A BILEVEL APPROACH TO PARAMETER TUNING OF OPTIMIZATION ALGORITHMS USING EVOLUTIONARY COMPUTING

Understanding optimization algorithms through optimization

MARTIN ANDERSSON
Informatics
A BILEVEL APPROACH TO PARAMETER TUNING OF OPTIMIZATION ALGORITHMS USING EVOLUTIONARY COMPUTING

Understanding optimization algorithms through optimization
DOCTORAL DISSERTATION

A BILEVEL APPROACH TO PARAMETER TUNING OF OPTIMIZATION ALGORITHMS USING EVOLUTIONARY COMPUTING

Understanding optimization algorithms through optimization

MARTIN ANDERSSON
Informatics

UNIVERSITY OF SKÖVDE
Title: A Bilevel Approach to Parameter Tuning of Optimization Algorithms Using Evolutionary Computing
Understanding optimization algorithms through optimization

University of Skövde, Sweden
www.his.se

Printer: BrandFactory AB, Göteborg

ISBN 978-91-984187-7-4
Dissertation Series No. 25 (2018)
ABSTRACT

Most optimization problems found in the real world cannot be solved using analytical methods. For these types of difficult optimization problems, an alternative approach is needed. Metaheuristics are a category of optimization algorithms that do not guarantee that an optimal solution will be found, but instead search for the best solutions using some general heuristics. Metaheuristics have been shown to be effective at finding “good-enough” solutions to a wide variety of difficult problems. Most metaheuristics involve control parameters that can be used to modify how the heuristics perform its search. This is necessary because different problems may require different search strategies to be solved effectively. The control parameters allow for the optimization algorithm to be adapted to the problem at hand. It is, however, difficult to predict what the optimal control parameters are for any given problem. The problem of finding these optimal control parameter values is known as parameter tuning and is the main topic of this thesis.

This thesis uses a bilevel optimization approach to solve parameter tuning problems. In this approach, the parameter tuning problem itself is formulated as an optimization problem and solved with an optimization algorithm. The parameter tuning problem formulated as a bilevel optimization problem is challenging because of nonlinear objective functions, interacting variables, multiple local optima, and noise. However, it is in precisely this kind of difficult optimization problem that evolutionary algorithms, which are a subclass of metaheuristics, have been shown to be effective. That is the motivation for using evolutionary algorithms for the upper-level optimization (i.e. tuning algorithm) of the bilevel optimization approach. Solving the parameter tuning problem using a bilevel optimization approach is also computationally expensive, since a complete optimization run has to be completed for every evaluation of a set of control parameter values. It is therefore important that the tuning algorithm be as efficient as possible, so that the parameter tuning problem can be solved to a satisfactory level with relatively few evaluations. Even so, bilevel optimization experiments can take a long time to run on a single computer. There is, however, considerable parallelization potential in the bilevel optimization approach, since many of the optimizations are independent of one another. This thesis has three primary aims: first, to present a bilevel optimization framework and software architecture for parallel parameter tuning; second, to use this framework and software architecture to evaluate and configure evolutionary algorithms as tuners and compare them with other parameter tuning methods; and, finally, to use parameter tuning experiments to gain new insights into and understanding of how optimization algorithms work and how they can be used be to their maximum potential.

The proposed framework and software architecture have been implemented and deployed in more than one hundred computers running many thousands of parameter tuning experiments for many millions of optimizations. This illustrates that this design and implementation approach can handle large parameter tuning experiments. Two types of evolutionary algorithms, i.e. differential evolution (DE) and a genetic algorithm (GA), have been evaluated as tuners against the parameter tuning algorithm irace. The as-
pects of algorithm configuration and noise handling for DE and the GA as related to the parameter tuning problem were also investigated. The results indicate that dynamic resampling strategies outperform static resampling strategies. It was also shown that the GA needs an explicit exploration and exploitation strategy in order not become stuck in local optima. The comparison with irace shows that both DE and the GA can significantly outperform it in a variety of different tuning problems.
SAMMANFATTNING

Praktiska optimeringsproblem är ofta så svåra att dom inte kan angripas med traditionella metoder som går ut på att hitta den optimala lösningen, utan istället måste alternativa metoder användas som effektivt kan hitta tillräckligt bra lösningar. En sådan alternativ metod är metaheuristiska optimeringsalgoritmer som har visat sig vara kraftfulla när det gäller att hitta bra, med nödvändigtvis inte optimala, lösningar för olika typer av svåra optimeringsproblem. Metaheuristiska optimeringsalgoritmer har också fördelen att de flexibelt kan anpassa sin sökstrategi för en mängd olika optimeringsproblem genom parametrar som påverkar hur sökningen utförs. Inställningarna för dessa parametrar har stor påverkan på hur väl algoritmen lyckas, men svårigheten är att det är nästan omöjligt att på förhand veta vilka inställningar som är de bästa för ett visst optimeringsproblem. Att automatiskt kunna hitta de bästa parameterinställningarna är därför önskvärt och det är denna utmaning som står i fokus för avhandlingen.

ACKNOWLEDGEMENTS

This thesis concludes a journey that was long and arduous, but also very fun and rewarding. There were plenty of challenges and problems along the way. I spent many long days working toward deadlines of papers or trying to find obscure software bugs. However, solving problems and overcoming difficulties are things I truly enjoy doing. Although I am proud of what I have accomplished, I could not have done it without the support of so many great people. First and foremost, I would like to thank my supervisors, Prof. Amos Ng, Dr. Sunith Bandaru, and Prof. Kalyanmoy Deb, for their invaluable expertise and guidance. A special thanks also goes to Prof. Anna Syberfeldt and Dr. Mats Jägstad for helping me start this journey; without them I would not be where I am today. I am also grateful for the many discussions and memorable moments that I had with Ingemar Karlsson, Dr. Jacob Bernedixen, Dr. Marcus Frantzén, Dr. Tehseen Aslam, and the rest of my department—your feedback was tremendously helpful to me. Finally, I would like to thank my family for their unwavering encouragement and support along this entire journey. Thank you all.
PUBLICATIONS

During my PhD studies I have written the following papers.

