design and operation principles for software systems that automatically provide the best possible utility to the user while producing the least possible cost.

It is based on the design-time part, which represents a new development method for self-optimizing systems and run-time parts, which concerns operation principles, namely, novel techniques to run-time self-optimization [16].

MQuAT allows software developers to build and run software systems, which are automatically adjusted to provide the optimal user utility for the minimal cost. Such systems are comprised of multiple variants differing in their non-functional behavior and can analyze the reasons for the trade-offs between the respective non-functional properties (NFPs). A key characteristic of MQuAT is the use of Quality of Service contracts to cover the inter-dependencies between NFPs, the distinction between hard- and software, and the use of behavioral models to simulate complex non-functional behavior like *energy consumption*. In this thesis, MQuAT is used as a software environment that provides an optimization problem.

MQuAT problem is described in [18], and it is a combination two problems:

- Resource allocation in which the mapping of software component implementation to hardware resource leads to the least cost;
- Variant selection, which provides the best utility by selecting better software implementations.

The problem description uses the JastAdd framework [14] based on specified grammar [17]. As a result, it gives a possibility of using reference attribute grammars (RAGs) [19] that adds computations in model nodes.

To solve this problem a new generic meta-model is described. Both problems are interrelated by user requests specifying minimum requirements on the provided non-functional properties (i.e., minimum utility) while searching for a selection and mapping both to maximize utility and minimize costs. Correctness denotes that only solutions that *do not violate* the user's minimum requirements are considered valid.

The MQuAT problem is selecting variants of software components and mapping them based on user requests to suitable hardware resources.

The MQuAT problem is also described as a meta-model (see Figure 2.9) which consists of:

Hardware meta-model, which consists of hierarchically structured resource types and resources as instances of these types. So the hardware model composes static

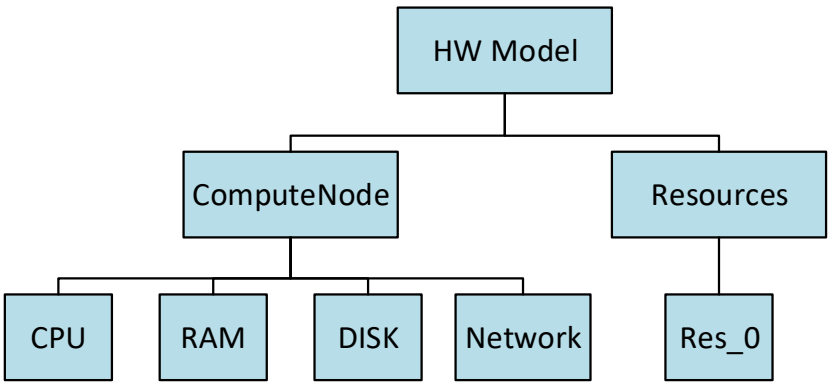


Figure 2.7. Hardware meta-model

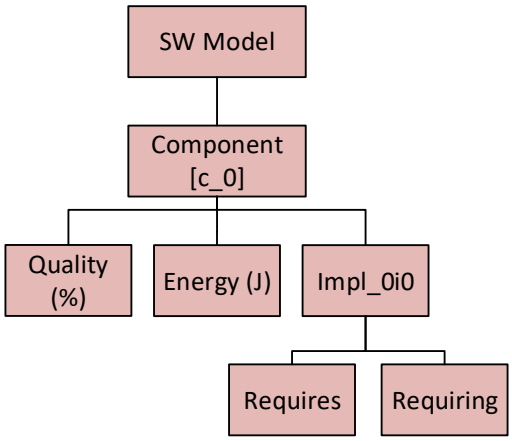


Figure 2.8. Software meta-model

resource types and run-time resource instances knowledge. Certain types of resources can run the software, i.e., they are valid targets for software implementation mapping. The container attribute is used to mark such types. Figure 2.7 depicted Hardware meta-model.

In addition, a set of properties further characterizes resource types. Resources specify then concrete values for these properties. As an example, the resource type RAM could be defined with a property amount of memory and marked as a container.

Software metamodel showed on Figure 2.8. Its main element, called component, provides some functionality required by the user. Each component could have different implementations that provide this functionality, requiring additional components or resources to complete their work.

Objective specifies how to calculate a solution's objective value, i.e., for which value(s) the problem is optimizing For example, minimize energy consumption.

Request represents a user's specified functionality that executes with its parameters and requirements. The request contains the functional requirements by referencing a target software component and limitations on non-functional requirements (e.g.,

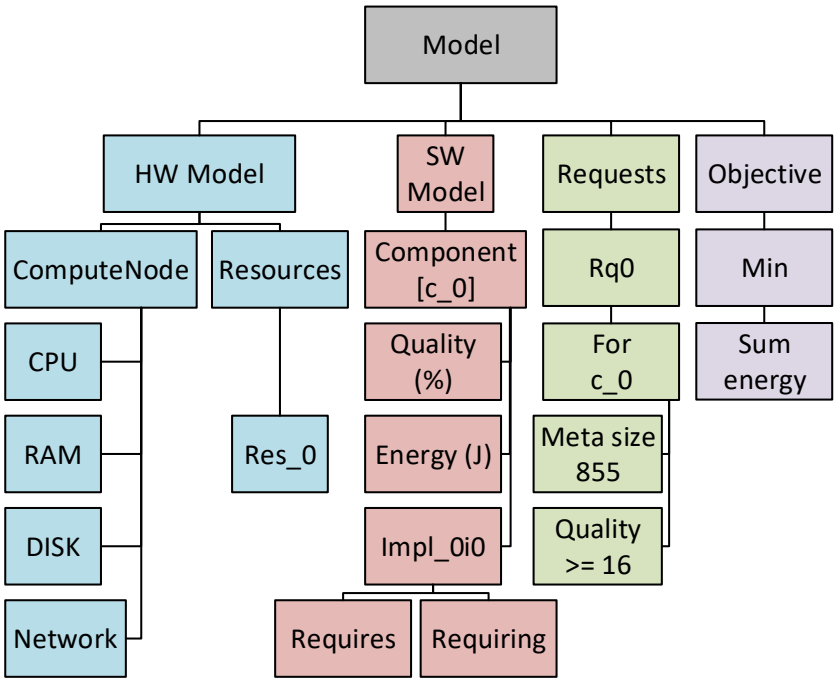


Figure 2.9. MQuAT problem model

quality).

Due to the complexity of the MQuAT problem, some constraints are grouped in Architectural, Request, and Negotiation constraints groups:

- **Architectural constraints** ensure that each request is fulfilled, selecting exactly one implementation per component and deploying no more than one implementation on one resource;
- **Request constraints** ensure components are selected for each request to provide the requested non-functional properties;
- **Negotiation constraints** ensure non-functional requirements are met depending on the implementation.

There are additional constraints due to synthetic problem generation:

- Structures for the software and hardware components are fixed to ensure comparability;
- `computeNode` that represents a regular computer hardware consist of one or more CPUs, RAM memory, disk, and a networking interface;
- Software model has a simple tree structure;
- Each software component could have 2 sub-components, or do not depend on sub-components. It means that each node of the tree structure has 2 sub-nodes, or it is a leaf of the tree.

Each MQuAT Problem is characterized by four parameters:

- **Software variants** is a number of implementations for each software component;
- **Number of requests** is a number of requests to run software components, that are described by the user;

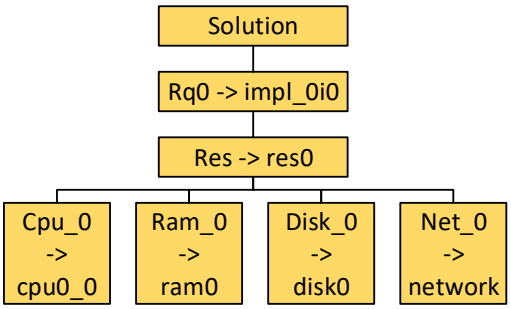


Figure 2.10. MQuAT solution model

- **Component tree depth** is a depth of the software tree that describes the dependency of software component that user requested;
- **Resources ratio** describes the number of available hardware resources. The number of HW resources calculated as the multiplication of the number of nodes of the component tree by the number of requests and multiplied by the resource ratio.

The MQuAT problem is an optimization problem. Many approaches to solving the problem exist. The MQuAT provides needed interfaces for these approaches. It is called Solvers. The Solution of MQuaT problem is computed by the MQuAT Solver. There are many Solvers:

- **Simple Solver**, which goes step by step from one solution candidate to another, enumerating solution candidates;
- **ILP Solver** [18], which generates Integer Linear Programming (ILP) Problem from the MQuAT Problem and solves it;
- **Random Solver**, which tries random solution candidates;
- **Simulated Annealing (SA) Solver** [27] is based on the Simulated Annealing metaheuristic;
- **Genetic Solver** [1], which uses an Evolution Algorithm to solve the problem.

In this thesis, we focus on the Genetic Solver in detail in the next sections. The solution of the MQuAT Problem represents a tree structure. An example of the Solution shown in Figure 2.10. It contains a list of assignments. Each assignment selects one implementation of the required component and maps it on the resources [18].

The Solution is valid if for each user request following requirements fulfilled:

1. an implementation is deployed for the target component;
2. an implementation is deployed for each required component;
3. all necessary (non-functional) property clauses (including request constraints) are met;
4. at most one implementation for each resource is deployed.

The Solution is optimal if the Solution is valid, and no other solution has a better objective value [18].

2.5. Genetic Solver

The Genetic Solver uses EA to solve the MQuAT Problem. The solver developed by Jamal Ahmad in [1] and further improved by Johannes Mey.

This Solver is based on Opt4J framework[1]. It is an open-source framework that gives the opportunity to implement an EA for custom optimization problem.

Several classes created to solve the MQuAT Problem using EA. There are:

1. Genotype is a custom developed genotype that represents tree shaped structure of the solution;
2. Crossover operator describes the algorithm of the crossover for custom genotype;
3. Mutation operator describes the algorithm of the mutation for custom genotype;
4. Creator is needed to create a random genotype for the initial population; In the genetic solver Creator creates the genotype by generating a random solution model and transform it into a Tree Shape Genotype structure;
5. Decoder is needed to perform decoding the tree shape genotype into phenotype. The phenotype, in this case, is a Solution Model of MQuAT;
6. Evaluator calculates the objective functions of the Solution. In the case of genetic solver, the Evaluator calculates two objectives:
   - Validity errors is a number of violated contracts;
   - Energy value is energy consumption.

2.5.1. Tree Shape Genotype

Because of the problem model of MQuAT, which requires mapping of implementations to resources, in genetic solver was created Tree Shape Genotype [1]. The example of this genotype shown on Figure 2.11

The first node in this genotype contains the input request. The second node represents the mapping of the user component. In this node, Impl A-1 is selected rather than Impl A-0 of the user component. Figure 2.11 shows that implementation A-1 is mapped to Hardware-Resources1. Moreover, Impl A-1 also requires software components B and C that have only one Impl B-0 and Impl C-0 implementation and are mapped to Hardware-Resource5 and Hardware-Resource4, respectively.

2.5.2. Crossover Operator

This operator performs a crossover between two Tree Shaped Genotypes. The process starts on the root of both trees. Crossover operator compares if the implementation and mapped resources of both nodes are the same. If not the same, then with the probability `CrossoverProbability`, it swaps the implementations, resources, and lower substructure of the tree. It swaps the sub-tree in order to maintain consistency. If comparison says that nodes are identical, then it performs the crossover process for each child node. Such a recursive algorithm gives a possibility to perform the crossover on a few points of the Tree Shape Genotype. Figure 2.12 shows these points. Crossover points have a gray color.

[1] http://opt4j.sourceforge.net

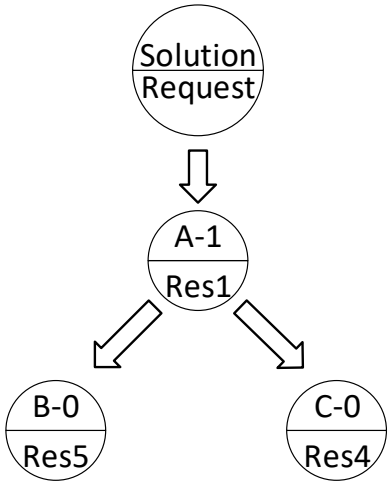


Figure 2.11. Example of the Tree Shape Genotype

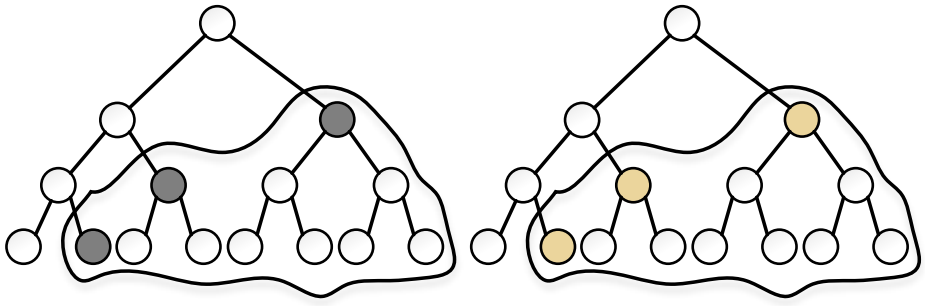


Figure 2.12. Crossover Points

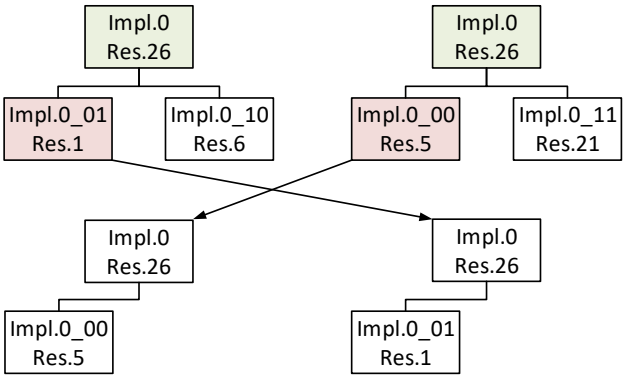


Figure 2.13. Example of crossover between two Tree Shape Genotypes

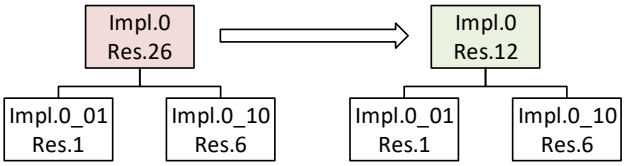


Figure 2.14. Example of mutation between two Tree Shape Genotypes

Figure 2.13 depicts the crossover between two genotypes. Each genotype has 3 nodes. The Crossover starts on the root nodes. These nodes have the same implementation and resources. As a result, crossover recursively goes down to children. The first child of both genotypes has different implementation and resource, and crossover swap them with the probability `CrossoverProbability`. After that crossover operator performs a comparison for the second child.

2.5.3. Mutation Operator

The mutation occurs for one random request. With the probability `MutationRate`, it mutates the root node. If a mutation operator mutates the top node of the tree, there is a probability of the resource mutation. Or it mutates the implementation of the top node and changes the lower structure for consistency. If the mutation occurs not in the root of the genotype, then the process recursively goes down to children. Figure 2.14 depicts the mutation process in which mutation performs on the resource in the root node.

2.5.4. Evaluator

As mentioned in Section 2.1, the evaluator calculates the quality of the solution. In the Genetic Solver, the quality function calculates the energy consumption. It uses the MQuAT energy consumption calculation and adds penalties to it depends on contract violations and errors. These penalties are characterized by two parameters:

- `ValidityWeight` is a coefficient that shows how each contract violation decreases the quality of the solution;
- `SoftwareValidityWeight` is a the coefficient that shows how each error in the software tree decreases the quality of the solution.

2.6. Parameter Tuning Strategies for Evolutionary Algorithms

As discussed above, one of the crucial tasks of EA is parameter tuning. A lot of variants of parameters relating to the generation, the population and operators [31]. There exist different commonly used rules and researches of setting the parameter values of EA.

The examples of such rules are:

- mutation rate should be low;
- use uniform crossover;
- high crossover rate.

It is not surprising that users of evolutionary algorithms are asking such questions:

- Are there optimal settings for the parameters of an EA in general [8]?
- Are there optimal settings for the parameters of an EA for a particular class of fitness landscapes?
- How do changes in a parameter affect the performance of an EA?

As a result of these questions, [8] describes generic recommendations for EA parameter values. The value of population size highly depends on the landscape of the optimization problem. If EA does not use parameter control, then the value of population size is based on existing studies or via manual tuning over multiple runs. It is recommended to use the uniform crossover.

In [30] researchers are looking for the best parameters to solve the problem of VLSI (Very Large Scale Integration) layout. To find optimized values of crossover rate and mutation rate, they use a meta-GA [7]. As a result, [30] contributes that the crossover rate should be in the range 0.2—0.4 and the mutation rate in the range 0.005—0.05.

Another work by [40] describes meta-GA that optimizes parameters for the nonlinear constrained mixed discrete-integer optimization problems. The results show that the solution does not depend on the value of the crossover rate. The recommended value for the mutation rate is in a range of 0.01-0.2. It is higher than in the previously discussed research.

Parameter analysis for the Sequencing Problems in [15] shows, that a mutation rate of 15% produces the optimal results independently of the population size

The experimental analysis of several problems in [31] describes own recommendations for parameter values:

- Population size need not be maximal;
- Generation count need not be maximal;
- At most, population size and generation count should not both be very low;
- Crossover rate can be high, low, or intermediate;
- Mutation rate can be high, low, or intermediate.

These recommendations partially contradict other approaches and rules.

Some researches [2, 33] conclude that even "commonly used" parameter values give a good result, so parameter tuning does not need.

From the preceding, we can conclude that generally accepted norms of parameter values can give a relatively good result, but in order to get better results, it is necessary to analyze and tune the parameters for a specific problem or a specific class of problems.

The MQuAT problem is not wide analyzed in terms of EA. There is no similar research on parameter tuning of EA that solves such problems. Let us proceed to parameter tuning of the Genetic solver.

3. Implementation

In this chapter we describe several approaches to improve the Genetic Solver by parameter tuning. We concentrate on solving the problem with **5 minutes** timeout and the following properties:

- **Software variants: 10**;
- **Number of requests: 15**;
- **Component tree depth: 2**;
- **Resources ratio: 5**;

on the Intel Core i7-8700 CPU machine with 64Gb of memory using Fedora Server 29 and OpenJDK 1.8.0 201-b09 for **5 repetitions** for statistical significance.

Our goal is to achieve valid results in terms of contract violations and then get an optimal or near-optimal solution in terms of energy consumption.

We mark base[1] version of the genetic solver as **B** (Base).

Figure 3.1 shown the box-plot of the number of contract violations in the base (**B**) version of the Genetic Solver. This plot shows that the base (**B**) version solves this task with the number of contract violations in a range from 13 to 22. As a result, there are no valid solutions for the current problem.

3.1. First Parameter Tuning

The first step to improve the genetic solver is parameter tuning. To get a set of optimized parameter values, we needed to perform a few essential steps:

1. Analyze the genetic solver to find all parameters that are changeable externally. These parameters are hyperparameters of BRISE;
2. Adapt BRISE to work with the genetic solver;
3. Prepare Experiment description for BRISE;
4. Start BRISE to get the near-optimal configuration of the Genetic Solver.

Step 1 involves the analysis of the solver for the presence of externally changeable parameters. After diving into the code of the genetic solver, the list of parameters is

[1] As a base version, we took version at the master branch from 24.09.2019, commit: 57845c126c30a1ea59cb35eb16af0bd37930dda9

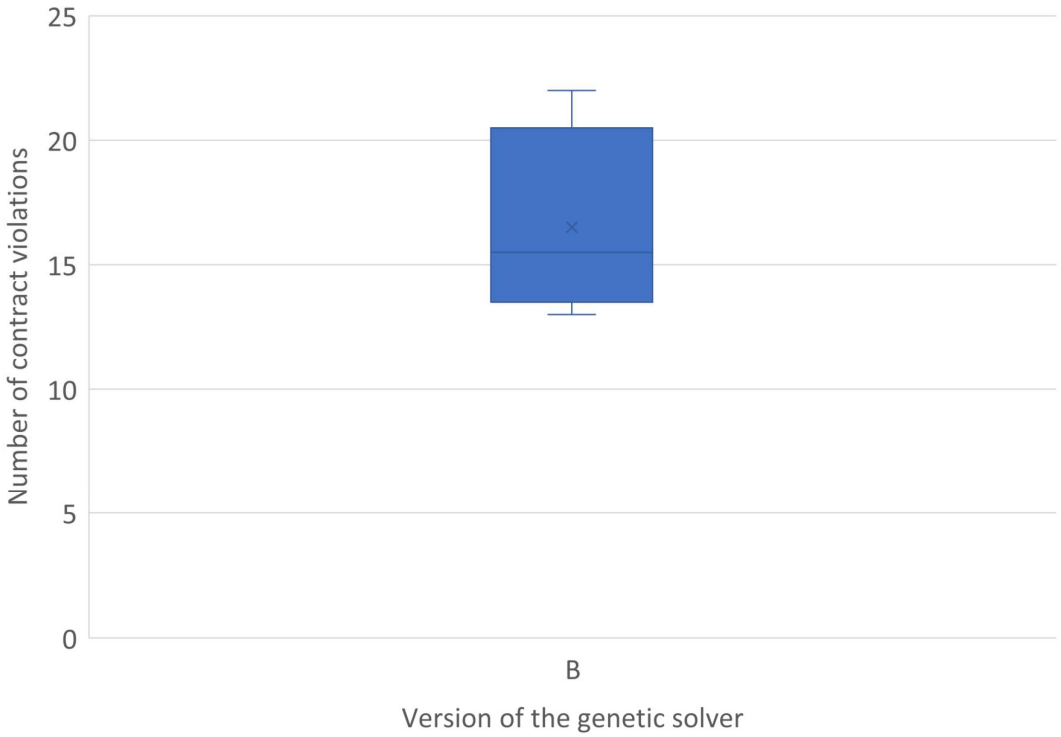


Figure 3.1. Box-plot of contract violations for the base version of genetic solver

prepared. Each parameter has a name, short name that consists of capital R and the order number of parameters, the default value, and a range of values for continuous parameters or a list of values for categorical parameters. The possible value we show with square brackets near the short name. There are:

`SelectorType` (R1) [NSGA2, SPEA2] is type of the selector algorithm, mentioned in Section 2.1.1. Default value is *NSGAII*;

`Generations` (R2) [Integer] — a number of generations, which the Genetic solver is do. It is working as a termination condition. In this thesis, we use only timeout termination. Let us set this parameter in a way, that it does not influence the termination (since the timeout-based termination is used instead);

`PopulationSize` (R3) [100-5000] [Integer] describes the number of individuals in a population. Default value is 500.

Adaptation of the genetic solver to work in BRISE (Step 2) consists of preparing the executable file of the genetic solver and implementing a method for the worker, which encapsulates execution of the Genetic Solver with specified parameters and returns the results with a number of contract violations and energy consumption.

Step 3 means that a user describes hyperparameters that need to be optimized and BRISE components configuration such as selector, repeater, model, and stop condition, list of hyperparameters described by a name, a range of values, or a set of values, the default value.

For parameter tuning of the genetic solver, we use Sobol sampling as a selector algorithm to get the new configuration of hyperparameters, student repeater with a

Table 3.1. Parameters of B and B-T versions of the Genetic Solver

Genetic Solver	R1	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
B	NSGA2	500	0.1	0.95	0.1	0.45	0.5	0.8	0.5	0.5	50	100
B-T	NSGA2	1533	0.1	0.95	0.1	0.45	0.5	0.8	0.5	0.5	50	100

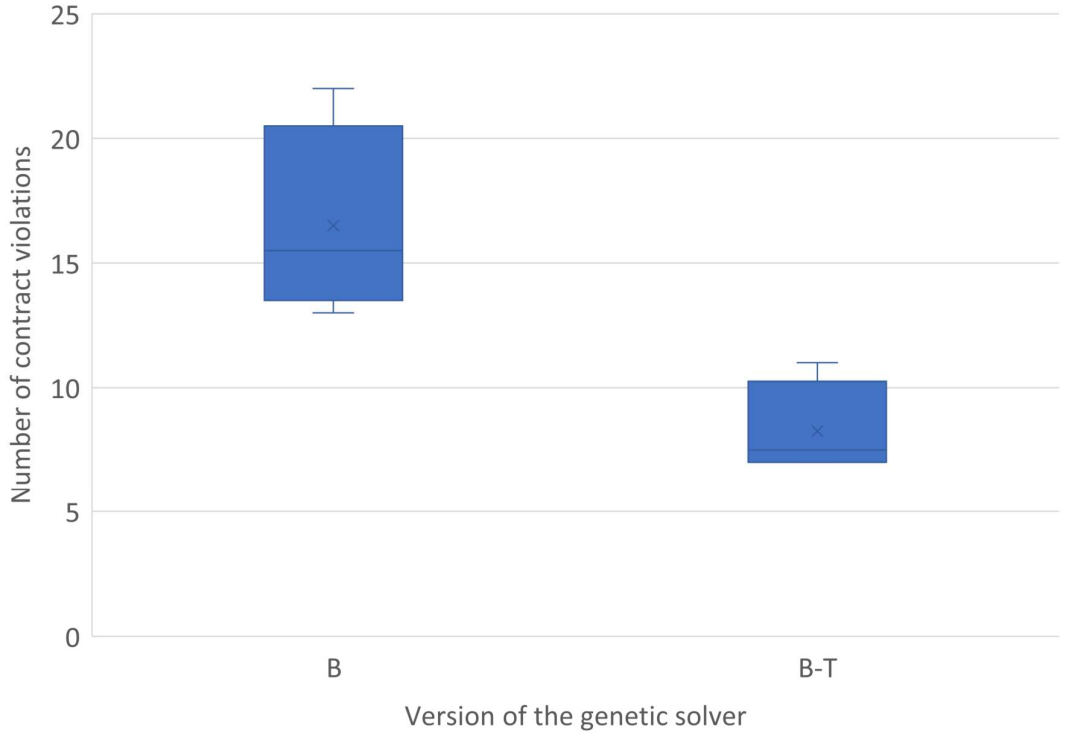


Figure 3.2. Box-plot with a number of contract violations for the base version of the Genetic Solver with tuned parameters

minimal number of measurements — 3 and max number - 6. BRISE uses Bayesian Optimization (BO) as a prediction model. To stop BRISE, we use a combination of time-based stop condition with a timeout of 12 hours and guaranteed stop condition, which ensures achieving a better result in comparison with the default set of parameters. The experiment description file is presented in Appendix A.

We use parameter values of the base (**B**) version of the Genetic Solver as a default configuration for BRISE. Such decision makes it possible to use a guarantee stop condition. The default parameter values are presented in Table 3.1. Also, there are a few parameters in gray-colored filling that have a hard-coded value, but we will discuss them in the next section.

We mark current version[2] of the genetic solver as **B-T** (Base-Tuned).

The result of hyperparameter optimization is presented in the Table 3.1. It shows, BRISE gives a set of parameters with a bigger number of population size (R3).

The results of solving the earlier described problem as a box-plot depicts in Figure 3.2. In the Figure 3.2 is also showed a comparison between the base (**B**) and its tuned (**B-T**) version of the genetic solver. The **B-T** version of the genetic solver gives a better result compared with the **B** version. The minimal number of contract violations in the **B-T** version is almost 2 times fewer than in the **B** version.

Received results answer the **RQ1**. Parameter tuning improves results, and it has a

[2]commit: e103f52b3333900d61c6218a1f2ca811bca10289

positive effect on the genetic solver, but results are still not valid.

3.2. Expose Parameters for External Tuning

In the previous section, we showed that not all parameters of the genetic solver exposed for tuning. To make them changeable externally, we use `GoogleGuiceDependencyInjectior` tool to change the values of these parameters during the solver call. Let us discuss these parameters:

`lambda` (R4) [0.0-1.0] [Float] was mentioned in Section 2.1.1, it is the number of new offspring per generation. We use this parameter as a part of the population size. The default value is 0.1 (from the base (B) version of the genetic solver).

`CrossoverRate` (R5) [0.0-1.0] [Float] was mentioned in Section 2.1.2, it describes the probability of two individuals to gene exchange. The default value is 0.95 (from the (B) version).

`mu` (R6) [0.0-1.0] [Float] was mentioned in Section 2.1.1, it describes the number of parents, selected by the selector to create new individuals. We use this parameter as a part of the population size. The default value is 0.1 (from the (B) version).

`MutationRate` (R7) [0.0-1.0] [Float] was mentioned in Section 2.1.3, it describes the probability of the individual to mutate. BRISE will tune this parameter with default value 0.45 from the base (B) version of the genetic solver.

`ResourceMutationProbability` (R8) [0.0-1.0] [Float] is a likelihood that the assigned resource mutates during the mutation. BRISE tune this parameter with default value 0.5 from the B version of the genetic solver.

`CrossoverProbability` (R9) [0.0-1.0] [Float] — it describes the probability of crossover that checks in the crossover operator. The default value of this parameter is 0.8. This parameter is removed in later versions of the genetic solver because it duplicates the functionality of `CrossoverRate` (R5). This fact answers RQ2. It is one bad design choice that we want to find, according to RQ2.

Within the genetic solver evaluator, two factors penalize the quality of the solution in the case of a contract violations or errors in dependencies in the software tree. These coefficients were not set for external tuning. The default value for all parameters is 0.5 from the (B) version.

`ValidityWeight` (R10) [0.0-1.0] [Float] is a coefficient that shows how each contract violation decreases the quality of the solution.

[3] `https://github.com/google/guice`

Table 3.2. Optimized parameters of WHC-T version of the genetic solver

Genetic solver	R1	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
WHC-T	SPEA2	2014	0.98	0.95	0.58	0.02	0.64	0.3	0.95	0.17	79	266

`SoftwareValidityWeight` (R11) [0.0-1.0] [Float] is a the coefficient that shows how each error in the software tree decreases the quality of the solution.

As described in Section 2.5, a creator creates random individuals for the initial population. This method has two features that are necessary to obtain better individuals at the initial stage. These features have parameters:

`RandomSoftwareAssignmentAttempts` (R12) [1-1000] [Integer] is a max number of attempts to set a valid software tree to the individual on the creation phase. The default value is 50.

`populateSoftwareSolutionAttempts` (R13) [1-1000] [Integer] is a max number of attempts to assign software components and resources to get a valid individual on the creation phase. The default value is 100.

We mark this version[4] of the genetic solver as **WHC-T** (Without Hard-Code-Tuned).

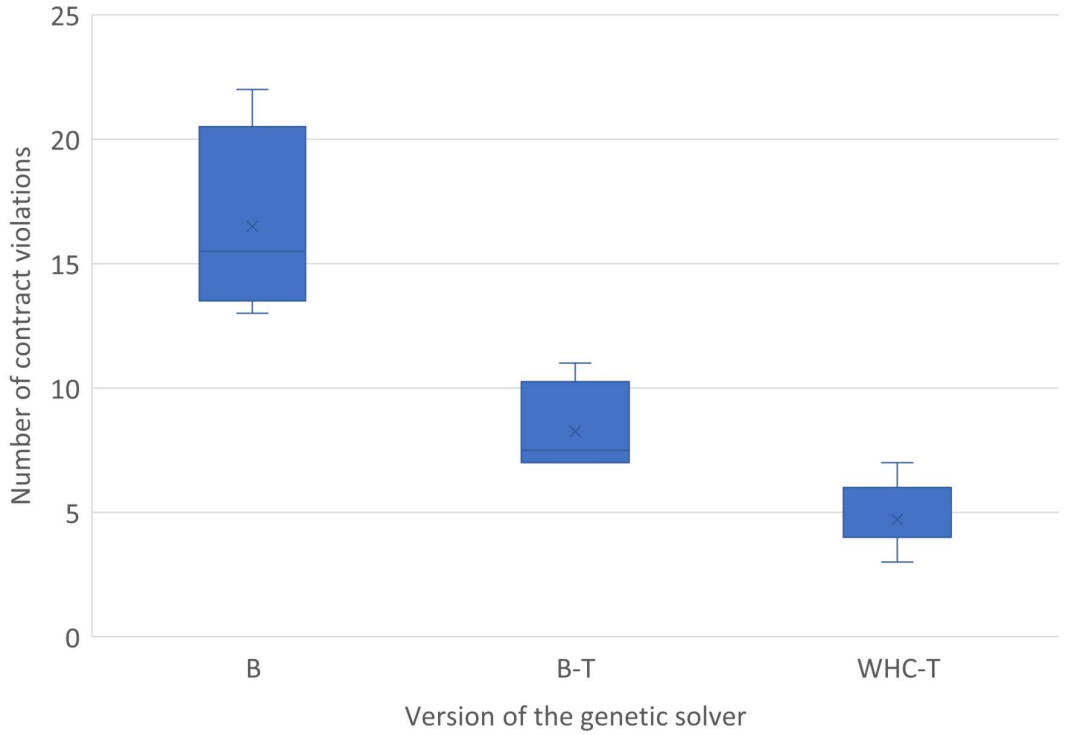


Figure 3.3. Box-plot with a number of contract violations for the genetic solver without hard-coded parameters in comparison with previous versions

After optimization, we use an optimized set of values of hyperparameters from BRISE. Table 3.2 depicts a set of optimized parameter values. With more parameters, BRISE

[4]commit: e1db4a941622f9a6609ffcfb5d05df9e7abaffc2

suggests using the *SPEA2* selector algorithm, but the value of the `CrossoverRate` (R5) is not changed.

Figure 3.3 depicts the results of the WHC-T version of the genetic solver as a box-plot in comparison with earlier versions (B, B-T). This box-plot shows that the WHC-T version of the genetic solver is better in comparison to B-T. The max number of contract violations in WHC-T is 7. This number is equal to a minimum number of contract violations from the B-T version. But solutions are still not valid for the current problem.

This section shows how correctly selected parameter values affect the result of the algorithm in terms of solution validity. For the described problem without complex changes in the algorithm, we get a result which, in the worst-case, gives a solution that is almost twice as good as the best solution in the B version of the genetic solver. However, parameter optimization is not enough to obtain valid solutions for the Problem described at the beginning of this chapter.

3.3. Fine-grained Crossover and Mutation Points

Since the parameter tuning from the previous steps do not give us valid results, we decide to use a parameter engineering approach. Detailed analysis of crossover and mutation operators shows that the crossover point and the mutation points are coarse-grained. The principle how these points define described in Section 2.1.2 and Section 2.1.3.

We add a few more parameters to crossover and mutation operators. These are probabilities, that allow change of the crossover/mutation points location in different ways in the Tree Shape Genotype. There are:

`CrossoverOnRandomChildProbability` (R14) [0.0-1.0] [Float] is a chance that crossover will occur on the random child or both children otherwise (since the MQuAT problem constraints described in Section 2.4);

`CrossoverOnRandomLevelProbability` (R15) [0.0-1.0] [Float] is a probability that describes a chance to do a crossover on random level of the tree;

`CrossoverOnRandomRequestProbability` (R16) [0.0-1.0] [Float] is a chance to do a crossover on random request,

`MutationOnRandomChildProbability` (R17) [0.0-1.0] [Float] is a probability that describes a chance of mutation on a random child;

`MutationOnRandomLevelProbability` (R18) [0.0-1.0] [Float] is a chance of mutation on a random level of the tree shape genotype.

The default value for there probabilities, except `CrossoverOnRandomRequestProbability` (R16), is 0.5. For the `CrossoverOnRandomRequestProbability` (R16) is 0.0. These values are received empirically during development.

This version[5] of the genetic solver is marked as NP (New Probabilities).

The set of parameters used during development is presented in Table 3.3. New parameters have the values described above. Furthermore, previously discussed

[5] commit: 817f7f1ef06f3acbbb4d5e24e32a26c5e6abc4b5

Table 3.3. Parameters of NP and NP-T versions of the genetic solver

Genetic solver	R1	R3	R4	R5	R6	R7	R8	R10	R11	R12	R13	R14	R15	R16	R17	R18
NP	NSGA2	500	0.1	0.95	0.1	0.45	0.5	0.5	0.5	50	100	0.5	0.5	0.0	0.5	0.5
NP-T	NSGA2	2550	0.5	0.5	0.5	0.5	0.5	0.5	0.5	500	500	0.5	0.5	0.5	0.5	0.5

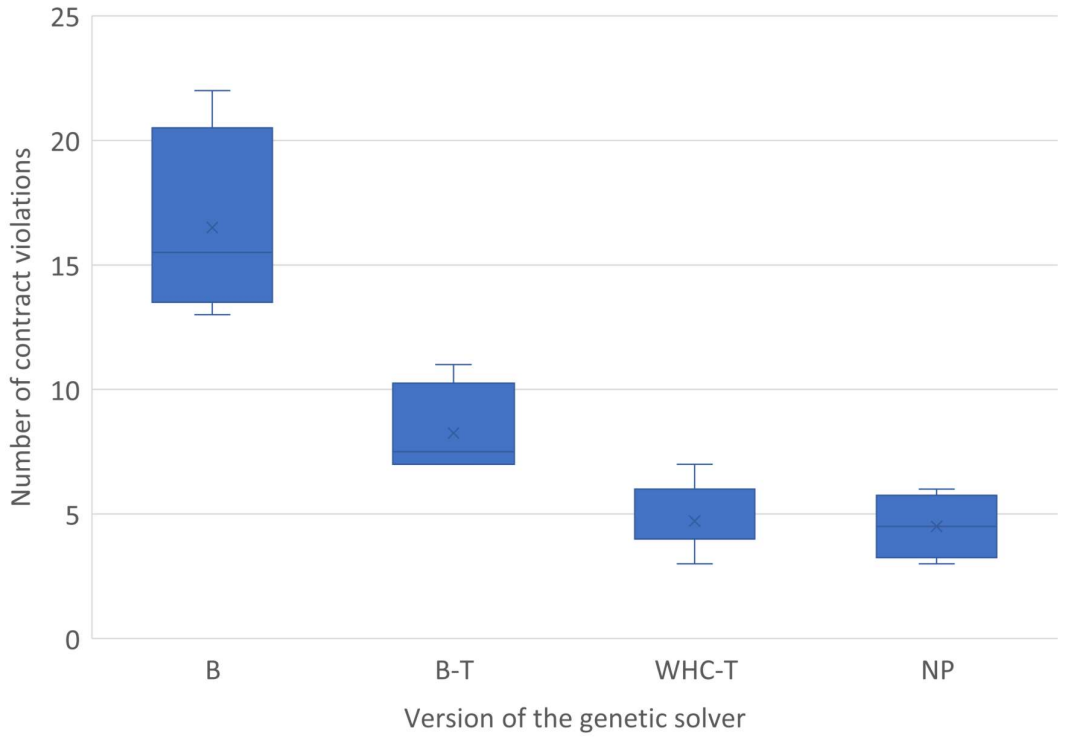


Figure 3.4. Box-plot with a number of contract violations for the genetic solver with added probabilities without tuning in comparison with previous versions

parameters have values from the B version of the genetic solver. That means that all parameters are not tuned.