PUBLICATIONS WITH HIGH RELEVANCE


PUBLICATIONS WITH LOW RELEVANCE

CONTENTS

1 Introduction 3
  1.1 Motivation ................................................................. 3
  1.2 Thesis Aim and Research Questions ........................................ 4
  1.3 Summary of the Included Papers ........................................... 4
    1.3.1 Paper I (Published) .................................................. 5
    1.3.2 Paper II (Published) .................................................. 5
    1.3.3 Paper III (Published) ................................................. 6
    1.3.4 Paper IV (Published) .................................................. 6
    1.3.5 Paper V (Under review) .............................................. 7
    1.3.6 Paper VI (Submitted) .................................................. 7
    1.3.7 Paper VII (Completed manuscript) ................................... 8
  1.4 Outline of the Thesis .................................................... 8

2 Frame of Reference 11
  2.1 Optimization Algorithms ................................................ 11
  2.2 Multi-Objective Optimization .......................................... 12
  2.3 Performance Indicators ............................................... 13
  2.4 Configuration of Optimization Algorithms .......................... 14
  2.5 Generalist and Specialist CPV Tuples and the No Free Lunch Theorem .... 15
  2.6 Simulation-Based Optimization ....................................... 16
  2.7 A Scalable Simulation Model ......................................... 17
  2.8 Parallel Metaheuristics ............................................... 18
  2.9 Parallel Evolutionary Algorithms .................................... 19

3 Research Methodology 23
  3.1 Design Science Guidelines ............................................. 24
  3.2 Empirical Research ..................................................... 24
    3.2.1 Experimental Research .............................................. 25
    3.2.2 Empirical versus Theoretical Approaches .......................... 26
    3.2.3 Noise and Hypothesis Testing ................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4 Internal and External Validity</td>
<td>27</td>
</tr>
<tr>
<td>4 Framework and Software Architecture for Parameter Tuning</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Existing Studies of Parameter Tuning</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Framework</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1 Upper Level Performance Indicators</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Evaluation of a CPV Tuple</td>
<td>36</td>
</tr>
<tr>
<td>4.2.3 Parallelization Potential</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Software Architecture</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1 Optimization Types</td>
<td>39</td>
</tr>
<tr>
<td>4.3.2 Implementation and Software Libraries</td>
<td>39</td>
</tr>
<tr>
<td>4.3.3 Information Flow</td>
<td>40</td>
</tr>
<tr>
<td>4.3.4 Controller</td>
<td>40</td>
</tr>
<tr>
<td>4.3.5 Storage</td>
<td>40</td>
</tr>
<tr>
<td>4.3.6 Optimization Manager</td>
<td>40</td>
</tr>
<tr>
<td>4.3.7 Worker Manager</td>
<td>41</td>
</tr>
<tr>
<td>4.3.8 Proxy Manager</td>
<td>41</td>
</tr>
<tr>
<td>4.3.9 Compute</td>
<td>42</td>
</tr>
<tr>
<td>5 Evolutionary Algorithms as Parameter Tuners</td>
<td>45</td>
</tr>
<tr>
<td>5.1 Genetic Algorithm</td>
<td>46</td>
</tr>
<tr>
<td>5.2 Differential Evolution</td>
<td>46</td>
</tr>
<tr>
<td>5.3 Noise Handling in Evolutionary Algorithms</td>
<td>47</td>
</tr>
<tr>
<td>5.4 Experimental Design</td>
<td>47</td>
</tr>
<tr>
<td>5.5 Results</td>
<td>50</td>
</tr>
<tr>
<td>6 Using Parameter Tuning to Improve Optimization Algorithms</td>
<td>57</td>
</tr>
<tr>
<td>6.1 Specialist Versus Generalist Parameters</td>
<td>57</td>
</tr>
<tr>
<td>6.1.1 Results</td>
<td>58</td>
</tr>
<tr>
<td>6.2 Evaluating Heuristics Using Parameter Tuning</td>
<td>59</td>
</tr>
<tr>
<td>6.2.1 Results</td>
<td>60</td>
</tr>
<tr>
<td>6.3 Multiple Parameter Sets</td>
<td>61</td>
</tr>
<tr>
<td>6.3.1 Results</td>
<td>62</td>
</tr>
<tr>
<td>6.4 The Trade-Off Between Runtime and Evaluation Efficiency</td>
<td>63</td>
</tr>
<tr>
<td>6.4.1 Results</td>
<td>64</td>
</tr>
<tr>
<td>6.5 The Trade-Off Between Runtime and Cost in a Cloud Computing Environment</td>
<td>65</td>
</tr>
<tr>
<td>6.5.1 Results</td>
<td>68</td>
</tr>
</tbody>
</table>
7 Conclusions and Future Research

7.1 Contributions to Knowledge .................................................. 74
7.2 Future Research ............................................................... 75
  7.2.1 Software Architecture and Implementation ....................... 75
  7.2.2 Evolutionary Algorithms as Parameter Tuners ................. 76
  7.2.3 Applications of Parameter Tuning ................................. 76

Included Papers

Paper I ................................................................. 81
Paper II ................................................................. 91
Paper III ............................................................. 108
Paper IV .............................................................. 119
Paper V ............................................................... 129
Paper VI ............................................................. 161
Paper VII .......................................................... 183

References ............................................................. 201

Publications in the Dissertation Series ................................ 209
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fitness landscape.</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Pareto-optimal points.</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Parallelization levels (Talbi, 2009).</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Design science framework (Hevner et al., 2004).</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Parameter tuning framework.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Software architecture.</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>The performance of each upper-level configuration when tuning DE on the Ack-</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>ley test function.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>The performance of each upper-level configuration when tuning the GA on the</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Ellipse test function.</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>The performance of each upper-level configuration when tuning the GA as a</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>generalist.</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>The performance of each upper-level configuration when tuning DE as a</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>generalist.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Applicability of $p_m = \frac{1}{N}$ rule of thumb for $p_m$ on ZDT1.</td>
<td>60</td>
</tr>
<tr>
<td>6.2</td>
<td>Applicability of $p_m = \frac{1}{N}$ rule of thumb for $p_m$ on ZDT4.</td>
<td>60</td>
</tr>
<tr>
<td>6.3</td>
<td>Box-and-whisker plots showing the number of FEs needed to reach target</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>IGD value in 100 optimization runs, for ZDT1, ZDT2, ZDT3, and ZDT4.</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Box-and-whisker plots showing the number of FEs needed to reach the target</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>IGD value in 100 optimization runs, for DTLZ1, DTLZ2, DTLZ3, and DTLZ4.</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Box-and-whisker plots showing the number of FEs needed to reach the target</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>IGD value in 100 optimization runs, for WFG1, WFG2, WFG3, and WFG4.</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>Trade-off fronts with respect to the number of evaluations and generations</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>for DTLZ1, DTLZ2, SIM, and ALL.</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>Parallel coordinate plot of the objectives and parameter settings of all</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>non-dominated CPV tuples for DE on DTLZ1 with 7 and 14 decision variables.</td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>Parallel coordinate plot of the objectives and parameter settings of all</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>non-dominated CPV tuples for NSGA-II on DTLZ2 with 12 and 24 decision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>variables.</td>
<td></td>
</tr>
<tr>
<td>6.9</td>
<td>Non-dominated solutions from parameter tuned NSGA-II and DE.</td>
<td>68</td>
</tr>
</tbody>
</table>
6.10 Non-dominated solutions from cloud experiments. .......................... 69
6.11 Parallel coordinate plot of the objectives and parameter settings of all non-dominated CPV tuples for DE. .......................................................... 70
LIST OF TABLES

1.1 Relationships between research questions and papers. ...................... 5

4.1 Literature overview. ........................................................................... 32

5.1 Problem configurations. ................................................................. 48
5.2 Experimental configuration for the upper-level GA. ......................... 49
5.3 Experimental configuration for the upper-level DE. ......................... 49
5.4 Combined ranking of upper-level tuners. ......................................... 50

6.1 Best, median, and mean results of 20 replications of parameter tuning experiments for specialized parameters (CMAES-S) and generalized parameters (CMAES-G). ................................................................. 58

6.2 The score as calculated by the objective function in Equation (6.1). The specialized parameter score (CMAES-S) is the summation of all individual results. 58
INTRODUCTION
CHAPTER 1
INTRODUCTION

1.1 MOTIVATION

Optimization problems can be found in many different areas, including agriculture, manufacturing, transportation and many more. The solutions to such optimization problems can have large potential benefits, such as reductions in cost, time, and energy usage, or improvements in quality, efficiency, and profits. However, these types of optimization problems are difficult to solve since they often involve factors such as interacting variables, noise (real-world problems usually have some stochastic factors), and multiple nonlinear objective functions with multiple local optima. It is impossible to solve such optimization problems exactly. Instead, a type of optimization algorithm called a metaheuristic is used to search for the best solutions. There is, however, no guarantee that such algorithms will find the optimal solution. Evolutionary algorithms (EAs) are a kind of metaheuristic that have been successfully used to find good-enough solutions to these types of difficult problems. EAs are inspired by evolutionary processes observed in nature, in particular natural selection and reproduction.