The result of the **NP** version of the genetic solver as a box-plot in comparison with earlier versions (**B**, **B-T**, **WHC-T**) shown in Figure 3.4. If we compare the **B** and the **NP** versions, then both versions have non-optimized parameter values. The NP version has a minimal number of contract violations equal to 3. It is **3** times less violations than in the best case if the **B** version. Moreover, this version of the genetic solver has results comparable to the results after parameter tuning in the **WHC-T** version. We can conclude, that better-designed parameters could also improve the target system.

After adding new parameters, we start BRISE to optimize all parameters. The optimized set of parameters is in a Table 3.3. We mark the **NP** version with an optimized set of parameters as **NP-T**(New Probabilities-Tuned)[6]. As it was shown, parameter tuning with described parameters returns *NSGA2* as a selector. Moreover, BRISE decides that optimized values of parameters located in the middle of the values range.

Box-plots in Figure 3.5 show the comparisons of the NP-T version of the genetic solver to earlier discussed versions (**B**, **B-T**, **WHC-T**, **NT**). As shown, the combined approach of parameter engineering and parameter tuning gives a zero number of contract violations. It means that the **NP-T** version has a **valid** solution for the MQuAT problem described at the beginning of this chapter. Even the worth-case of the **NP-T** version with 2 contract violations is better than all versions presented in previous

[6]commit: e103f52b3333900d61c6218a1f2ca811bca10289

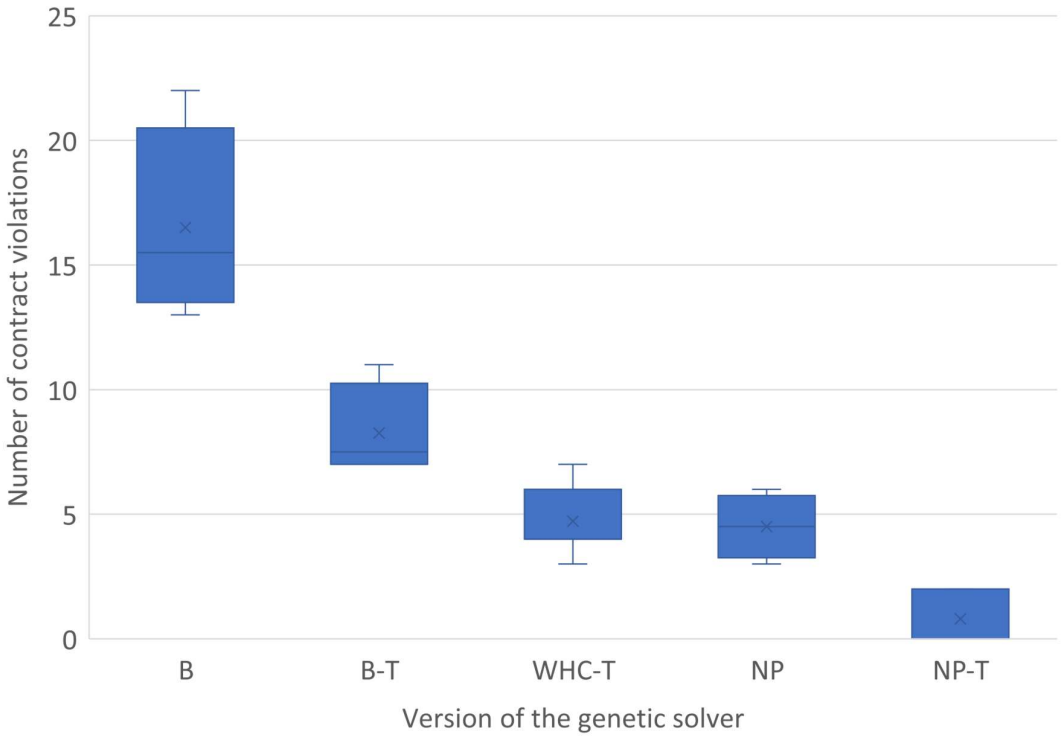


Figure 3.5. Box-plot with a number of contract violations for the genetic solver with added probabilities and tuned parameters in comparison with previous versions

sections.

This section showed that the parameters thought out in the algorithm are as important as the values of these parameters. If the algorithm is not designed correctly, the tuning parameter may improve the results, but this may not be enough. But not all solutions of the NP-T version are valid. The reason is that all versions of the genetic solver that were previously discussed trap into the local optimum. It could not get out because, at a certain moment, the majority of the individuals in the population becomes the same, and any newly created individual in most cases has a worse value of the objective function. But we try to solve it.

3.4. Simple Parameter Control

The reason why all individuals are the same is the creation of the offspring. To create a new individual, the selector takes the best individuals. A crossover takes two individuals from the selector, and performs a gene exchange, creating two new individuals. If these individuals give a worse solution, then the selector in the new iteration drops them, but if they are better, it takes them to create new individuals. At a certain point, there is a situation that the crossover occurs between two identical genotypes, which creates two identical to parents individuals. There is a case with a probability `1 - mutationRate` that mutation does not happen. It increases the number of identical elements and leads to the collapse of the genetic diversity of the population.

The idea of solving this issue is in internally changeable crossover rate. It changes the chance of the crossover depending on the number of identical individuals in a

Table 3.4. Parameters of ICCR and ICCR-T versions of the genetic solver

Genetic solver	R1	R3	R4	R5	R6	R7	R8	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20
ICCR	NSGA2	500	0.1	0.95	0.1	0.45	0.5	0.5	0.5	50	100	0.5	0.5	0.0	0.5	0.5	0.25	0.75
ICCR-T	SPEA2	2550	0.5	0.5	0.5	0.5	0.5	0.5	0.5	500	500	0.5	0.5	0.5	0.5	0.5		

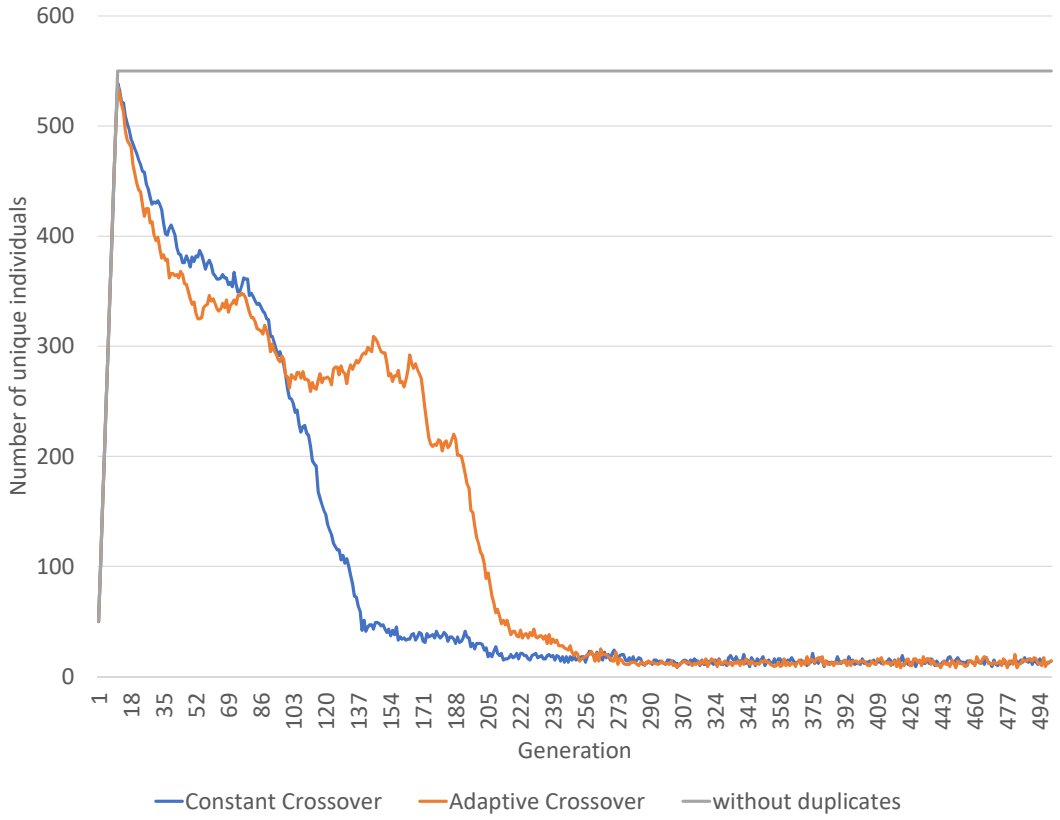


Figure 3.6. Number of unique individuals in population per generation

population. The principle of operation is as follows. After each iteration, the ratio of unique genotypes in the population is calculated. If the resulting number is smaller than the value of `PartOfUniqueIndividualsToStopCrossover` (R19) [0.0-1.0] [Float] parameter, then the probability of a crossover sets to 0.0, and vice versa, the crossover rate returns its value if the ratio of unique genotypes in the population is bigger than the value of `PartOfUniqueIndividualsToReturnCrossover` (R20) [0.0-1.0] [Float] parameter.

The value for `PartOfUniqueIndividualsToStopCrossover` is 0.25 and for `PartOfUniqueIndividualsToReturnCrossover` is 0.75.

The version of the genetic solver with internally changeable crossover rate marked as **ICCR** (Internally Changeable Crossover Rate)[7] and tuned version[8] was marked as **ICCR-T** (Internally Changeable Crossover Rate-Tuned).

Table 3.4 shows the parameter values for both versions. If we compare the optimized values of parameters from the previous section, we can see that *NSGA2* selector changes to *SPEA2*. All other values are the same.

The internally changeable crossover rate slightly delay the collapse of the diversity of individuals in the population. Figure 3.6 shows the deterioration in population diversity for a constant crossover rate and an internally changeable crossover rate.

[7] commit: c89422c6e46a5f4e8bc09205df7713ad8fe6907c
[8] commit: 128a6f2f844edd70d0e8ee616f09ac897cb86f4e

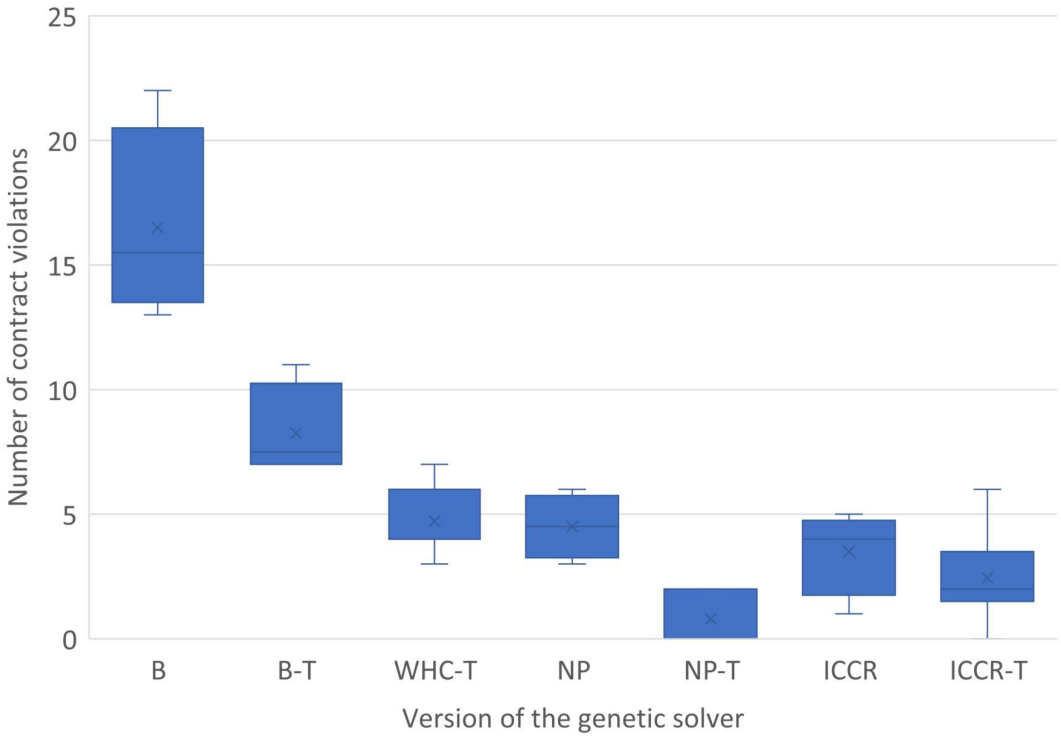


Figure 3.7. Box-plot with a number of contract violations for the genetic solver with internally changeable crossover rate in comparison with previously discussed versions

The comparison of the results of ICCR and ICCR-T versions with all earlier discussed versions is presented as box-plots shown in Figure 3.7. The ICCR version gives better results than all untuned versions of the genetic solver. However, the range between max and min number of contract violations in the ICCR version is bigger than in the NP version. The ICCR also better than tuned versions such as B-T and WHC-T, but worse than the NP-T version. The ICCR-T version as an NP-T version gives a valid solution, but in more like an exception case. The box-plot of the ICCR-T version also shows that parameter tuning for internally changeable parameters works not so efficient as for static parameters like in NP version, at least for the discussed problem.

This section shows that parameter control improves the results and postpone stuck in a local optimum. However, parameter tuning for static parameters does not work so well on parameters that change their values during the run-time.

3.5. Unique Genotypes in a Population

Since the changeable crossover rate from the previous section only delays the collapse of the diversity of individuals in the population, we try a more radical solution. We block the possibility of adding to the population a new individual if the population already contains the same individual. It means that at any moment, the population will consist of unique individuals. In this version of the genetic solver, the use of the changeable crossover rate does not make any sense, so it is disabled, and constant value is used.

Untuned version of the genetic solver with blocked duplicates in the population

Table 3.5. Parameters of WD and WD-T versions of the genetic solver

Genetic solver	R1	R3	R4	R5	R6	R7	R8	R10	R11	R12	R13	R14	R15	R16	R17	R18
WD	NSGA2	500	0.1	0.95	0.1	0.45	0.5	0.5	0.5	50	100	0.5	0.5	0.0	0.5	0.5
WD-T	SPEA2	1057	0.49	0.66	0.82	0.04	0.95	0.09	0.04	929	383	0.68	0.85	0.51	0.41	0.65

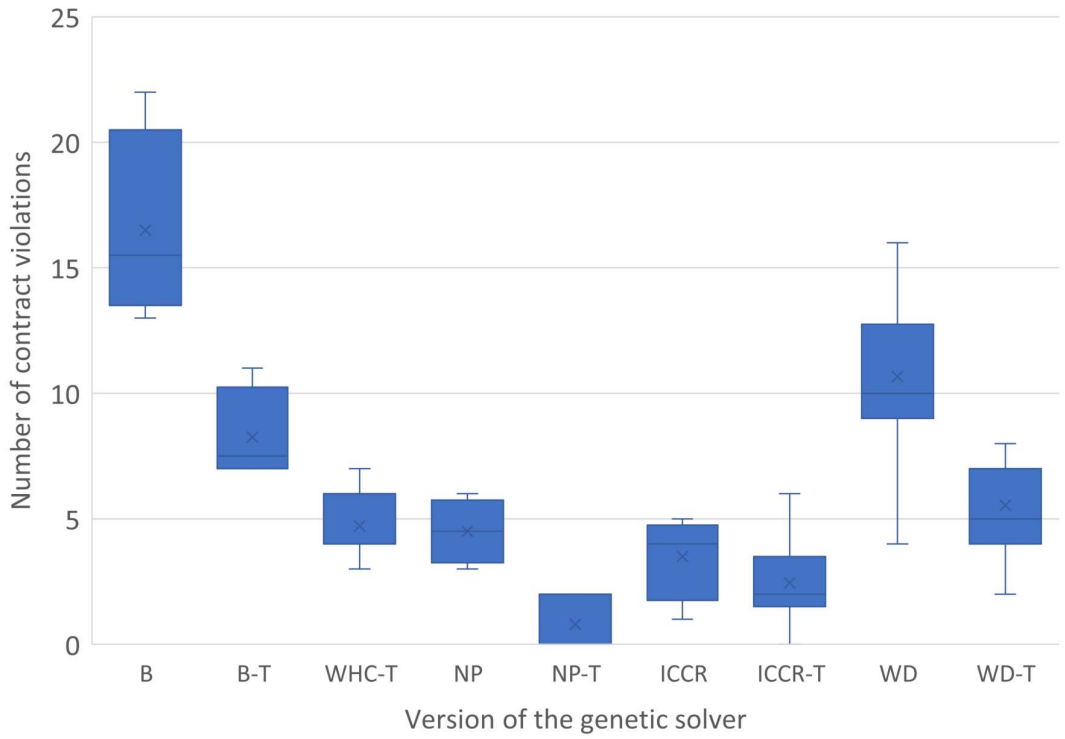


Figure 3.8. Boxplot with a number of contract violations for the genetic solver without duplicates in the population in comparison with previous versions

marked as **WD**[9] (Without Duplicates) and tuned version[10] marked as **WD-T** (Without Duplicates-Tuned).

The number of unique individuals in the population for the WD version is presented in Figure 3.6. In each generation, after the initial population has been created, the number of unique elements in the population is not changing.

Table 3.5 shows parameter values for both versions .

Figure 3.8 shows the results of this approach with and without parameter tuning. Obtain results show that the WD version of the genetic solver can give better results only in comparison to the B version. The WD-T version gives better results with less range between min and max number of contract violations than WD. But it much worse than NP-T.

There are two main reasons why the results of this approach are much worse than previous versions. The first one is a calculation speed. It is slower than in previous versions. As a result, the number of created generations is lower. For the NP version, this number is 15177. For the WD version, this number is 2605, more than five times less according to our tests.

The second reason is a few "good" individuals are selected to create offspring. This number is low because the selector chooses the best individual from the population only once per selection. As a result, the best individual gives a few new individuals.

[9]commit: 1d79c50d7932c9216c653bf6d0354d990f6aecbc
[10]commit: 24504c77024ac383c77849c5121471e0f06ce913

Table 3.6. Parameters of the Genetic Solvers

Genetic solver	R1	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20
B	NSGA2	500	0.1	0.95	0.1	0.45	0.5	0.8	0.5	0.5	50	100							
B-T	NSGA2	1533	0.1	0.95	0.1	0.45	0.5	0.8	0.5	0.5	50	100							
WHC-T	SPEA2	2014	0.98	0.95	0.58	0.02	0.64	0.3	0.95	0.17	79	266							
NP	NSGA2	500	0.1	0.95	0.1	0.45	0.5		0.5	0.5	50	100	0.5	0.5	0.0	0.5	0.5		
NP-T	NSGA2	2550	0.5	0.5	0.5	0.5	0.5		0.5	0.5	500	500	0.5	0.5	0.5	0.5	0.5		
ICCR	NSGA2	500	0.1	0.95	0.1	0.45	0.5		0.5	0.5	50	100	0.5	0.5	0.0	0.5	0.5	0.25	0.75
ICCR-T	SPEA2	2550	0.5	0.5	0.5	0.5	0.5		0.5	0.5	500	500	0.5	0.5	0.5	0.5	0.5	0.5	0.5
WD	NSGA2	500	0.1	0.95	0.1	0.45	0.5		0.5	0.5	50	100	0.5	0.5	0.0	0.5	0.5		
WD-T	SPEA2	1057	0.49	0.66	0.82	0.04	0.95		0.09	0.04	929	383	0.68	0.85	0.51	0.41	0.65		

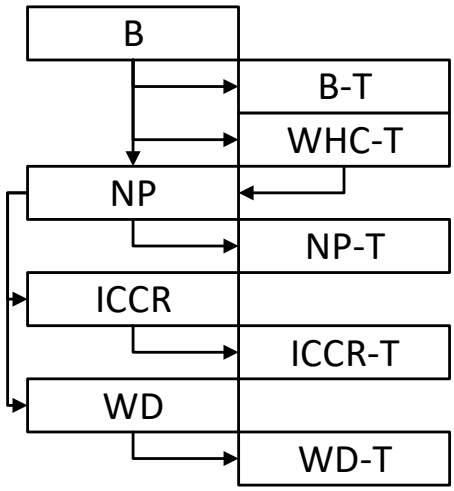


Figure 3.9. Genetic Solver version history

Theoretically, the small value of the `mu` parameter could help. Nevertheless, BRISE gives the value of the `mu` parameter even higher than the default value.