The heuristics are what make this type of optimization algorithm so effective, since they evaluate only a small fraction of the vast search spaces. However, the heuristics are not always equally effective for all problems. That is why an optimization algorithm can perform very differently from one problem to the next. For example, a greedy algorithm can perform very well on a simple problem, but fail miserably on a more difficult one. Alternatively, it could search broadly and be successful on a multi-modal problem, but be inefficient for a problem with a flat fitness landscape. In other words, there is no optimization algorithm that is better than all other optimization algorithms on all problems; this is known as the no free lunch theorem (Wolpert and Macready, 1997). Since an optimization algorithm cannot be equally efficient across all problems, most such algorithms extract important design decisions as control parameters. These control parameters, which are often kept constant, can be used to change the search strategy of the algorithm so that it can be adapted to the problem at hand. It is well known (Bäck, 1994; Wessing et al., 2010) that these settings can greatly affect the search process and the overall performance of the algorithm. However, how to set them for a particular problem is not always intuitive. A strategy often used is to choose parameter values that have been shown to be effective for similar problems. Selecting the optimal parameter configuration for a given algorithm and problem combination is difficult without a-priori knowledge, and this is known as the parameter tuning problem.

There are many different approaches to tackling the parameter tuning problem. Manually constructing and evaluating control parameter values in an ad hoc manner is only feasible when a small number of them are being tested. Another approach is to formulate the parameter tuning problem as an optimization problem and solve it with an optimization algorithm. This is a challenging optimization problem because of nonlinear objective functions, interacting variables, multiple local optima, and noise. However, it is on exactly this kind of difficult optimization problem that evolutionary algorithms have
been shown to be effective (Eiben and Smit, 2011). This can be referred to as the meta-EA approach (Eiben and Smit, 2011), which is essentially a bilevel optimization (Sinha et al., 2014) problem. Parameter tuning is not only restricted to tuning the parameters of existing algorithms, but can also be applied during algorithm development. In fact, all internal algorithmic decisions can be extracted as parameters and subjected to tuning.

Since the parameter tuning problem can be both difficult and computationally expensive, it is important to have effective and efficient methods and algorithms with which to solve it. This will enable more difficult parameter tuning problems to be solved, which will help to improve our knowledge of optimization algorithms and the effects of their control parameters. The benefit to optimization practitioners is a better understanding of how to get the most out of their optimization algorithms. This will in turn help them solve more difficult optimization problems and obtain their optimization results faster.

1.2 THESIS AIM AND RESEARCH QUESTIONS

The aim of this thesis is to design and implement a bilevel optimization approach that could discover new knowledge of the efficiency of optimization algorithms through parameter tuning. This aim will be accomplished by answering the following research questions:

• **RQ1.** How can an effective, distributed, and parallel computing software architecture be designed and implemented for running bilevel optimization experiments?

• **RQ2.** How should evolutionary algorithms be configured in order to increase their effectiveness at solving parameter tuning problems?

• **RQ3.** How can bilevel optimization be used to gain new knowledge of optimization algorithms and the influences of their control parameters?

The motivation for RQ1 is the fact that the bilevel optimization approach to parameter tuning is computationally expensive. Hence, there is a need for an effective, distributed, and parallel architecture that can handle large-scale parameter tuning experiments. By definition, this approach requires an optimization algorithm on the upper level that solves the parameter tuning problem. The motivation for RQ2 is that a poorly performing optimization algorithm on the upper level will limit the usefulness of the approach. The purpose of RQ3 is to illustrate how this approach can be used to gain new knowledge of optimization algorithms. The aim of this research is therefore not only to design a bilevel optimization approach that can be used to effectively tune the control parameters of optimization algorithms, but also to gain useful knowledge of these algorithms and of the influences of their control parameters, through the bilevel parameter tuning process.

1.3 SUMMARY OF THE INCLUDED PAPERS

This section will present the main contributions of each paper included in this thesis. The relationships between the papers and the research questions are shown in Table 1.1.
Table 1.1: Relationships between research questions and papers.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RQ2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RQ3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1.3.1 PAPER I (PUBLISHED)

©2015 IEEE. Reprinted, with permission, from:


This paper demonstrated the significant performance gains that can be obtained by tuning the parameters of an optimization algorithm. It used the bilevel optimization approach to tune the CMA-ES algorithm applied to the single-objective CEC’15 expensive problems. Both specialist and generalist parameter configurations were tuned. The combined median performance of the specialists was 81% better than that of the generalist. These results illustrate how large the performance difference can be between specialists and generalists. Also, the parameter configuration of the generalist was different from all those of the specialists. In other words, the generalist parameter values was not simply some of the specialist parameter values. The tuned parameter values were also compared with the recommended parameter configuration of CMA-ES. As expected, both the specialist and generalist configurations outperformed the recommended parameter configuration.

1.3.2 PAPER II (PUBLISHED)


This paper first used the bilevel optimization approach to tune multi-objective algorithms. It demonstrated that parameter tuning can be used both to test and to further improve knowledge of heuristics for setting control parameters. The parameters of NSGA-II and NSGA-III were tuned on the two objective ZDT test problems. The function evaluation budget and number of decision variables were varied to study their effects on the optimal parameter configurations. The results indicate that the well known heuristic for setting the mutation probability, $\frac{1}{\sqrt{N}}$, in which $N$ is the number of decision variables, is a good heuristic for NSGA-II and NSGA-III on the ZDT problems. It was also demonstrated that mutation probability decreased as the budget increased. Finally, a comparison of the performance of the NSGA-II and NSGA-III algorithms was conducted. The difference between the algorithms was small, but NSGA-II had a slight advantage on ZDT problems.
1.3.3 PAPER III (PUBLISHED)


It has been shown to be suboptimal to keep the control parameters of a metaheuristic static during the course of the optimization. However, coming up with good control mechanisms is difficult because there can be complex interactions between the different parameters. This paper explored the idea of approximating the optimal adaption of the parameter values using multiple parameter sets. By tuning these parameter sets, the optimization can use different parameter configurations in different phases of the optimization. For example, the beginning of the search can use parameters that promote exploration; later, at the end of the optimization, the search can switch to parameters that are instead good at exploitation.

The tuning was performed using the NSGA-II parameters for the ZDT, DTLZ, and WFG test problems, using one, two, and three parameter sets. The results indicate that the performance can be significantly improved by using two parameter sets. Only two problems, WFG2 and WFG4, saw a significant improvement using three parameter sets. This indicates that most of the performance benefit of adapting parameters can be gained with only two parameters sets. However, not all problems display an advantage with multiple parameter sets, indicating that it will be difficult to develop a parameter control mechanism that works equally well over all problems. However, for the problems that displayed an improvement with two parameter sets, the first set tended to have parameter configurations that promoted exploration, such as high population sizes and high mutation probabilities, whereas the second set had parameter configurations that favored exploitation.

1.3.4 PAPER IV (PUBLISHED)

©2016 IEEE. Reprinted, with permission, from:


This paper took a step towards optimal parameter configurations for simulation-based optimizations. Instead of applying parameter tuning directly to real-world problems, a scalable discrete-event simulation (DES) model was used as an intermediate step. The reasoning was that it can be challenging to perform parameter tuning experiments on real-world problems, because of their relatively long evaluation times. That is why NSGA-II was tuned on a simple but realistic DES model over an unpaced flow line. The effects of different function evaluation budgets and decision variables were studied using both a bilevel optimization approach and the non-iterative Latin hypercube sampling (LHS) method. The results indicate that the bilevel optimization found better parameter configurations, with the difference between methods growing with the number of decision variables (N). The best parameter configuration was found to be better performing (in terms of hypervolume) than the recommended parameter configuration.

The well known heuristic for setting the mutation probability, $\frac{1}{N}$, was also evaluated for this problem. The results indicate that the mutation probability does decrease as
the number of decision variables increases. However, the heuristic underestimates the optimal mutation probability by about a factor of two.

1.3.5 PAPER V (UNDER REVIEW)


The performance of an optimization algorithm is usually measured in terms of the number of function evaluations (FEs) needed to reach a predetermined performance target or alternatively, in terms of the quality of the solution(s) reached after a maximum number of FEs. Here, this performance aspect is referred to as the function evaluation efficiency, or simply the efficiency, of the optimization algorithm. This paper tuned NSGA-II and DE with respect to two objectives, i.e. parallelization potential (controlled by the offspring population size) and function evaluation efficiency, to find the optimal control parameters that provide the best trade-off between these objectives. The motivation for this paper was that optimizations of computationally expensive simulation models can be very time consuming. By better utilizing parallel computing resources, the runtime of such optimizations can be reduced. That is why the EAs were tuned against DTLZ1, DTLZ2, and a simple discrete-event simulation model. In a generational EA, the offspring population size determines the number of solutions that can be evaluated in parallel. Increasing the offspring population size means that more new individuals are created in each generation. However, changing the offspring population will also affect the search strategy of the algorithm, possibly negatively. The trade-off is therefore between a shorter runtime and fewer function evaluations. This paper also demonstrated that the framework can be used to solve two-objective parameter tuning problems.

The results indicate that there is indeed a trade-off between the runtime and the function evaluation efficiency of EAs. DE was able to achieve better trade-off fronts than was NSGA-II mainly because it did not lose as much of its function evaluation efficiency when its offspring population size was increased.

1.3.6 PAPER VI (SUBMITTED)


This paper continues from Paper V, adding steady-state asynchronous versions of NSGA-II and DE, in addition to the generational versions. The focus is still on investigating the trade-off between optimization runtime and the number of function evaluations. The aim of this paper is to evaluate this trade-off by running optimizations in a cloud computing environment. One of the main benefits of cloud computing is that resources can be created on demand and destroyed when no longer needed. In such flexible environments, the amount of available computing resources can be exactly matched to the offspring population size. The trade-off can therefore be rewritten as runtime versus cost. Since a larger offspring population size will decrease the optimization runtime, at least up to a point, it will also increase the cost, because the optimization will use more computing resources caused by a decrease in the function evaluation efficiency.

The results indicate that the steady-state asynchronous versions of both NSGA-II and DE outperformed their generational versions. Even though they evaluated slightly more
function evaluations, that was more than compensated for by their better utilization of computing resources. Also, DE was again able to outperform NSGA-II on the discrete-event simulation optimization problem used in this paper.

1.3.7 PAPER VII (COMPLETED MANUSCRIPT)


This paper presents a framework and software architecture for bilevel optimizations that can be used to solve large-scale parameter tuning problems using hundreds of parallel computing resources. The importance of lower-level optimization replications is investigated by comparing different resampling strategies, including both static and dynamic strategies. Two evolutionary algorithms, DE and a GA, are evaluated as tuners against *irace*, which is a popular parameter tuning method. Different configurations of both EAs are tested in order to determine how to best configure them.

The results of the tuner comparison indicate that both DE and the GA could significantly outperform *irace* on the parameter tuning problems used in this paper. The best number of static optimization replications was observed to differ greatly between parameter tuning problems, indicating that no one resampling strategy is optimal for all parameter tuning problems. However, the dynamic strategies provided a good trade-off between efficient use of the tuning budget and finding robust parameter configurations. It was also observed that the GA benefited from an explicit exploration and exploitation strategy by using different control parameter settings for different stages of the optimization.

1.4 OUTLINE OF THE THESIS

The rest of this thesis is structured as follows. Relevant topics are introduced in Chapter 2. Chapter 3 presents the research methodology and how it was used to guide the work performed in this thesis. The proposed framework and software architecture are presented in Chapter 4. The experimental results of configuring EAs as tuners and how they compare with other parameter tuning methods are presented in Chapter 5. Chapter 6 presents the experimental results of using parameter tuning to improve optimization algorithms. Finally, the conclusions and future research are discussed in Chapter 7.
CHAPTER 2
FRAME OF REFERENCE

This chapter introduces topics relevant to this thesis.

2.1 OPTIMIZATION ALGORITHMS

Mathematical optimization, usually abbreviated simply as optimization, is the process of finding the best solution from a large number of possible alternatives in order to minimize or maximize an objective function. The set of alternatives is defined by the decision variables and the constraints of the problem. The objective function forms what is called a fitness landscape, and the purpose of the optimization is to navigate this landscape to reach the global minimum or maximum, while avoiding local optima. An example of a two-dimensional fitness landscape is shown in Figure 2.1.

![Fitness landscape](image)

Figure 2.1: Fitness landscape.

The solution techniques to solve optimization problems can be classified into two main classes: exact and approximate algorithms.

A problem that is small and well defined can usually be solved with an exact algorithm, meaning that the exact optimal solution to the problem can be found. Unfortunately, most problems in the real world are not of this type. In this context, "well defined" means that the entire problem can be formulated in strictly mathematical terms. For some problems this can be relatively straightforward, but for others, such as those that have stochastic factors, it is almost impossible. Another problem is that the time complexity of many of these exact algorithms is such that it is usually only possible to solve small to medium-sized problem instances. This is, of course, problem and context specific, changing as algorithms improve and computers get faster.
Approximate algorithms can be divided into specific heuristics and metaheuristics. A specific heuristic is a problem-dependent algorithm designed to work for only a small set of problems, while metaheuristics are general algorithms that can be used to solve a variety of optimization problems. In contrast to an exact algorithm, an approximate algorithm can never guarantee that the optimal solution has been found. This is usually not a problem in real-world scenarios in which any solution better than the current one is considered acceptable.