The conclusion of this idea is quite pessimistic because other approaches work better on this problem. Still, it may work better than others with different problems or on long term optimization.

There are many modifications. To get some understanding of how all versions appear, we create a version history scheme. Figure 3.9 shows it. The time goes from the top to bottom of the scheme. Some versions such as **B-T** or **NP-T** change parameter values. The **NP** version uses modification of the **WHC-T** version and additional parameters, but use parameter values from the **B** version.

This chapter shows that parameter tuning improves the results of the Genetic Solver for the specified problem. Table 3.6 presents parameter values comparison of all presented versions of Genetic Solver. In this chapter, we mentioned that we are focusing on solving the specified problem. But how optimized parameter values work with other problems of the same class? To analyze it, we perform the evaluation.

4. Evaluation and Analysis

In the previous chapter, we present a few approaches that could improve the genetic solver. The best results give the NP version of the genetic solver. It is a combination of the genetic solver with added probabilities and parameter tuning. Other methods like a crossover rate with parameter control or population without duplicated individuals have less effect on the results.

However, we get valid results only for one problem. But what if we change the complexity of the problem? We perform the benchmark to find out how all approaches work with different problem sizes.

4.1. Evaluation

To evaluate the genetic solver, we perform a set of benchmarks that consist of 36 problems (see Appendix B). Each problem has different parameters that describe (Section 2.4) it:

- Software variants: [2, 4];
- Number of requests: [1, 2, 4];
- Component tree depth: [2, 3, 4];
- Resources ratio: [50, 100].

This set of problems is tested with several versions of the genetic solver:

- base (B);
- basic with tuned parameters (B-T);
- without hard-coded parameters and tuning (WHC-T);
- with added parameters (NP);
- with added parameters and tuning (NP-T);
- with internally changeable crossover rate (ICCR);
- with internally changeable crossover rate and tuning (ICCR-T);
- without duplicates in the population (WD);
- without duplicates in the population and tuning (WD-T).

Each version of the genetic solver tries five times to solve each problem with 5 minutes timeout .

We use the Intel Core i7-8700 CPU machine with 64Gb of memory using Fedora Server 29 and OpenJDK 1.8.0 201-b09.

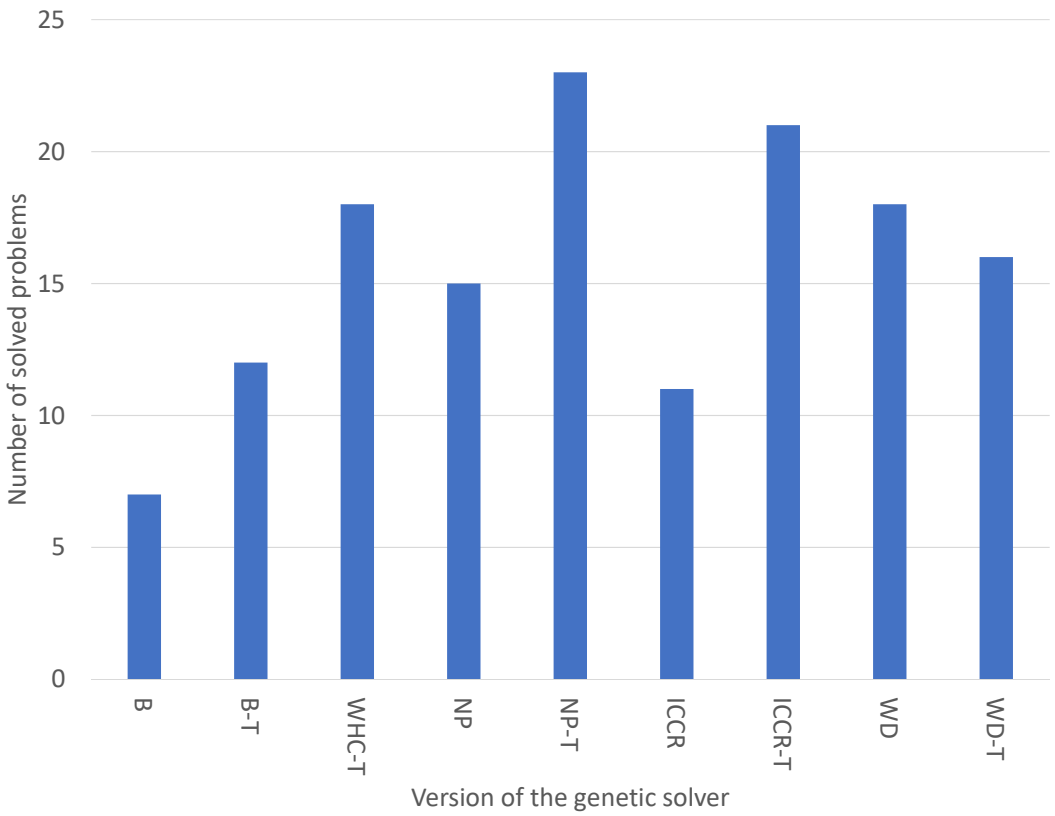


Figure 4.1. Number of solved problems for each version of the genetic solver

The genetic solver tries to solve each Problem. If a solution is valid, genetic solver start to solve the next problem. If, after five attempts, the genetic solver does not find a valid solution, it proceeds to the next problem.

Figure 4.1 shows the results of the benchmark. For each version of the genetic solver, a vertical bar is build. The height of the bar shows the number of solved problems from the benchmark set.

The B version solved the least number of problems than any other version. Almost in all cases, parameter tuning increases the number of solved problems. It also confirms the answer to RQ1.

The comparison of B-T, WHC-T, and NP versions shows that the NP bar is higher than B-T but lower than WHC-T. This comparison confirms the first conclusion from Section 3.3, that well-designed parameters are important for any algorithm. The combination of parameter engineering and parameter tuning in NP-T gives the best results. The same situation was observed for one problem in the previous chapter (Figure 3.8).

Figure 4.1 shows the results of the benchmark. For each version of the genetic solver, a vertical bar is build.

The benchmark also shows that no version of the genetic solver solves all problems from the set (see Appendix B). Table B.1 shows the results of the benchmark in the context of a solved or unsolved problem. Each row describes the result of a specific problem. The column *Problem Id* contains problem numbers. Black filled cell means that solver solved the problem. Table B.1 also shows ids of problems that were not solved by all versions of the genetic solver.

Unsolved problems are presented in Table 4.1. It shows parameters that specified the problem. We can see there exist a few types of unsolved problems. All versions of the genetic solver could not solve the problem that contains 2 or more requests,

Table 4.1. Not solved problems

Problem Id	Software variants	umber of requests	Component tree depth	Resources ratio
6	2	2	4	50
8	2	4	3	50
9	2	4	4	50
15	2	2	4	100
17	2	4	3	100
18	2	4	4	100
24	4	2	4	50
26	4	4	3	50
27	4	4	4	50
30	4	1	4	100
33	4	2	4	100
35	4	4	3	100
36	4	4	4	100

and the depth of the tree is 3 or more. The exception is problem number 30. It has 1 request and depth equal to 4, but it also has 4 variants, and 100 resources and genetic solvers could not solve it.

The depth of the tree has the most significant impact since it exponentially increases the number of nodes. As a result, there are more possible mappings between the software components and hardware resources. Other parameters with higher value give little increase in the number of possible combinations.

If a genetic solver could solve some problems, then solutions have a quality. Let us now discuss the quality of the received results. Figure 4.2 shows the percentage of the deviation from the optimum for problems that solved by all versions of the genetic solver. The optimum values for problems were received by the ILP solver. If the percentage of the deviation is zero, that means that the received solution is optimal. The max deviation is near 30% for the B version with problem 31. However, other versions give solutions with deviation from optimal that less than 10% or even optimal solutions for the NP-T version. There are high deviations from optimum in problems number 13 for tuned versions of the genetic solver such as B-T, NP-T, ICCR-T, and WD-T. Nevertheless, WHC-T and untuned versions give a near-optimal solution with minimal deviation. The ICCR version gives a much higher percentage of deviation than other versions in problem 19.

Figure 4.2 also shows that the deviation is increasing for big size problems. To confirm that fact, we take two problems with different sizes (Figures 4.3). These problems are solved by all versions of the genetic solver except the B version. Figure 4.3a shows the percentage of deviation from optimum for a small size problem. This problem's parameters are:

- Software variants: 2;
- Number of requests: 2;
- Component tree depth: 2;
- Resources ratio: 50;
- Timeout to solve the problem: 5 minutes.

The quality of results for the problem is near-optimal because the deviation is less than one percent. The NP-T and WD-T versions give an optimal solution for the problem.

Figure 4.3b shows the percentage of deviation from optimum for a bigger size problem. This problem's parameters are:

- Software variants: 4;

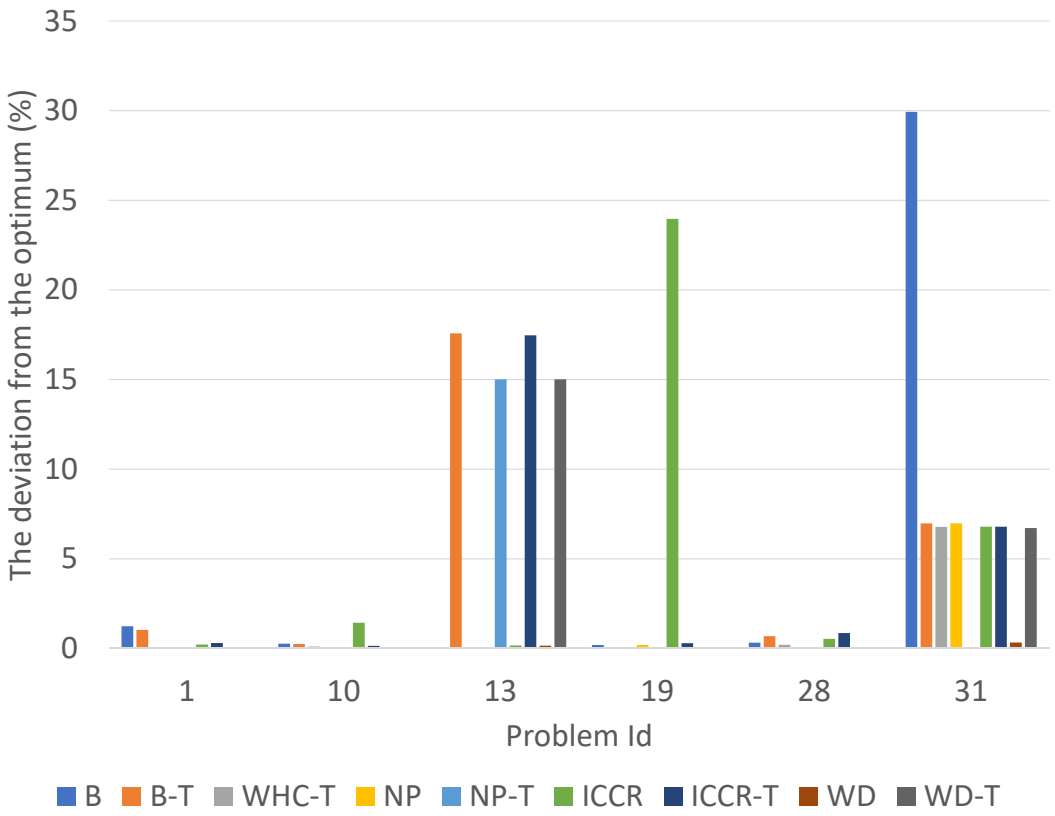


Figure 4.2. The deviation from the optimum of solved by all solvers problems

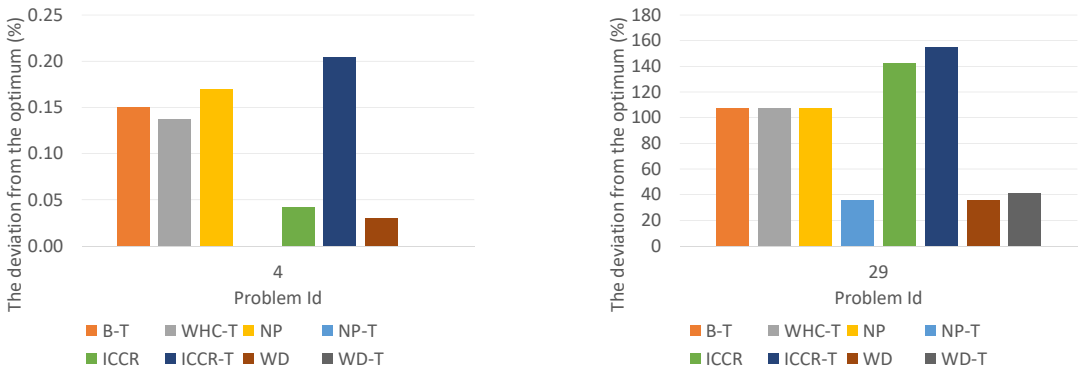
- Number of requests: 1;
- Component tree depth: 3;
- Resources ratio: 100;
- Timeout to solve the problem: 5 minutes.

The quality of results in a case of the bigger problem is much worse. The minimal deviation is 35% for the **NP-T** version. The **WD** and **WD-T** versions give less number of valid solutions, but with better quality. The reason why the quality of solutions is decreasing with a bigger size of the problem could be a higher number of hardware resources. The higher number of resources means that more hardware resources could satisfy the requirements of the software component. The genetic solver could not find best-suited resources for requested components, and as a result, quality is decreasing.

This section shows that our modifications improve the results of the genetic solver. The **NP-T** version solved **3 times more** problems than the **B** version. We evaluated the quality of solutions. As a result of the quality evaluation, we make two conclusions. First, the modified versions of the genetic solver also improve the quality of the results. Second, the quality is decreasing for bigger size problems. However, there are problems from the set that the genetic solver can not solve.

4.2. Analysis

Benchmark showed that a genetic solver could not solve all MQuAT problems from the evaluation set. In this section, we present a discussion of results and reasons why



a) Small sized problem **b)** Big sized problem

Figure 4.3. The deviation from the optimum for small and big sized problem

described in Chapter 3 enhancements, and optimizations are not giving the desired efficiency.

Firstly, let us discuss the reasons why modifications improve the genetic solver. After that, we will describe the reasons why not all problems could be solved.

Positive reasons There are a few reasons why the genetic solver with our modifications could solve more problems than the B version.
First, parameter tuning is performed for all versions of the genetic solver. Optimized values of parameters give better results, as it showed in the previous section.
The second reason is in new probabilities that we added in Section 3.3. These probabilities give a possibility to change the position of the crossover and mutation points randomly, but as a result, the genetic solver gives results that a bit worse than parameter tuning.

Negative reasons There are a few reasons why not all problems are solved.
One of the reasons for such a result is that we optimized parameters for a specific problem that we described at the beginning of Chapter 3.
Obtained results could be explained with a "no free lunch" (NFL) theorem [38, 39]. It states that if an algorithm is performing well with a certain class of tasks, then it must pay for it with a deterioration in performance on the set of all remaining problems.
Another reason that we optimize parameters without knowledge about dependencies between them. Parameters that we found or added did not fit well for our goals. Let us analyze earlier discussed parameters.

4.2.1. Search Space

We start the analysis by determining the search space structure. We perform measurements for more than three thousand configurations using BRISE in the Search space exploration mode with the NP version of the genetic solver. This mode was described in Section 2.3. For this type of analysis, we use the same problem as in Chapter 3 and it has following properties:

- Software variants: 10;
- Number of requests: 15;
- Component tree depth: 2;
- Resources ratio: 5.

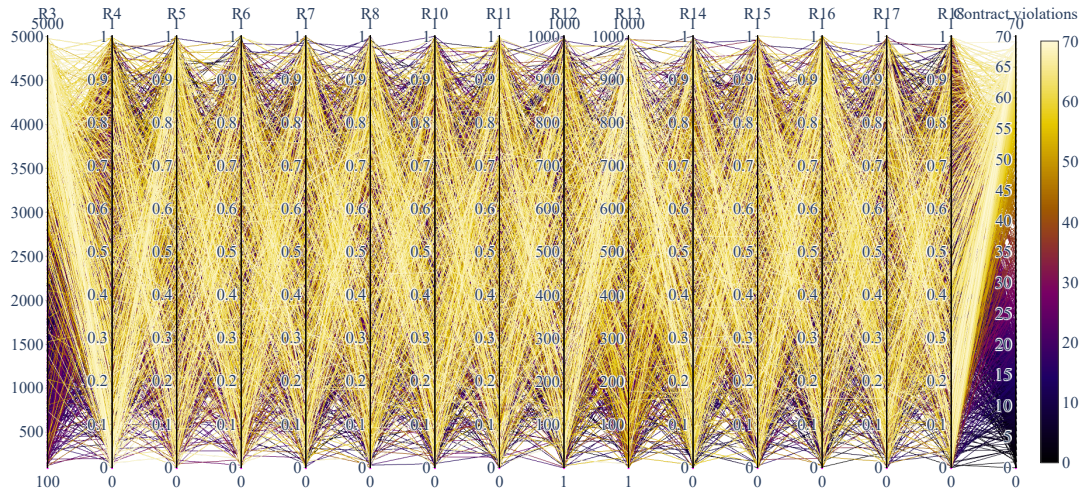


Figure 4.4. Search space representation

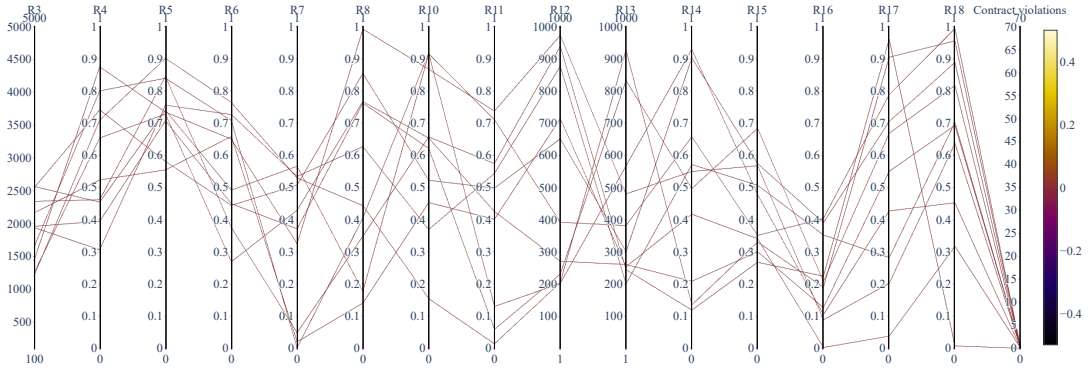


Figure 4.5. Search space representation of valid results

The search space for the SPEA2 selector as a parallel coordinates plot is shown in Figure 4.4. Each vertical line represents a parameter, its values range. Each measured configuration is shown on the plot as a line that connects parameter values for the Genetic Solver. The last vertical line, as well as the color, indicates the number of contract violations that the configuration gives. The more dark color means fewer contract violations and vise versa, the configuration with lighter color gives a higher number of contract violations.

Figure 4.4 shows that all values of parameters could give "good" and "bad" results. There are no visible dependencies between parameters.