A metaheuristic can be broadly defined as an exploration of a search space using an objective function as a guide. The objective function can be relatively simple, such as evaluating a tour in the traveling salesman problem, or more complex, such as simulating a manufacturing process. This makes it possible to apply metaheuristics to a wide variety of problems. Another important aspect is how a solution is represented, since this will affect which search operators can be applied. Metaheuristics also have some drawbacks. For example, even though they are meant to be general-purpose optimizers, not all metaheuristics work well on all problems (Wolpert and Macready, 1997). Metaheuristics usually also have several parameters that can affect how they perform, and it can be difficult to know how to set them for a given metaheuristic and problem. Examples of metaheuristics are: simulated annealing, tabu search, genetic algorithms, differential evolution, particle swarm optimization, and the Nelder-Mead method. This thesis will consider only metaheuristics and not exact optimization algorithms.

2.2 MULTI-OBJECTIVE OPTIMIZATION

This thesis considers multi-objective optimization problems as described in Equation (2.1), where \( M \) is the number of objectives, \( x \in \mathbb{R}^N \) is a vector of \( N \) decision variables, and \( f \) represents a vector of objective functions that map a solution in the decision space to the objective space as \( f_i : \mathbb{R}^N \to \mathbb{R} \).

\[
\begin{align*}
\text{Minimize} \quad & \{ f_1(x), f_2(x), \ldots, f_M(x) \} \\
\text{Subject to} \quad & g_j(x) \geq 0 \quad \forall \ j \in \{1, 2, \ldots, J\} \\
& h_k(x) = 0 \quad \forall \ k \in \{1, 2, \ldots, K\} \\
& x_l \leq x \leq x_u
\end{align*}
\tag{2.1}
\]

Comparing solutions in a single-objective optimization is trivial, since the value of the objective can be used directly in the comparison. Another approach is needed for optimization problems with multiple conflicting objectives, since a solution can be better for one objective and worse for another. One approach is to convert the multi-objective problem into a single-objective problem by assigning a weight to each objective and using the sum of these modified objectives when comparing solutions. The biggest drawback of this approach is that the weights must be assigned before the optimization starts, and it can be difficult to know what the appropriate weights should be. Instead, the concept of domination can be used to compare solutions in a multi-objective problem. For a minimization problem, a solution, \( u \), is said to dominate another solution, \( v \), if \( f_i(u) \leq f_i(v) \) for all \( i = 1, \ldots, M \) and \( f_j(u) < f_j(v) \) for at least one \( i = 1, \ldots, M \). When neither solution dominates the other, the two solutions are said to be non-dominated. The result of a multi-objective optimization is a set of non-dominated solutions. When no other feasible solutions in the search space dominate these solutions, they are said to be Pareto-optimal. Two main properties are desired in a set of non-dominated solutions; convergence and diversity. The convergence of a non-dominated set is a measure of how close it is to the Pareto-optimal set, whereas the diversity measures the spread of the
solutions within the set. For example, consider the set of solutions shown in Figure 2.2. The gray solutions are the Pareto-optimal set and the red solutions are dominated by at least one other solution, so they are not in the Pareto-optimal set. The solutions in the box are all dominated by the solution labeled $x$, because they are worse than $x$ in terms of both objectives.

Figure 2.2: Pareto-optimal points.

2.3 PERFORMANCE INDICATORS

To compare optimization experiments with one another, there must be a way of assessing the performance of an optimization. Measurements used to make such comparisons are called performance indicators. For single-objective problems, the performance indicator could be directly related to the best function value attained by the algorithm. For multi-objective problems, there is no single best solution; instead, the output is a set of non-dominated solutions. As mentioned above, the quality of the non-dominated set is usually measured in terms of its diversity and convergence. Unary performance indicators can be used to obtain a single performance measurement from a set of non-dominated solutions.

Two such indicators are the inverted generational distance ($I_{IGD}$) (Bosman and Thierens, 2003) and hypervolume ($I_H$) (Zitzler, 1999).

1. The inverted generational distance of a non-dominated solution set $S$ obtained from an multi-objective evolutionary algorithm (MOEA) is defined as,

$$ I_{IGD}(S, P) = \frac{1}{|P|} \sum_{x^* \in P} \min_{x \in S} \| \mathbf{F}(x) - \mathbf{F}(x^*) \|, \tag{2.2} $$

where $P$ is a reference set of Pareto-optimal solutions representing the desired density and distribution of points on the Pareto-optimal front, $\| \cdot \|$ denotes the Euclidean distance, and $| \cdot |$ denotes the size of a set. Solution sets with lower $I_{IGD}$
values are preferred since they closely resemble the reference set, \( P \). Thus, the best possible value for \( I_{GD} \) is 0.

2. The hypervolume of a non-dominated solution set \( S \) obtained from an MOEA is defined as,

\[
I_H(S, r) = \lambda \left( \bigcup_{x \in S} H(F(x), r) \right),
\]

where \( r \) is a reference point that is dominated by all solutions in \( S \), \( \lambda(\cdot) \) is the Lebesgue measure, and \( H(a, b) \) represents the hypercube with the body diagonal \( ab \). Solution sets with higher \( I_H \) values are better because they dominate a larger region of the objective space. Hypervolume is the only known unary quality indicator that is Pareto-compliant (Zitzler, Brockhoff, and Thiele, 2007), i.e. for any two solution sets, \( S_1 \) and \( S_2 \) (\( \neq S_1 \)), \( S_1 \subseteq S_2 \Rightarrow I_H(S_1) > I_H(S_2) \).

2.4 CONFIGURATION OF OPTIMIZATION ALGORITHMS

Most optimization algorithms involve one or more parameters that can be used to change the search behavior of the algorithm. This allows the algorithm to be adapted to better suit different problems. However, it is often difficult to know the optimal parameter configuration without performing parameter tuning experiments. These experiments are computationally expensive, since a complete optimization has to be performed for each parameter configuration. It is therefore unrealistic to expect parameter tuning experiments to be performed for every new algorithm and problem combination. Furthermore, the optimal parameter configurations will differ from problem to problem. Instead, each algorithm has certain recommended (default) parameter values and heuristics that can be used for new problems. There is, however, no guarantee that the default parameter values will be effective for any particular problem. The values assigned to the control parameters are referred to as the control parameter values (CPVs), and a complete set of control parameter values for a particular algorithm is called a CPV tuple. The terms parameter configuration and CPV tuple are used interchangeably in this thesis.

The problem of setting the parameters can be broadly divided into two types:

- **Parameter tuning**—the parameters are set at the beginning and are kept constant during the optimization. This is also known as offline parameter tuning.

- **Parameter control**—the parameters are initialized at the beginning and are then adapted according to certain rules or heuristics during the optimization. This is also known as online parameter tuning.

The parameters of an optimization algorithm are generally kept constant throughout the optimization. However, studies have shown that this is suboptimal and that the efficiency of the optimization algorithm can be increased by adapting the parameters during the course of optimization (Bäck, 1992; Hesser and Männer, 1990). It is difficult to theoretically derive what the optimal adaption of the parameters should be for anything but the simplest problems, because of complex interactions between the parameters. Control mechanisms are therefore usually certain heuristics that are inspired by concepts in statistics and nature, in the same way as many optimization algorithms are. Parameter control mechanisms usually only consider at most a few parameters, because it is difficult to understand and predict the interactions between the parameters. However, that
Several different parameter control mechanisms are proposed for various evolutionary algorithm components (see Karafotias, Hoogendoorn, and Eiben (2015) for a comprehensive overview). These mechanisms can be divided into three categories:

- **A parameter-dependent mechanism**—controls a specific component, such as population size or mutation rate.

- **A control ensemble**—combines multiple mechanisms to balance the trade-offs between exploring and exploiting the search space.

- **A parameter-independent mechanism**—adapts the parameters based on heuristics that are not specific to any parameter and can also use feedback on how well a particular parameter is performing.

Parameter control mechanisms can also be considered specialists or generalists, in the sense that they do not violate the no free lunch theorem (Wolpert and Macready, 1997). However, it is reasonable to assume that because of their greater flexibility they can provide better trade-offs than can static parameters.

It is possible to distinguish three layers in parameter tuning: the application, the algorithm (i.e. lower), and design (i.e. upper) layers. The problem to be solved is located in the application layer and the metaheuristic that solves that problem is in the algorithm layer. In the design layer, it is the parameter tuner that tests different CPV tuples for the metaheuristic in the algorithm layer. To avoid confusion, the quality of solutions to the problem in the application layer is called *fitness* (Eiben and Smit, 2011).

Parameter tuning methods can be divided into two main categories: iterative and non-iterative. Non-iterative tuners generate all the parameters at the start, usually in a systematic fashion. This allows the utility landscape to be modeled from the utility of the evaluated parameters. Iterative tuners, on the other hand, generate the parameters iteratively as the tuner progresses. This makes them more suitable for finding the near-optimal parameter configurations, because they can perform a search of the utility landscape.

### 2.5 Generalist and Specialist CPV Tuples and the No Free Lunch Theorem

Another important aspect of parameter tuning is that of *specialist* versus *generalist* CPV tuples. A generalist CPV tuple is designed to work across a wide range of optimization problems and is suitable to use for new problems. A specialist CPV tuple is instead fine tuned for a small set of specific problems. The performance difference between a specialist and generalist can be substantial (Smit and Eiben, 2010b). It can therefore be worthwhile to tune a specialist when similar optimization problems are solved repeatedly.

A generalist cannot violate the no free lunch theorem. It is only a generalist in the sense that it is designed to work well for a particular group of problems. In the same sense that optimization algorithms can be designed to be effective at a specific set of problems. However, the no free lunch theorem does not necessarily lead to the conclusion
that research into optimization algorithms is pointless, just because it is impossible to guarantee that any optimization algorithm is better than random search for all problems. Many real-world problems exhibit structure, such as local smoothness or symmetries, that can be exploited by an optimization algorithm (Joyce and Herrmann, 2018). This naturally leads to the related problem of identifying the exploitable structures of any given problem. Even partially solving this problem would be helpful in selecting an appropriate algorithm and parameter configuration for a given problem. This is still an open research question without a definitive answer.

How many problems a generalist should be tuned against is a difficult question to answer. A smaller set allows for more specific structures to be exploited, but will probably cause the algorithm to perform worse on problems not exhibiting such structures. A large set will improve the likelihood that the parameters will work for problems that the generalist was not tuned against, but the performance loss compared with a specialist can be substantial (Smit and Eiben, 2010b; Andersson, Bandaru, Ng, and Syberfeldt, 2015a). Unfortunately, the answer to this question is context dependent and relies on the experience of the practitioner.

2.6 SIMULATION-BASED OPTIMIZATION

Simulation models of real-world processes are often built to test new ideas and improvements before they are implemented in the real world. This enables simulation studies whose aim is to find the best-performing set of design variables according to predetermined criteria. Any changes in the design variables are evaluated against these criteria. Such simulation studies enable rapid experimentation with minimal disturbance and provide the decision maker with additional confidence in the effectiveness of the changes.

The number of possible solutions grows exponentially as the number of design variables is increased. It is therefore only feasible to perform manual or exhaustive experimentation when there are relatively few design variables. An alternative is to formulate the question to be answered as an optimization problem and solve it with an optimization algorithm. Since the optimization algorithm will test only a small fraction of the available solutions, more design variables can be included in the problem formulation than would have been possible with manual experimentation. This is referred to as simulation-based optimization (SBO) or simulation-based multi-objective optimization (SMO) for problems with more than one objective.

SMOs are difficult to solve for the following reasons:

- **Interacting variables**—changing a buffer size might require a corresponding change in the processing time of an earlier machine.

- **Noise**—Simulation models can have stochastic factors such as breakdowns.

- **Multiple nonlinear objectives**—with multiple conflicting objectives, such as throughput and lead time, there is more than one optimal solution and the optimization needs to find this trade-off set.

- **Multiple local optima**—when there are multiple local optima, the optimization can get stuck and fail to find the global optima.

The computational cost of solving an optimization problem with these characteristics can be significant because of the vast search spaces that need to be navigated. This is
especially true for SBO, since the time needed to run a simulation can be measured in minutes or even hours.

2.7 A SCALABLE SIMULATION MODEL

The simple stochastic discrete-event simulation model considered in the experiments of this thesis represents an unpaced flow line, consisting of \( s \) workstations with \( s - 1 \) inter-station buffers. The terms “workstation” and “machine” are interchangeable in the following discussions, since there is only one machine in each workstation. The productivity of each machine \( i \) is governed by its availability \((\alpha_i)\), processing time \((\beta_i)\), and repair time \((\gamma_i)\). In the initial state, the workloads of all the workstations are perfectly balanced, each having a processing time of \( \beta_{orig} = 80 \) seconds per job. All machines have an availability of \( \alpha_{orig} = 90\% \), and repair time of \( \gamma_{orig} = 300 \) seconds. The processing times are assumed to be constant, which is realistic for automated machining processes. The times to failure of the workstations are modeled using exponential distributions and the randomness of the repair times, \( \gamma_i \), is modeled using Erlang distributions.

In a complex flow line with unbalanced workloads, the detection of bottlenecks is essential for any subsequent improvement of the production rate or throughput. The location of bottlenecks depends on many factors, including the job flow logic, variability and disturbance of the machines and the buffer allocations. Even for a simple, straight flow line with balanced workloads as described above, detecting which workstation(s) to improve in order to increase the overall throughput of the line to a certain level is not a trivial task. The concept of treating this throughput improvement problem as a multi-objective optimization problem of identifying the optimal (minimal) number of changes to maximize the throughput was first proposed by Pehrsson (2013) and later further elaborated by Ng, Bernedixen, and Pehrsson (2014). In such an optimization formulation, the system throughput \((T_H)\) is the primary objective of improvement, so that \( f_1(x) = \max\{T_H(x)\} \).