If we filter out all configurations that give not valid results, we get the search space with a few configurations. Figure 4.5 demonstrates these configurations. As shown, there are ranges of values that give valid results. Parameters such as R4 and R5 have the range on the higher values. However, R16 needs to be smaller.

Since Figure 4.4 and Figure 4.5 show that the search space is very complex, we perform another method of analysis to analyze dependencies between parameters.

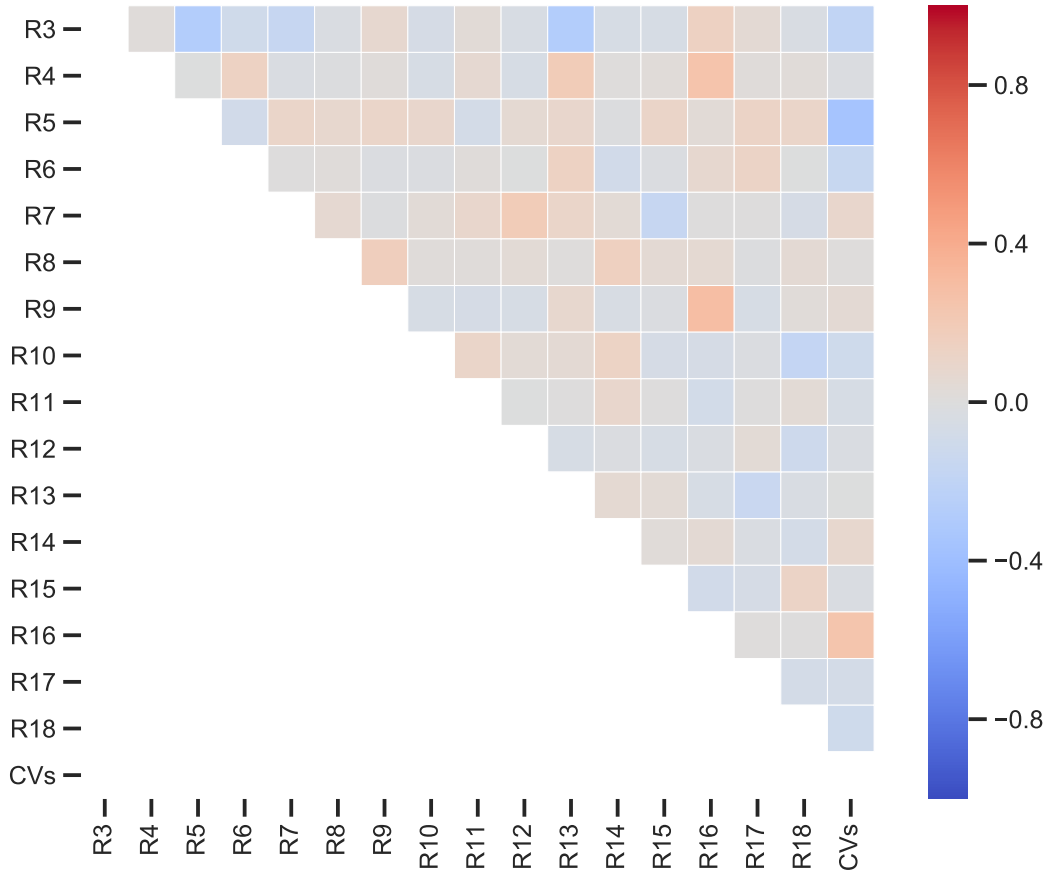


Figure 4.6. Correlation analysis matrix

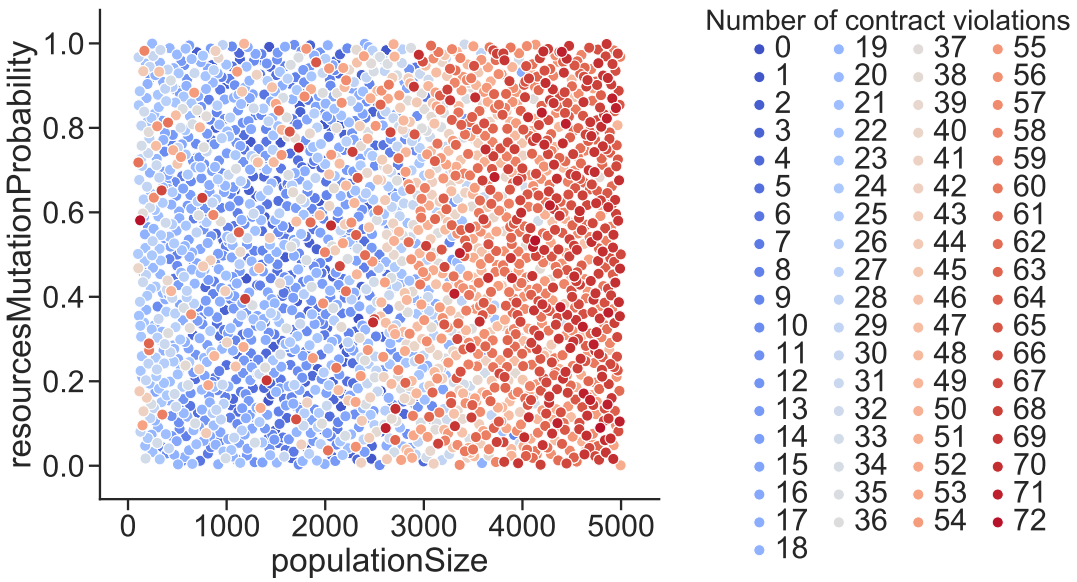


Figure 4.7. Parameter pair distribution of `populatioSize` (R3) and `resourcesMutationProbability` (R8)

4.2.2. Correlation Analysis

We perform the correlation analysis of parameters to understand how they rely on each other. Figure 4.6 shows the results as a correlation matrix where the intensity of the correlation is the color of the cell. The more intense color means a stronger correlation. Each row and column in this matrix represent the parameter of the genetic algorithm. We display only an upper triangle of the matrix because it is symmetric. The last column of the matrix represents the correlation between parameters of the genetic solver (NP version) and a number of contract violations (CVs). There are two types of correlation. A direct correlation has a red color, and an inverse correlation has a blue color. In the case of correlation to CVs, direct correlation means that a bigger value o parameter gives a bigger number of contract violations. Inverse correlation, in the same case, means that the bigger value of parameter gives a smaller number of contract violations. Grey color shows that there is no correlation.

From the correlation analysis, we can conclude that there are no strong dependencies between the number of contract violations and parameters. There are two more important parameters: `CrossoverRate` (R5) and `CrossoverOnRandomRequestProbability` (R16). The analysis showed that the `CrossoverRate` parameter needs to have big value and value of `CrossoverOnRandomRequestProbability` needs to be as low as possible. In this case of the small value of probability, it may be a good idea to remove this parameter for parameter tuning.

Correlation analysis shows that some parameters have dependencies. Nevertheless, we do not know yet how they depend.

4.2.3. Parameter Pairs Distributions

For further investigation of dependencies, we construct plots that demonstrate all parameter values of two different parameters. Each point on the plot is a combination of two parameter values. We are calling them as distribution of parameter pairs. We will discuss only two distribution. The first plot will show one particular distribution, and the second plot represents how most of the constructed distributions look like.

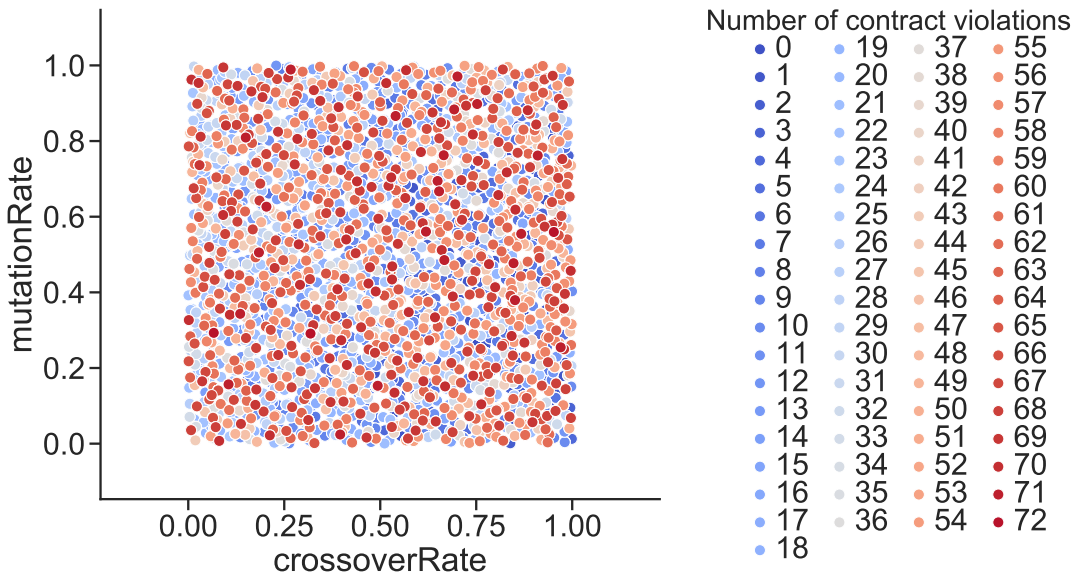


Figure 4.8. Parameter pair distribution of `CrossoverRate` (R5) and `mutationRate` (R7)

Figure 4.7 shows the representative distribution. Each point on this plot represent a value combination of two parameters `populationSize` (R3) and `resourcesMutationProbability` (R8). The color of the point is the number of contract violations. In this case, blue color is a small number of contract violations. Red color represents a big number of contract violations. As we can see, there is gradient coloring from the mostly blue on the left size to mostly red on the right size. Such result means that `populationSize` (R3) have a bigger influence on the result than `resourcesMutationProbability` (R8). The figure also shows that for any value of `resourcesMutationProbability` (R8), there are results with any number of contract violations.

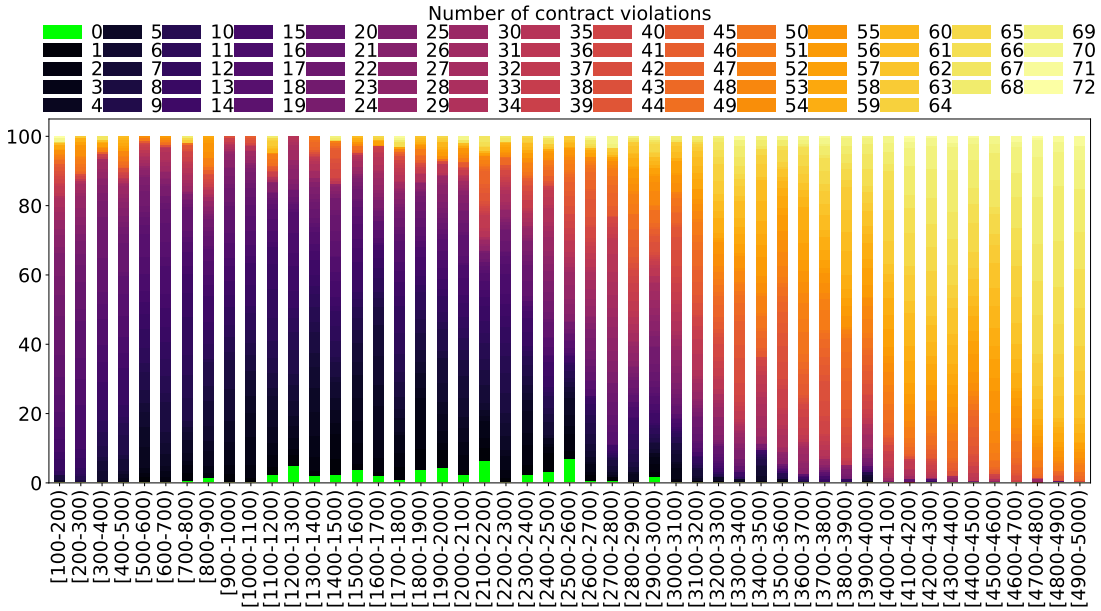
However, most distributions look like the distribution of `crossoverRate` (R5) and `mutationRate` (R7). It is showed in Figure 4.8. As we can see, "good" and "bad" results scatter over all values of the described parameters. That means that these parameters do not depend on each other.

Distributions of combinations of two parameters shows that some parameters have a more significant impact on the result than others. Moreover, there are parameters on the value of which the result does not depend. More examples of such distributions showed in Appendix C.

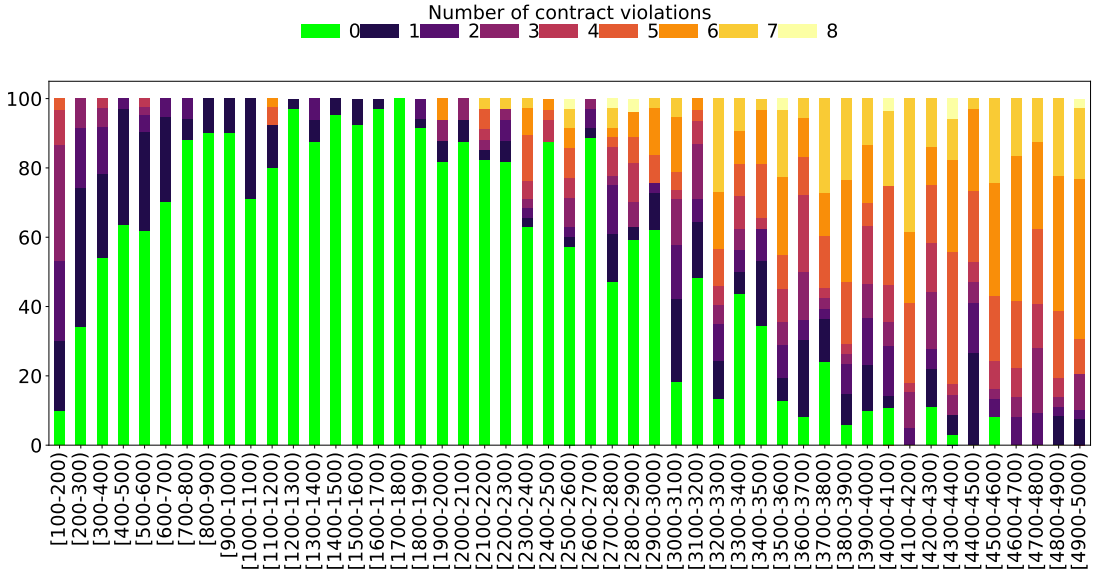
4.2.4. Parameter Values Distributions in Terms of Contract Violations

Previously discussed plots show that the results quality of the genetic solver highly depends on parameters such a `populationSize` (R3). So we need to analyze how the value of this parameter affects the result. Firstly we discuss the `populationSize` (R3) parameter and later the `mutationRate` (R7) parameter.

Figure 4.9a shows the distribution of the `populationSize` (R3) parameter values and the number of contract violations that this value could give. The X-axis is a set of ranges of values for the parameter `populationSize` (R3). Each range consists of 100 values. All ranges are sorted. A normalized bar is built for each range. Each bar consists of segments of different heights and colors. The segment color shows the number of contract violations, where dark is a few violations, and yellow is many violations. The green color of the segment indicates valid results (zero contract violations). The



a) Bigger sized problem



b) Smaller sized problem

Figure 4.9. `populationSize` (R3) parameter values distribution of two problems in terms of contract violations

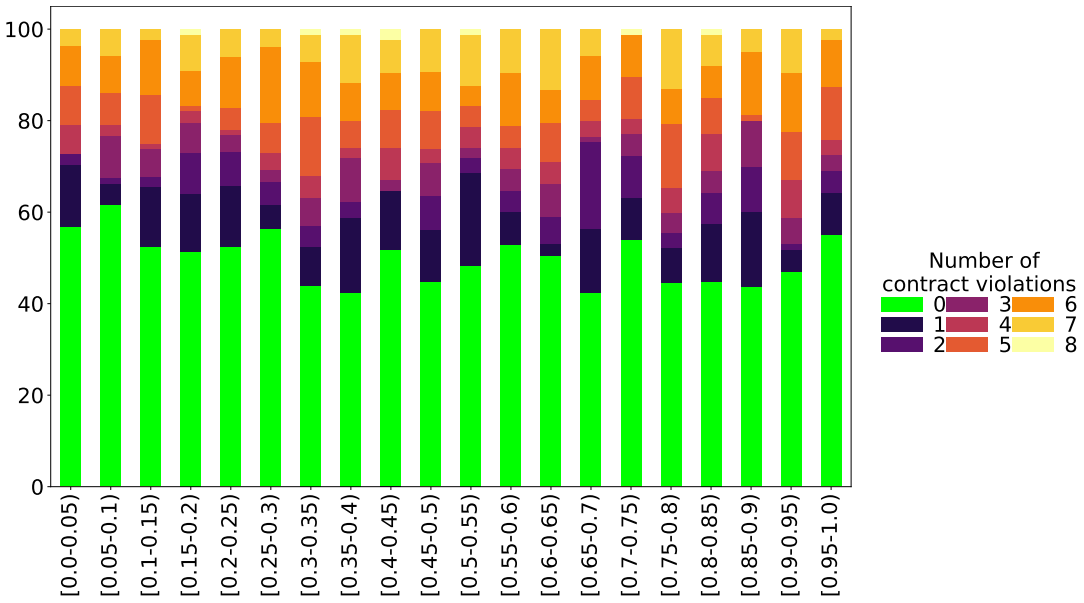


Figure 4.10. `mutationRate` (R7) parameter values distribution in terms of contract violations

segment height of one color means a percentage of the total number of configurations that have the same parameter value and which give the result with the same number of contact failures.

This distribution shows that the left side and the central part of the plot give a smaller number of contract violations because sectors have mainly dark colors and green color. The right side of the distribution contains higher values of the parameter. Moreover, solutions with those values give more contract violations. As can be seen, valid solutions are located in the range from 1000 to 2600.

To highlight it visually, we construct a similar plot for the smaller problem. This problem is described with parameters:

- Software variants: 2;
- Number of requests: 2;
- Component tree depth: 2;
- Resources ratio: 5;
- Timeout to solve the problem: 5 minutes.

It is shown in Figure 4.9b. If we compare Figure 4.9b and Figure 4.9a, we can see that they have a similar distribution.

The second discussed parameter is `mutationRate` (R7). The values distribution of this parameter are shown in Figure 4.10. For better visual understanding, the distribution is built for a smaller problem described above. As we can see, the percentage of valid results for any described range of values of the `mutationRate` (R7) parameter varies from 50 to 60. The distributions of most parameters look like the distribution described here. Such a distribution means that the result of the genetic solver does not depend on the value of the parameter.