The total number of changes, i.e. improvement actions, can be defined as the secondary objective function, \( f_2(x) \). There are three integer improvement variables \( \{\alpha_i, \beta_i, \gamma_i\} \). The available improvement actions for availability, processing times, and repair times and their corresponding step sizes are:

\[
\begin{align*}
\alpha &= \{80, 96\} & \Delta \alpha &= 1 \\
\beta &= \{60, 80\} & \Delta \beta &= 1 \\
\gamma &= \{180, 360\} & \Delta \gamma &= 1
\end{align*}
\]

The second objective, \( f_2(x) \), can then be written as a summation of improvements (Ng, Bernedixen, and Pehrsson, 2014):

\[
f_2(x) = \min \left\{ \sum_{i=1}^{s} \alpha_i + \sum_{i=1}^{s} \beta_i + \sum_{i=1}^{s} \gamma_i \right\}
\]

where

\[ \alpha_i = \frac{\alpha_i - \alpha_{orig}}{\Delta \alpha} , \quad \beta_i = \frac{\beta_{orig} - \beta_i}{\Delta \beta} , \quad \gamma_i = \frac{\gamma_{orig} - \gamma_i}{\Delta \gamma} . \]

Additionally, to simultaneously solve the lean buffer problem (Enginarlar, Li, and Meerkov, 2005), the capacities of inter-station buffer spaces are also optimized, \( B_i = \{1, 2, \ldots, 10\} \) \( \forall i \in \{1, \ldots, s - 1\} \), by adding a third objective of minimizing the total number of buffers, i.e. \( f_3(x) = \min\{\sum_{i=1}^{s-1} B_i\} \).

The advantage of such a scalable simulation model is that it can be used to control the difficulty of the optimization problem, in much the same way as in other test problems.
It is also designed to resemble real-world improvement problems while still being conceptually simple.

2.8 PARALLEL METAHEURISTICS

Metaheuristics have traditionally been sequential in the sense that the solutions are both generated and evaluated in sequence. Much of the increase in computing power in recent years comes from the addition of parallel computing and not from advances in processor speed. It therefore becomes increasingly important for metaheuristics to be able to efficiently exploit these parallel computing resources.

There are three major parallelization strategies for metaheuristics, which fall into three distinct hierarchical levels, as shown in Figure 2.3 (Talbi, 2009). On the algorithmic level, multiple independent or cooperating metaheuristics are used to achieve parallelization. This is problem independent and also could affect the behavior of each metaheuristic. The iteration level concerns the parallelization of a single metaheuristic. An example of this would be a population-based metaheuristic in which all the solutions in a generation are evaluated in parallel. This is also problem independent, but it does not affect the metaheuristic behavior. The last level is the solution level, which concerns the parallelization of a single solution evaluation. Either the objective function is divided into several parts that can be executed in parallel or the input data are partitioned. This is problem dependent, since both these approaches are tightly coupled to the problem at hand. Solution level parallelization is useful for speeding up a single evaluation, but it cannot make the optimization itself more efficient, so this type parallelization will not be considered further.

Another aspect that needs to be considered at the algorithmic level is how solutions are migrated between metaheuristics. In single-objective problems, this issue is mitigated by the fact that only a limited set, usually only the best one, of solutions needs to be considered at any given time. This means that the amount of information that needs to
be exchanged is also limited. In a multi-objective problem, there may be multiple non-dominated solutions, making the task of choosing which solutions to migrate and when to do so a difficult one.

There are several reasons why parallelizing metaheuristics can be beneficial. The first one is to speed up the search, which is accomplished by utilizing more resources. This also allows larger optimization problems to be solved. Another benefit is that since parallelization on the algorithmic level can change the behavior of the metaheuristics, there is also potential to improve the quality of the search—in other words, finding better solutions within the same computational budget.

2.9 PARALLEL EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are a category of metaheuristics inspired by concepts from nature in general and evolution by natural selection in particular. They maintain a population in which individuals have to compete for survival. New offspring are created by cross-breeding, and mutating individuals selected from the population. A selection pressure that drives the population towards better solutions is created by favoring better individuals when selecting parents and when deciding which individuals will survive to the next generation. There are two main strategies for handling new offspring in an EA, generational and steady-state. A generational EA has separate stages, performed in series, for generating new individuals and merging those individuals back into the population. In contrast, a steady-state EA will immediately update the population after an individual is created and evaluated. An asynchronous steady-state algorithm performs multiple creation, evaluation, and updating steps in parallel.

The advantage of a steady-state asynchronous algorithm in a parallel context is high utilization of the available resources, since the algorithm never has to wait for any specific evaluation. In contrast, a generational algorithm must wait for all evaluations to be completed before it can proceed to the next generation. However, more simultaneous individuals will also increase the age of the information available to the optimization algorithm. Depolli, Trobec, and Filipič (2013) call this selection lag and define it as the number of solutions created while an observed solution is being evaluated. The result of a steady-state asynchronous optimization is also dependent on the order in which evaluations are completed. This means that reruns of an optimization are not guaranteed to find the same solution(s), even when the same seed is used for random number generation.

There is a concern that steady-state asynchronous algorithms will have a bias towards fast evaluating solutions when evaluation times are stochastic. The concern is that multiple fast-evaluating solutions may be born, evaluated, and merged into the population while a slow-evaluating solution is still being evaluated. This question was investigated by Scott and De Jong (2015) who found that this effect is negligible.

The parallelization potential of both generational and steady-state asynchronous EAs is determined by the offspring population size. For a generational EA, offspring population size controls the number of new individuals created in each generation, and for a steady-state asynchronous EA, it is the maximum simultaneous individuals that can exist. This means that the parallelism is capped by the offspring population size. Increasing this number will increase the parallelism but it will also affect the search behavior of the algorithm, possibly negatively. The trade-off is therefore between increased parallelism potential and worsened search behavior.
The parallelism architecture used in this thesis is the master-slave model, which is arguably the simplest to implement. There are, however, other parallel models in which the parallelism of an EA can be increased, for example, by combining multiple cooperating EAs that exchange candidate solutions among each other using distributed islands or cellular models (Alba, Luque, and Nesmachnow, 2013). The advantage of these methods, which belong on the algorithmic level, are that it is possible to achieve synergy effects, but they are also more complicated to implement and it is not trivial to know which algorithms to combine and how.
RESEARCH METHODOLOGY
CHAPTER 3
RESEARCH METHODOLOGY

The aim of this thesis is to add to the existing body of knowledge regarding the parameter tuning problem. An artifact that can perform large-scale parameter tuning experiments will be designed and implemented to accomplish this aim. This artifact will be used to evaluate EAs as tuners and to solve parameter tuning problems. A design and creation research strategy (Hevner et al., 2004) is used to design, develop, and evaluate this artifact. The focus of the design and creation research strategy is to create new IT products, also called artifacts. For an IT product to qualify as research, it needs to contribute to the body of knowledge in some manner. Such research should also display academic qualities, such as analysis, explanation, argument, justification, and evaluation.

Figure 3.1: Design science framework (Hevner et al., 2004).
3.1 DESIGN SCIENCE GUIDELINES

Hevner et al. (2004) suggests seven guidelines for the successful application of design science research. The following section relates these guidelines to this research.