The two types of distribution of parameter values and the number of contract violations are discussed. Distributions for both problems for each parameter presented in Appendix E. These plots confirm the conclusions that we made above. Values of some parameters such as `populationSize` (R3), `crossoverRate` (R5), `populateSoftwareSolutionAttempts` (R13) and `crossoverOnRandomRequestProbability` (R16) have a higher influence on the result. The distribution of the `crossoverOnRandomRequest-`

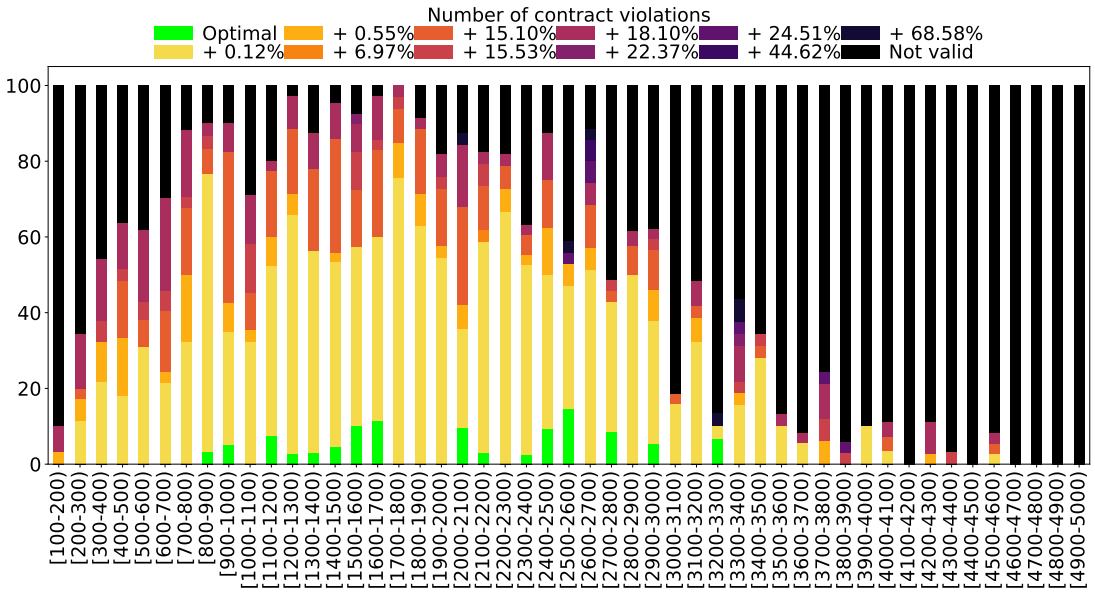


Figure 4.11. `populationSize` (R3) parameter values distribution in terms of quality deviation

`Probability` (R16) parameter, is showed in Appendix D, confirms our conclusions from correlation analysis that lower value gives better result. Some parameters give "good" and "bad" results for all values. That means that we could remove them from the parameter tuning and use constant value. Furthermore, for other parameters, values ranges could be adjusted.

4.2.5. Parameter Values Distributions in Terms of Quality

To answer the question of what value of parameters with no obvious advantage in distribution, we construct a similar distribution with the quality of the solution. In the previous analysis, we presented two distribution for `populationSize` (R3) and `mutationRate` (R7) parameters for smaller problem for smaller problem.

Figure 4.11 depicts the quality distribution for the `populationSize` (R3) parameter. This distribution differs from the previous one shown in Figure 4.9b in terms that all non-valid configurations are located in one segment that marked as "non-valid" and have a black color. Valid results consist of segments with different high and color. Each segment represents the percentage of the total number of configurations that have the same parameter value and which give the result near the same **quality**. All colors except black show the deviation of the quality of the solution from optimum in percent. **Green** color represents **optimal** solutions.

As we can see, if a Genetic Solver gets valid results, the quality of the Solution is optimal, or near-optimal. This distribution also confirms conclusions in Section 4.1.

For current analysis the distribution of the `mutationRate` (R7) parameter is more interesting and it is shown in Figure 4.12. The distribution of contract violations shows that any value of the parameter could give a valid result. However, the distribution of quality shows that some values of the parameter could give an optimal solution to the problem. Let us compare two ranges of values [0.45, 0.5) and [0.75, 0.8). Both ranges have near the same percentage of valid results of 50%. Nevertheless, the first range could give an optimal solution. The second range could not give such a solution. Moreover, the percentage of near-optimal solutions for the second range is lower.

The discussed above plots show that validity of the result in terms of number of

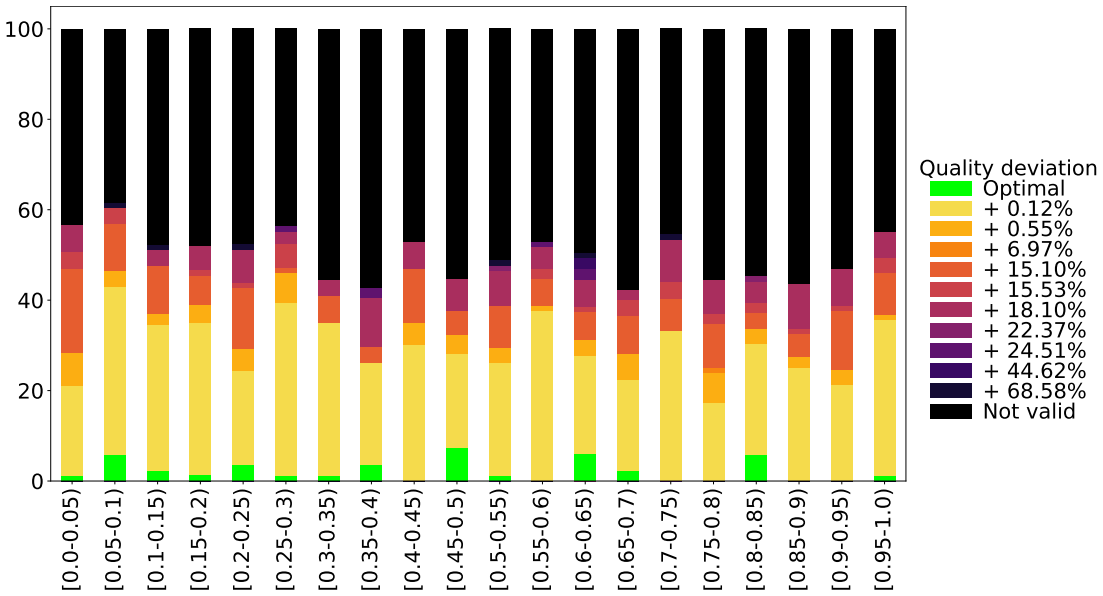


Figure 4.12. `mutationRate` (R7) parameter values distribution in terms of quality deviation

contract violations could not depend on the value of the parameter. However, the quality of the solution depends on the parameter value.

As a conclusion, we could say that we could remove some parameters from the parameter tuning if our goal is to find a valid solution. But we need all parameters if we are looking for the best quality of the solution.

5. Conclusion

In this thesis, we aim to improve a previously developed genetic solver by tuning its parameters. To achieve the goal, we analyzed the problem that the genetic solver tries to solve and how it solves the problem. We performed the analysis of parameter tuning techniques for evolutionary algorithms. The results of the analysis showed that the parameters of the genetic algorithm could be tuned empirically by the user. However, some automatic approaches help with parameter tuning. These approaches are distinguished by the presence of a model that predicts new parameter values. The first type is trying to find the best configuration manually from the landscape of all values. The second type, which relies on prediction models, after building the model predicts configurations that will give a better result of the genetic algorithm. It was decided to use the BRISE framework for parameter tuning of the Genetic Solver algorithm.

In Chapter 3, we iterative performed the parameter tuning. We analyzed the parameters of the genetic solver that exposed for external change and searched for the optimal parameters using the BRISE framework. The first iteration of parameter tuning was made on a single parameter. It showed that parameter tuning has a positive effect on the solver's final performance, at least for one specific problem. It is an answer for the RQ1. We exposed the parameters that had a hard-coded value for tuning. We created five new parameters describing the probabilities that move crossover and mutation points and also deleted one parameter that completely duplicated functions of the other parameter. That gave us an answer for the RQ2 that the genetic solver contains poor design decisions that can affect the results. Within the development process, we also tried to solve the issue of getting trapped at a local optimum. For this, we use a changeable crossover rate and ban the possibility to add a solution that is already in the population. We tested all modifications and parameter values on the problem described in Chapter 3 All the described modifications improve the result for a specific problem. It means that we completed the goal of this work for a specific problem.

We performed a benchmark to verify that optimized values of parameters and developed new parameters scale onto other Problem instances. The benchmarks set consists of 36 tasks with different complexity.

The results of the evaluation are described in Section 4.1. Benchmark showed that parameter values, optimized for a particular Problem, gave a better result for the whole set of problems. This fact means we have fulfilled our goal. There is also an exception that will be described in the next chapter and is a matter for further research.

The benchmark results show that the quality of the solution found deteriorates with

the increasing complexity of the problem.

A detailed analysis of optimized parameters was also performed. We conclude that the parameters do not have distinct dependencies and that not all parameters have the same importance for the genetic solver. Such parameters as the populationSize and crossoverRate have a more considerable influence on obtaining a valid result. At the same time, the quality of the final solution depends on the totality of all parameters.

Summarizing, we reached the research objective of this thesis. We identified and presented parameters that influence the quality of the solution. We improved the performance and increased the scalability of the genetic solver.

6. Future work

The research presented in this thesis shows that the Parameter Tuning could improve the Genetic Solver. However, it is not enough to solve big sized problems. As mentioned in Section 2.1.1, there is a framework limitation of used selector algorithms. There is a possibility that another selection algorithm for EA could further improve results. Because during the tuning parameter, BRISE gives optimized parameters that differed only in the selector.

Genetic solver without duplicates (WD) is a possible starting point for the feature research. We show that this modification works slower than other approaches because it compares each new individual with a set of unique individuals in the population. Nevertheless, the WD and WD-T versions of the genetic solver give results with good quality in less number of generation. Better implementation of this approach could improve the results of the genetic solver.

Another starting point for the future work is a parameter control inside the genetic solver. These approaches highly recommended by many researches.

Bibliography

[1] Jamal Ahmad. *A Comparative Study of Genetic Optimization Approaches in Multi-Quality Auto-Tuning*. 2018.

[2] Andrea Arcuri and Gordon Fraser. "Parameter tuning or default values? An empirical investigation in search-based software engineering". In: *Empirical Software Engineering* 18.3 (2013), pp. 594–623.

[3] Thomas Bartz-Beielstein, Christian WG Lasarczyk, and Mike Preuß. "Sequential parameter optimization". In: *2005 IEEE congress on evolutionary computation*. Vol. 1. IEEE. 2005, pp. 773–780.

[4] Thomas Bartz-Beielstein, Konstantinos E Parsopoulos, Michael N Vrahatis, et al. "Analysis of particle swarm optimization using computational statistics". In: *Proceedings of the international conference of numerical analysis and applied mathematics (ICNAAM 2004)*. 2004, pp. 34–37.

[5] 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.

[6] Arthur E Carter and Cliff T Ragsdale. "A new approach to solving the multiple traveling salesperson problem using genetic algorithms". In: *European journal of operational research* 175.1 (2006), pp. 246–257.

[7] Jeff Clune et al. "Investigations in meta-GAs: panaceas or pipe dreams?" In: *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*. 2005, pp. 235–241.

[8] Kenneth De Jong. "Parameter setting in EAs: a 30 year perspective". In: *Parameter setting in evolutionary algorithms*. Springer, 2007, pp. 1–18.

[9] Kenneth Alan De Jong. *Analysis of the behavior of a class of genetic adaptive systems*. Tech. rep. 1975.

[10] K. Deb and H. Jain. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints". In: *IEEE Transactions on Evolutionary Computation* 18.4 (Aug. 2014), pp. 577–601.

[11] Kalyanmoy Deb et al. "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II". In: *International conference on parallel problem solving from nature*. Springer. 2000, pp. 849–858.

[12] Agoston E Eiben and Selmar K Smit. “Parameter tuning for configuring and analyzing evolutionary algorithms”. In: *Swarm and Evolutionary Computation* 1.1 (2011), pp. 19–31.

[13] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*. Vol. 53. Springer, 2003.

[14] Torbjörn Ekman and Görel Hedin. “The JastAdd system—modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[15] Nicole Gockel and Rolf Drechsler. “Influencing parameters of evolutionary algorithms for sequencing problems”. In: *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*. IEEE. 1997, pp. 575–580.

[16] Sebastian Götz. “Multi-Quality Auto-Tuning by Contract Negotiation”. PhD thesis. Saechsische Landesbibliothek-Staats-und Universitaetsbibliothek Dresden, 2013.

[17] Sebastian Götz et al. “A JastAdd-and ILP-based Solution to the Software-Selection and Hardware-Mapping-Problem at the TTC 2018.” In: *TTC@ STAF*. 2018, pp. 31–36.

[18] Sebastian Götz et al. “Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem.” In: *TTC@ STAF*. 2018, pp. 3–11.

[19] Görel Hedin. “Reference attributed grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[20] Tad Hogg and Dmitriy Portnov. “Quantum optimization”. In: *Information Sciences* 128.3-4 (2000), pp. 181–197.

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

[22] Frank Hutter et al. “Sequential model-based parameter optimization: An experimental investigation of automated and interactive approaches”. In: *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, 2010, pp. 363–414.

[23] Elizabeth Montero et al. “Are state-of-the-art fine-tuning algorithms able to detect a dummy parameter?” In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2012, pp. 306–315.

[24] Volker Nannen and Agoston E Eiben. “Efficient relevance estimation and value calibration of evolutionary algorithm parameters”. In: *2007 IEEE congress on evolutionary computation*. IEEE. 2007, pp. 103–110.

[25] Martin Pelikan, David E Goldberg, and Fernando G Lobo. “A survey of optimization by building and using probabilistic models”. In: *Computational optimization and applications* 21.1 (2002), pp. 5–20.

[26] Dmytro Pukhkaiev. *Energy-efficient Benchmarking for Energy-efficient Software*. 2016.

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

[28] Ingo Rechenberg. “Evolutionsstrategie—Optimierung technischer Systeme nach Prinzipien der biologischen Information”. In: *Stuttgart-Bad Cannstatt: Friedrich Frommann Verlag* (1973).

[29] Filip Rudzinski. “Finding sets of non-dominated solutions with high spread and well-balanced distribution using generalized strength Pareto evolutionary algorithm”. In: *2015 Conference of the International Fuzzy Systems Association and the European Society for Fuzzy Logic and Technology (IFSA-EUSFLAT-15)*. Atlantis Press. 2015.

[30] Khushro Shahookar and Pinaki Mazumder. “A genetic approach to standard cell placement using meta-genetic parameter optimization”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 9.5 (1990), pp. 500–511.

[31] Moshe Sipper et al. “Investigating the parameter space of evolutionary algorithms”. In: *BioData mining* 11.1 (2018), p. 2.

[32] Selmar K Smit and AE Eiben. “Parameter tuning of evolutionary algorithms: Generalist vs. specialist”. In: *European conference on the applications of evolutionary computation*. Springer. 2010, pp. 542–551.

[33] Selmar K Smit and Agoston E Eiben. “Comparing parameter tuning methods for evolutionary algorithms”. In: *2009 IEEE congress on evolutionary computation*. IEEE. 2009, pp. 399–406.

[34] Selmar Kagiso Smit. *Parameter tuning and scientific testing in evolutionary algorithms*. Vrije Universiteit, 2012.

[35] IM Sobol and Yu L Levitan. “A pseudo-random number generator for personal computers”. In: *Computers & Mathematics with Applications* 37.4-5 (1999), pp. 33–40.

[36] Elvar Theodorsson-Norheim. “Friedman and Quade tests: BASIC computer program to perform nonparametric two-way analysis of variance and multiple comparisons on ranks of several related samples”. In: *Computers in Biology and Medicine* 17.2 (1987), pp. 85–99.

[37] P. A. Vikhar. “Evolutionary algorithms: A critical review and its future prospects”. In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. Dec. 2016, pp. 261–265.

[38] David H Wolpert. “The lack of a priori distinctions between learning algorithms”. In: *Neural computation* 8.7 (1996), pp. 1341–1390.

[39] 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.

[40] Shyue-Jian Wu and Pei-Tse Chow. “Genetic algorithms for nonlinear mixed discrete-integer optimization problems via meta-genetic parameter optimization”. In: *Engineering Optimization+ A35* 24.2 (1995), pp. 137–159.

[41] Li Zhihuan, Li Yinhong, and Duan Xianzhong. “Improved strength pareto evolutionary algorithm with local search strategies for optimal reactive power flow”. In: *Information Technology Journal* 9.4 (2010), pp. 749–757.

[42] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. “SPEA2: Improving the strength Pareto evolutionary algorithm”. In: *TIK-report* 103 (2001).

A. Experiment Description of Genetic Solver for BRISE


```
{
  "DomainDescription":{
    "HyperparameterNames"       : [
      "selectorType",
      "populationSize",
      "lambda",
      "crossoverRate",
      "mu",
      "mutationRate",
      "resourcesMutationProbability",
      "crossoverProbability",
      "evaluatorValidityWeight",
      "evaluatorSoftwareValidityWeight",
      "randomSoftwareAssignmentAttempts",
      "populateSoftwareSolutionAttempts"
    ],
    "DataFile"          : "./Resources/GA/GAExperimentData.json"
  },
  "TaskConfiguration":{
    "TaskName"          : "genetic",
    "Scenario":{
      "numTopLevelComponents": 1,
      "avgNumImplSubComponents": 0,
      "implSubComponentStdDerivation": 0,
      "avgNumCompSubComponents": 2,
      "compSubComponentStdDerivation": 0,
      "componentDepth": 2,
      "numImplementations": 10,
      "excessComputeResourceRatio": 5,
      "numRequests": 15,
      "numCpus": 1,
      "seed": 0,
      "timeoutValue": 1,
      "timeoutUnit": "MINUTES",
```

```
      "generations": 1000000000
    },
    "TaskParameters"   : [
      "selectorType",
      "populationSize",
      "lambda",
      "crossoverRate",
      "mu",
      "mutationRate",
      "resourcesMutationProbability",
      "crossoverProbability",
      "evaluatorValidityWeight",
      "evaluatorSoftwareValidityWeight",
      "randomSoftwareAssignmentAttempts",
      "populateSoftwareSolutionAttempts"
    ],
    "ResultStructure"  : ["Validity", "Generation"],
    "ResultDataTypes"  : ["int", "int"],
    "ExpectedValuesRange": [[0, "inf"], [-2, "inf"]],
    "RepeaterDecisionFunction"  : "student_deviation",
    "MaxTasksPerConfiguration": 6,
    "MaxTimeToRunTask": 1800
  }
}
```

B. MQuAT Problem Evaluation Set

Table B.1. Problem solving status by specific solver

Problem Id	Software variants	Number of requests	Component tree depth	Resources ratio	B	B-T	WHC-T	NP	NP-T	ICCR	ICCR-T	WD	WD-T
1	2	1	2	50	■	■	■	■	■	■	■	■	■
2	2	1	3	50	□	□	■	□	■	□	□	■	■
3	2	1	4	50	□	□	□	□	■	□	□	■	□
4	2	2	2	50	□	■	■	■	■	■	■	■	■
5	2	2	3	50	□	□	□	□	■	□	■	□	□
6	2	2	4	50	□	□	□	□	□	□	□	□	□
7	2	4	2	50	□	■	■	■	■	□	■	■	■
8	2	4	3	50	□	□	□	□	□	□	□	□	□
9	2	4	4	50	□	□	□	□	□	□	□	□	□
10	2	1	2	100	■	■	■	■	■	■	■	■	■
11	2	1	3	100	□	□	■	■	■	■	■	■	■
12	2	1	4	100	□	□	□	□	■	□	■	■	□
13	2	2	2	100	■	■	■	■	■	■	■	■	■
14	2	2	3	100	□	□	□	□	■	□	■	□	□
15	2	2	4	100	□	□	□	□	□	□	□	□	□
16	2	4	2	100	□	■	■	■	■	□	■	□	■
17	2	4	3	100	□	□	□	□	□	□	□	□	□
18	2	4	4	100	□	□	□	□	□	□	□	□	□
19	2	1	2	50	■	■	■	■	■	■	■	■	■
20	4	1	3	50	□	□	■	■	■	■	■	■	■
21	4	1	4	50	□	□	□	□	■	□	■	□	□
22	4	2	2	50	■	■	■	□	■	■	■	■	■
23	4	2	3	50	□	□	■	■	■	□	■	■	□
24	4	2	4	50	□	□	□	□	□	□	□	□	□
25	4	4	2	50	□	□	■	■	■	□	■	□	■
26	4	4	3	50	□	□	□	□	□	□	□	□	□
27	4	4	4	50	□	□	□	□	□	□	□	□	□
28	4	1	2	100	■	■	■	■	■	■	■	■	■
29	4	1	3	100	□	■	■	■	■	■	■	■	■
30	4	1	4	100	□	□	□	□	□	□	□	□	□
31	4	2	2	100	■	■	■	■	■	■	■	■	■
32	4	2	3	100	□	□	■	■	■	□	■	■	□
33	4	2	4	100	□	□	□	□	□	□	□	□	□
34	4	4	2	100	□	■	■	□	■	□	■	■	■
35	4	4	3	100	□	□	□	□	□	□	□	□	□
36	4	4	4	100	□	□	□	□	□	□	□	□	□

■: solved, □: unsolved

C. Parameter Pairs Distributions

More plots on `https://github.com/RomanKosovnenko/master_thesis/tree/master/images/PairsDistr`

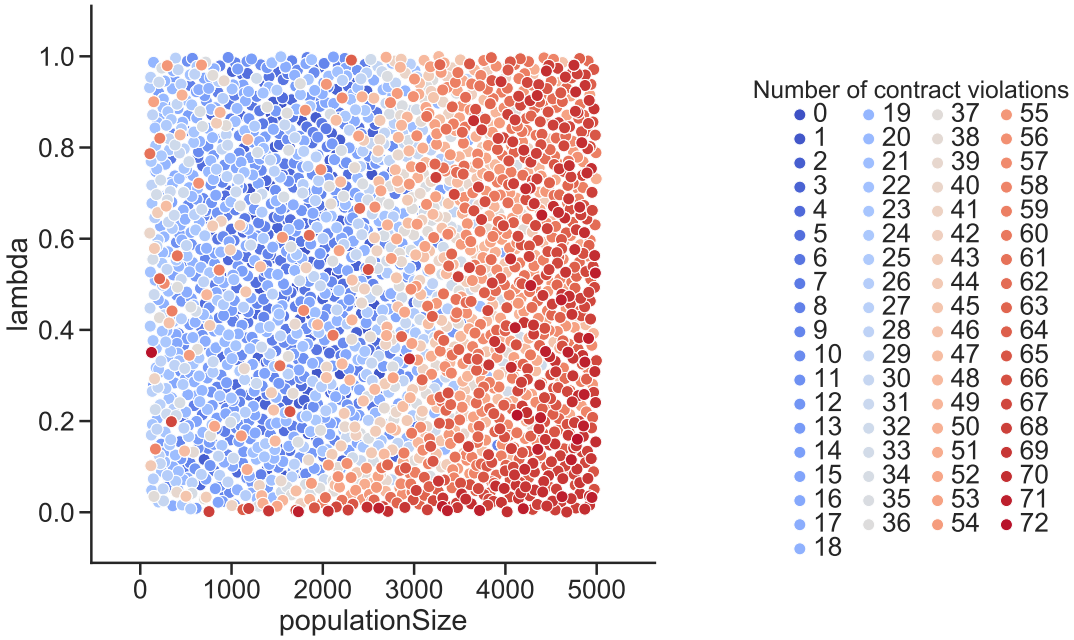


Figure C.1. populationSize and lambda pairs distribution

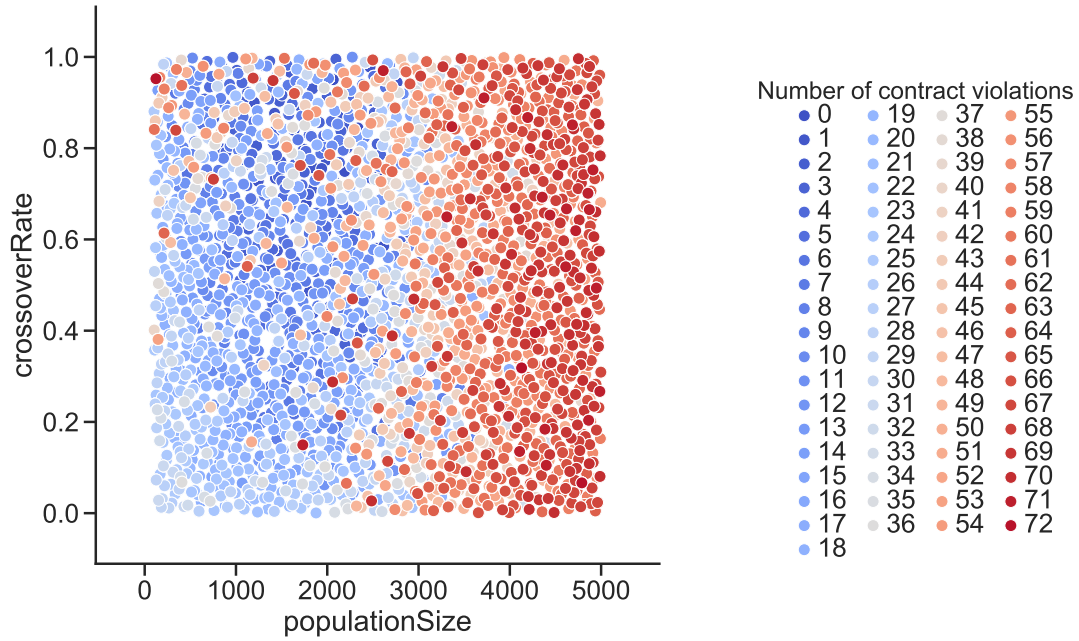


Figure C.2. populationSize and crossoverRate pairs distribution

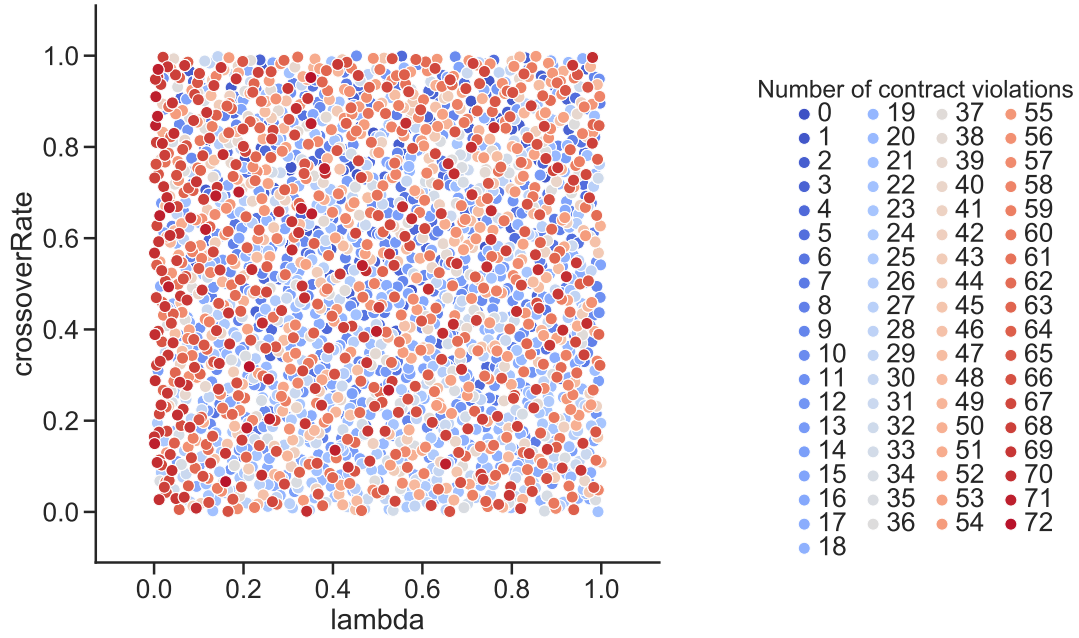


Figure C.3. lambda and crossoverRate pairs distribution

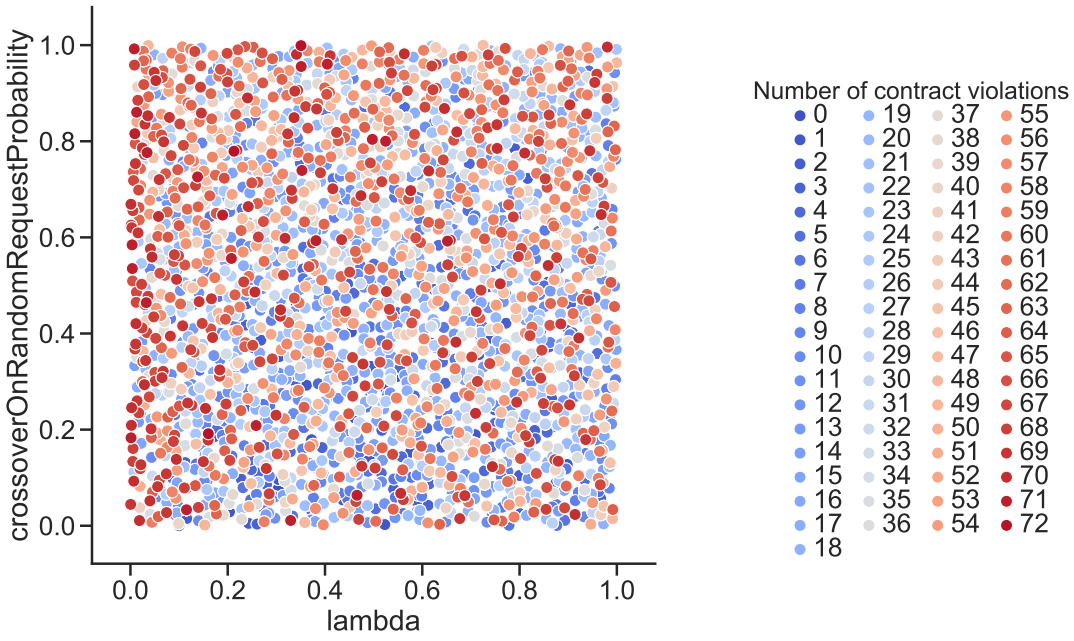


Figure C.4. lambda and crossoverOnRandomRequestProbability pairs distribution

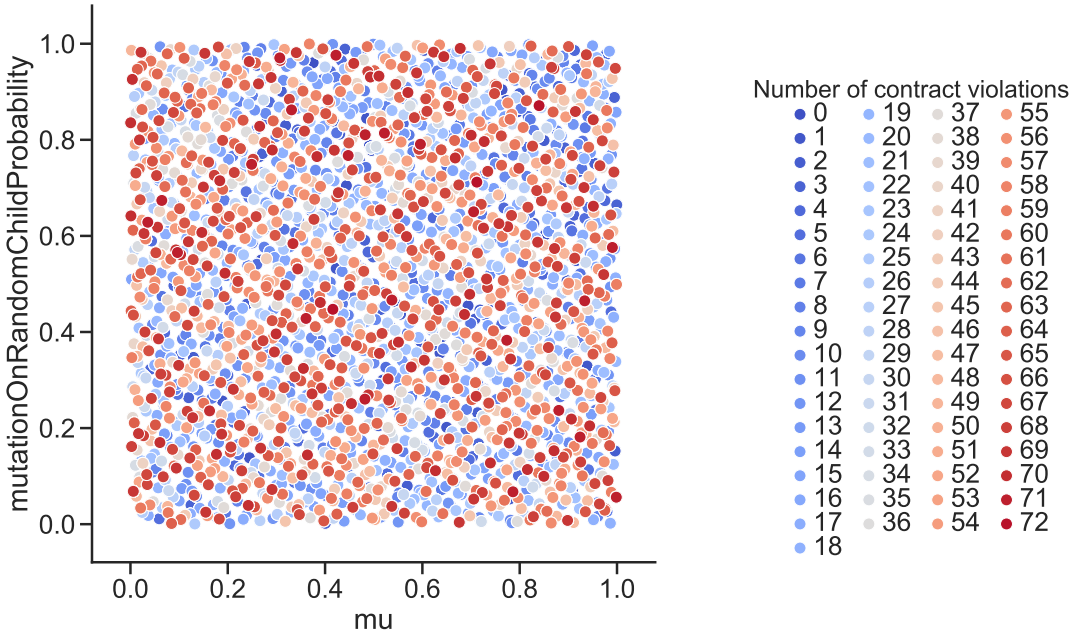


Figure C.5. mu and mutationOnRandomChildProbability pairs distribution

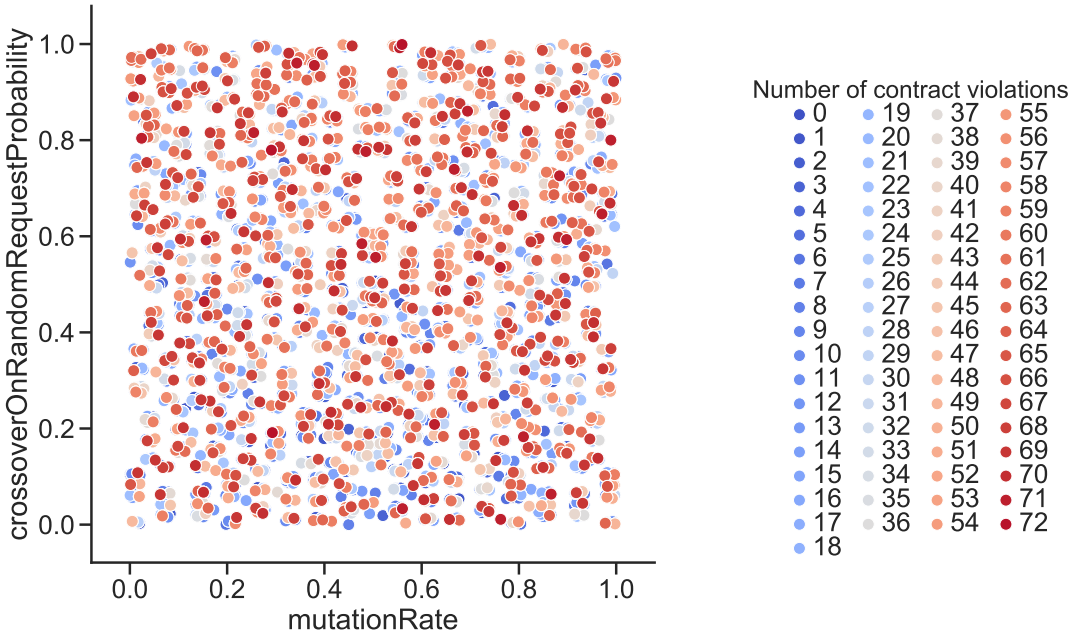


Figure C.6. mutationRate and crossoverOnRandomRequestProbability pairs distribution

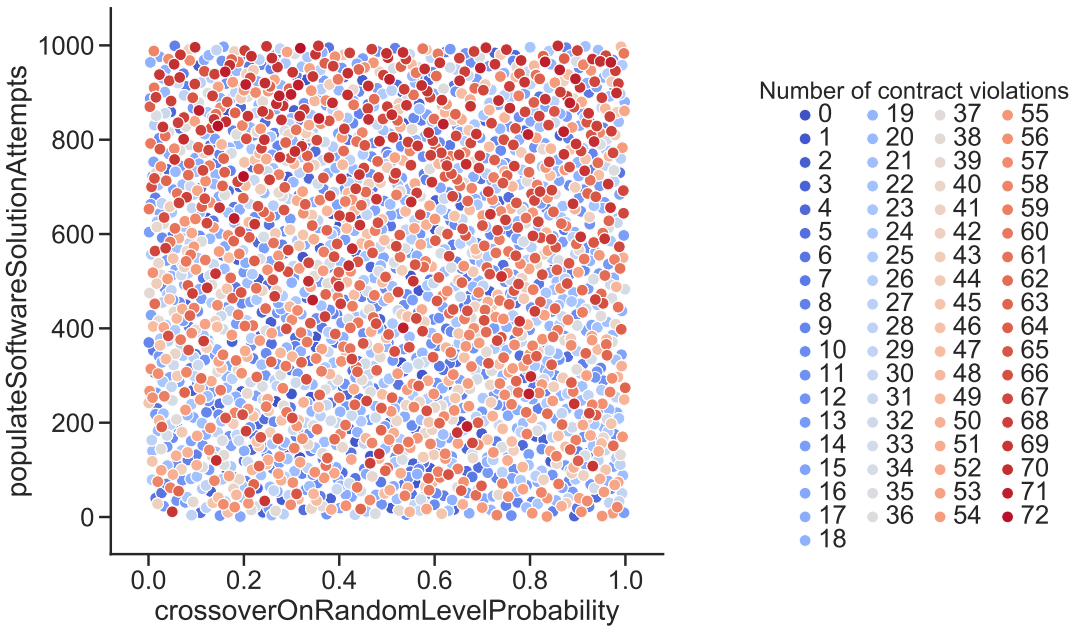


Figure C.7. crossoverOnRandomLevelProbability and populateSoftwareSolutionAttempts pairs distribution

D. Parameter Values Distributions in Terms of Contract Violations

More plots on `https://github.com/RomanKosovnenko/master_thesis/tree/master/images/DistrValidityBig` and `https://github.com/RomanKosovnenko/master_thesis/tree/master/images/DistrValiditySmall`

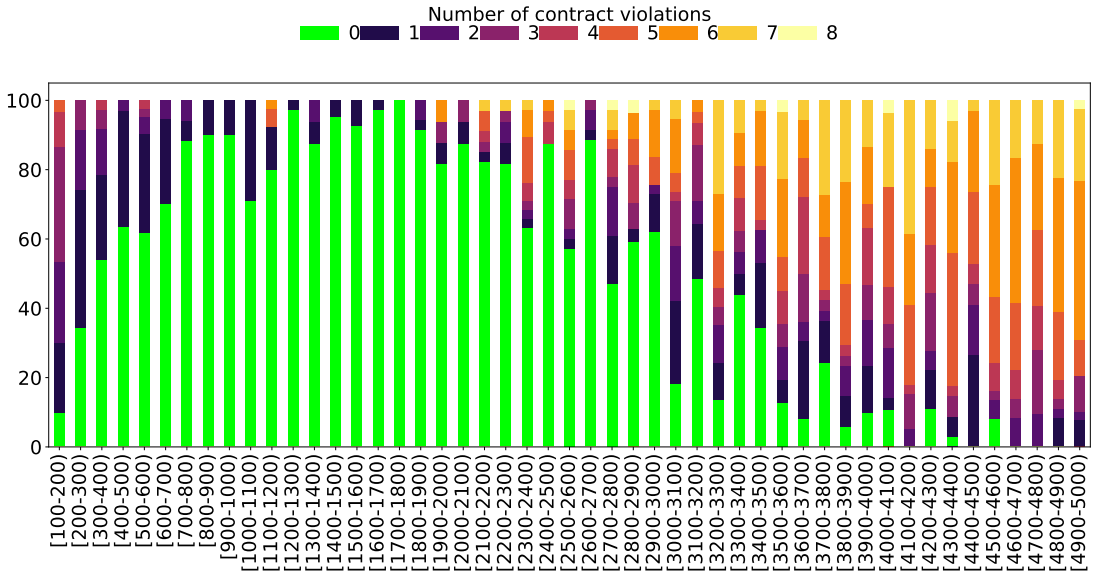


Figure D.1. populationSize parameter values distribution for smaller problem

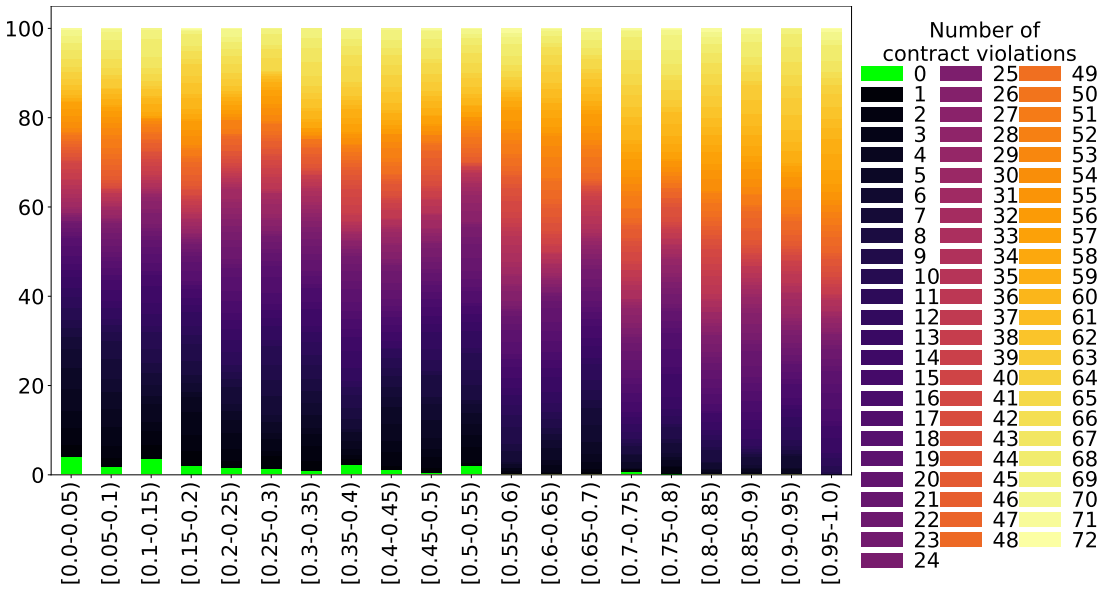


Figure D.2. crossoverOnRandomRequestProbability parameter values distribution for bigger problem

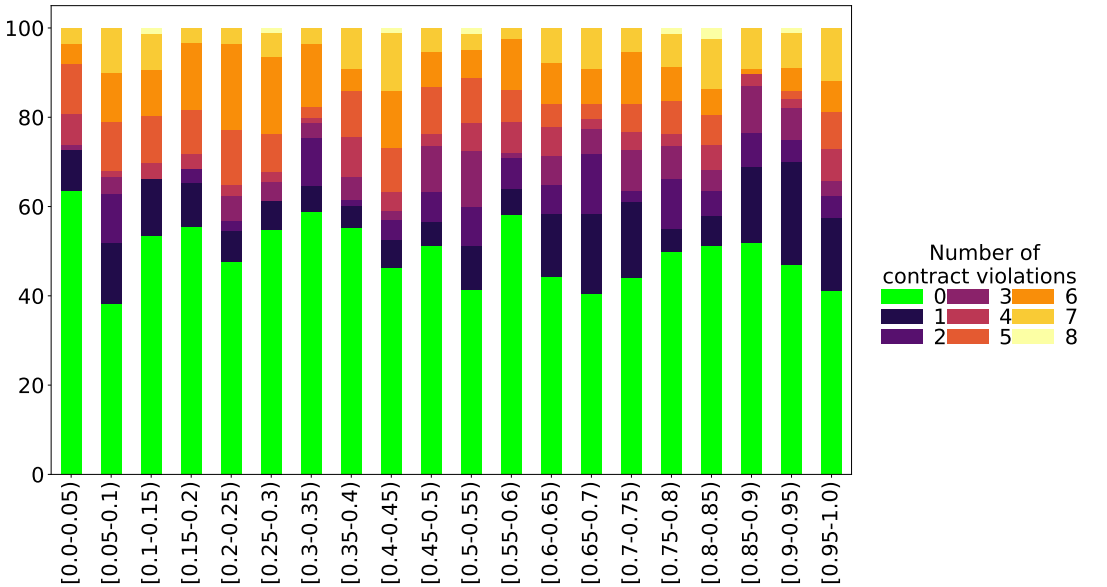


Figure D.3. crossoverOnRandomRequestProbability parameter values distribution for smaller problem

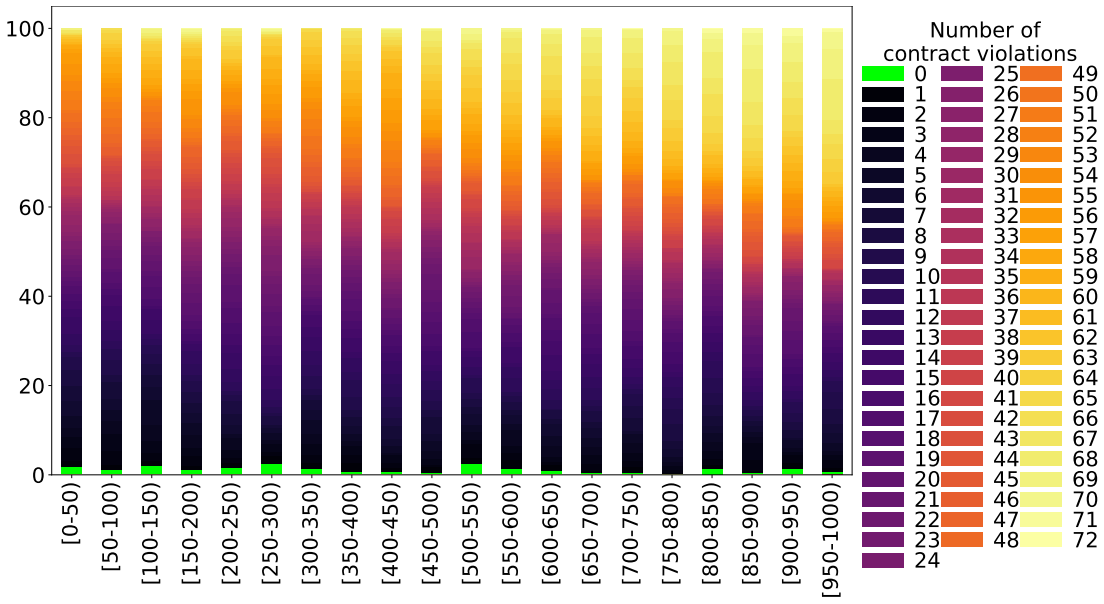


Figure D.4. populateSoftwareSolutionAttempts parameter values distribution for bigger problem

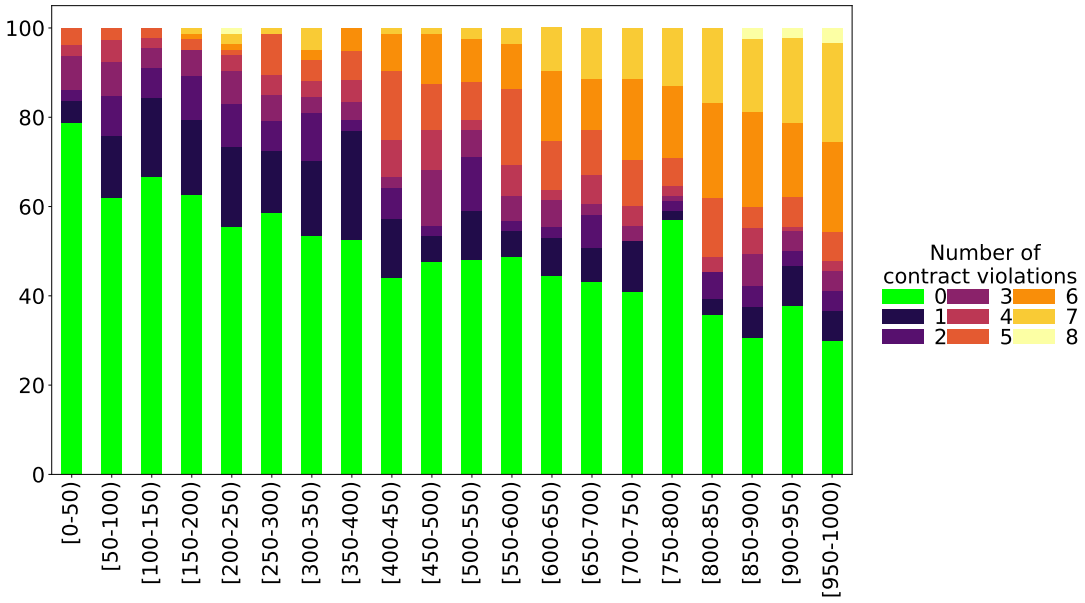


Figure D.5. populateSoftwareSolutionAttempts parameter values distribution for smaller problem

E. Parameter Values Distributions in Terms of Quality

More plots on `https://github.com/RomanKosovnenko/master_thesis/tree/master/images/DistrObj`

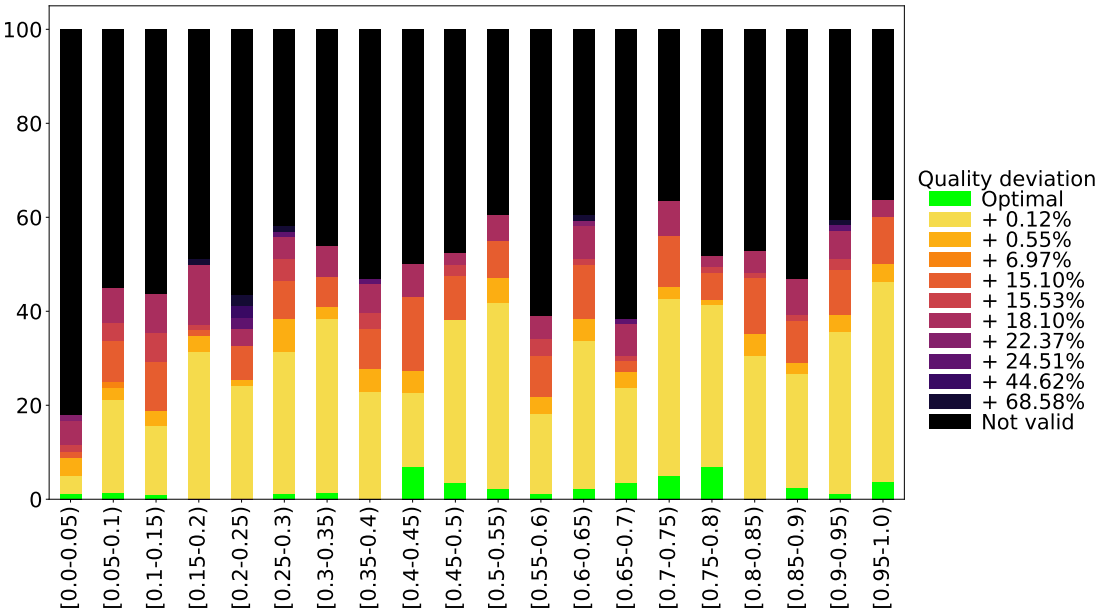


Figure E.1. lambda parameter values distribution for smaller problem

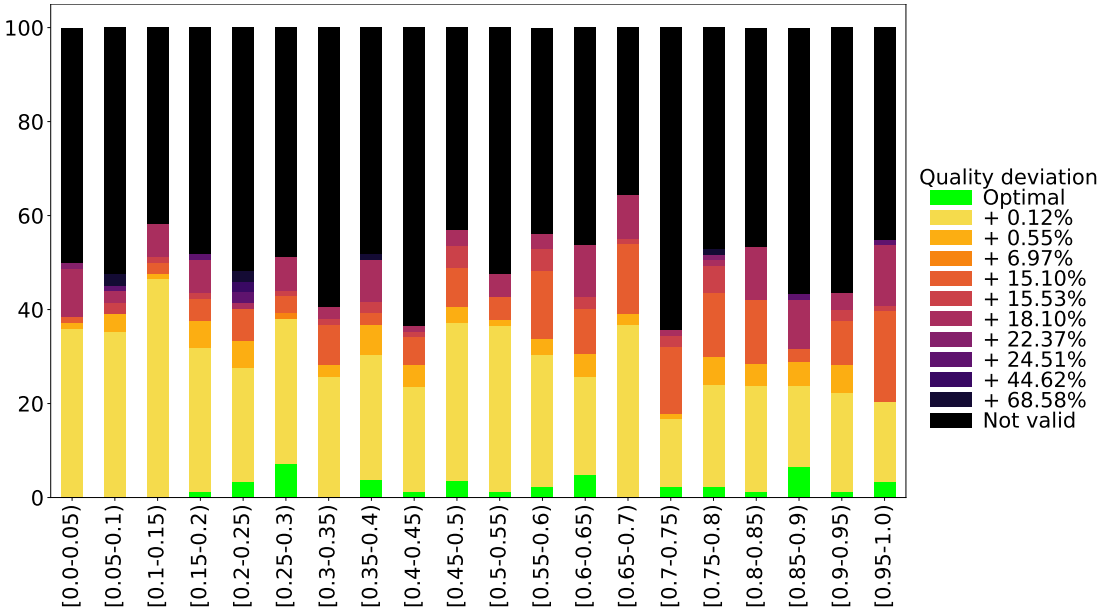


Figure E.2. crossoverRate parameter values distribution for smaller problem

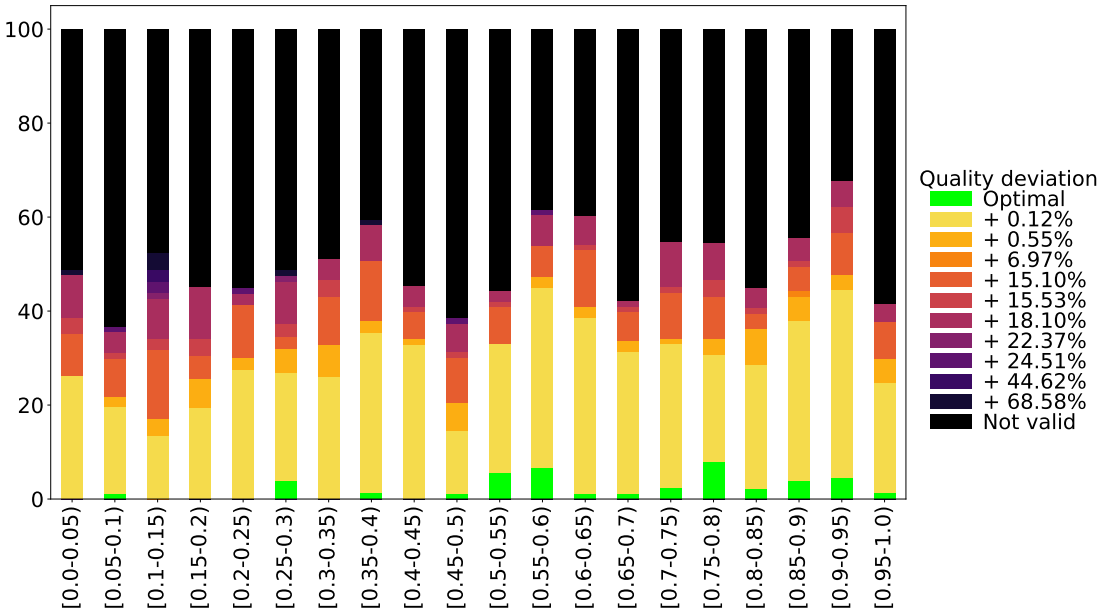


Figure E.3. resourcesMutationProbability parameter values distribution for smaller problem

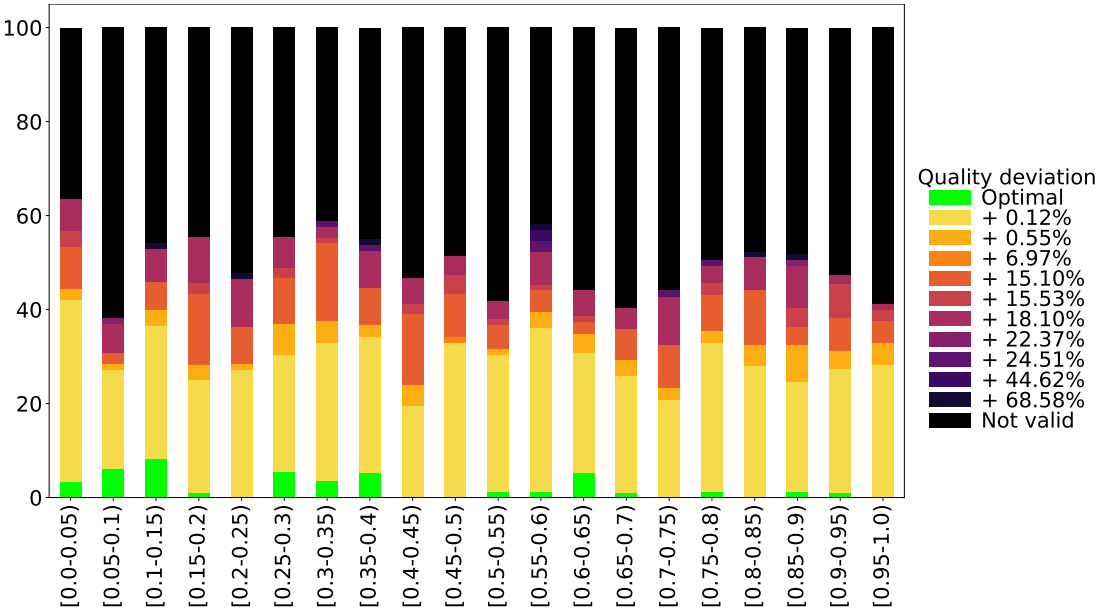


Figure E.4. crossoverOnRandomRequestProbability parameter values distribution for smaller problem

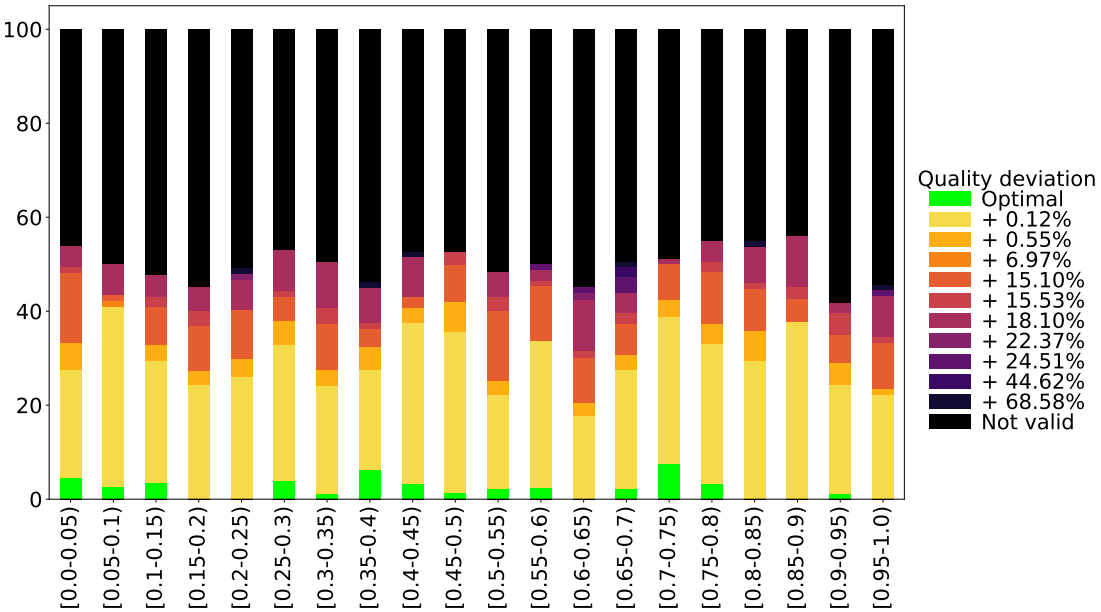


Figure E.5. evaluatorValidityWeight parameter values distribution for smaller problem

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *Systematic Parameter Tuning of Genetic Optimization Approaches* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 9th March 2020

Roman Kosovnenko