**Design as an artifact.** The result of design science should be useful artifacts, such as models, methods, or instantiations. This research will present a framework, software architecture, and an instantiation that can be used to perform the parameter tuning of optimization algorithms using parallel computing resources.

**Problem relevance.** The problem solved by the artifact should have business relevance. Optimization problems can be found in many different domains, and improving optimization algorithms through parameter tuning will enable more difficult optimization problems to be solved.

**Design evaluation.** The utility, quality, and efficacy of the artifact must be demonstrated using evaluation methods. The usefulness of the artifact will be evaluated using extensive empirical experiments and the results analyzed with appropriate statistical methods. The experiments will demonstrate that the artifact is suitable for tuning the control parameters of various optimization algorithms. They will also demonstrate that it is competitive with other available state-of-the-art parameter tuning methods.

**Research contributions.** Design science must provide clear and verifiable contributions to areas related to the artifact. The contributions of this research are the framework, software architecture, and viability of evolutionary algorithms for solving parameter tuning problems using parallel computing resources. This research will also contribute to the body of knowledge of optimization algorithms in general and of the effects of their control parameters in particular.

**Research rigor.** Design science requires the application of rigorous methods in both constructing and evaluating the artifact. This research will build on the existing literature regarding optimization algorithms and how they have been used to solve the parameter tuning problem. The experiments in this thesis will be well defined and the results will be analyzed using statistical methods.

**Design science as search method.** Finding an effective artifact requires utilizing the available means while staying within the problem environment. The framework and software architecture will be developed through a prototyping method using a generate-and-test cycle. The viability of the artifact will first be demonstrated by using it to improve the commonly used control parameter values for both single- and multi-objective optimization algorithms, and by later comparing it with other state-of-the-art parameter tuning methods.

**Communication of research.** Design science research must be presented to both technically and management-oriented people. The research in this thesis will be made available in the form of published papers.

3.2 EMPIRICAL RESEARCH

The main strategy for evaluating the artifact is to perform empirical experiments. This is motivated by the fact that optimization algorithms use heuristics to navigate the fitness landscape. Because of this, impossible to investigate these methods using a purely theoretical approach. Instead, experiments are a vital component of research into optimization algorithms (Bartz-Beielstein, 2006).
Bartz-Beielstein (2006) compared the approach of new experimentalism with the discipline of experimental algorithmics. It is said that many approaches in experimental algorithmics are based on Popperian paradigms:

1. No experiment should be conducted without theory.
2. Theories should be falsifiable.

Instead, based on the work of Hacking (1983) and Mayo (1994), it is argued that:

1. An experiment can have a life of its own.
2. Falsifiability should be complemented with verifiability.

This is known as the new experimentalism (Mayo, 1994). It provides a statistical methodology for gaining knowledge from experiments. It also highlights the importance of distinguishing between the statistical significance of an experimental result and its scientific meaning. New experimentalists learn from both negative and positive results. They also seek scientific conclusions that can be validated independently of complex abstract theories. This is also the meaning of the statement that an experiment can have a life of its own, i.e. an experiment does not necessarily need a supporting theory to be useful (Bartz-Beielstein, 2006).

3.2.1 EXPERIMENTAL RESEARCH

Experiments on metaheuristics need to follow a good methodology, otherwise it will be difficult to draw sound conclusions from them, especially if the results are to be compared with those of other studies. Eiben and Jelasity (2002) identified four main issues with many experimental studies:

1. The test problems are chosen in an ad hoc manner.
2. The performance outcomes are used without relating them to research objectives.
3. The conclusions are over generalized.
4. The results are difficult to reproduce.

The first issue can be addressed by justifying the inclusion or exclusion of test problems. For studies intended to assess the performance of an algorithm, it is common to include a mixture of problems. However, it is often difficult to know which problems to include, because there are no standard test problems that are universally used. There are test problems that are often used, but even those are not used consistently in the literature. Furthermore, the test problems are usually configurable and not all studies report the configuration used. For real-world problems this issue is even worse, since the selection of standardized real-world problems is even smaller. For application studies, the entire point of which is to solve particular problems, this becomes a non-issue. However, for such studies the last issue is even worse, since their results are often difficult for others to reproduce. It could be impossible, for legal or commercial reasons, to reveal exact details of the problem formulations. The formulation could also require special software that is either not freely available to the public or very expensive to purchase. The second issue can be addressed by ensuring that the performance indicators used actually can be used to support or falsify the hypothesis and/or objectives. In order not to formulate the conclusions too broadly, the researcher needs to ensure that the conclusions are actually supported by the results of the study derived from thorough statistical analysis.
3.2.2 EMPIRICAL VERSUS THEORETICAL APPROACHES

Empirical approaches are susceptible to researcher biases, such as discarding or ignoring results and designing experiments that favor specific outcomes. There is also a lack of standards for empirical methods. This, among other reasons, is why some researchers do not regard empirical approaches as scientific. On the other hand, some researchers believe that theoretical approaches are not well-suited for assessing algorithm performance (Bartz-Beielstein, 2006). The difficulty of reproducing the results of empirical studies discredits the empirical approach (Eiben and Jelasity, 2002). That is why it is important to incorporate standards and methodology, which make it harder to cheat and easier to reproduce the results; they also help minimize the influence of experimenter biases.

The objections to empirical approaches stem from statements such as: “Algorithms are defined as formal systems and should be studied with formal methods.” However, Bartz-Beielstein (2006) argued that the complexity of self-explanatory formal systems is such that they cannot be applied in real-life situations. Furthermore, he stated that studying algorithms with formal systems requires some kind of reductionism, and that reductionism only works in some cases. Based on his experience as an experimenter, he claims that: “Reductionism often fails in algorithmic science.”

For these reasons, Bartz-Beielstein (2006) concludes that a purely theoretical approach is not satisfactory. They also state the following two reasons why experiments are useful. Theories could be incomplete or only work in some cases; in those cases, observations and experiments can be used to test how well a theory holds up. Experiments may also lead to new theories or the refinement of existing ones; they can also help determine the average case results, which can be difficult to deduce using theoretical methods.

3.2.3 NOISE AND HYPOTHESIS TESTING

Even if a hypothesis is true, it is still possible that observational data may not exactly agree with it. The error could be due to, for example, precision issues with measurements or random fluctuations beyond the experimenter’s control. Metaheuristics generally require some source of randomness as an important part of the algorithm. For example, an evolutionary algorithm that uses mutation applies the mutation to the variables based on the outcome of a random process. Even in cases in which the metaheuristic is completely deterministic, the initial solutions are usually still generated randomly. Furthermore, simulation-based problems usually have stochastic factors, such as machine failure rate and processing times.

Since the algorithms are run on computers that are deterministic, the random numbers are not truly random, but instead called pseudorandom. A pseudorandom random generator is initialized with a seed value, from which subsequent pseudorandom numbers are generated. This means that the same seed will repeatedly generate the same sequence of random numbers. This is a useful property, because it means that even experiments with random factors can be reproduced exactly, at least on the same hardware and software configuration. There could still be differences in precision between, for example, different CPU architectures. The randomness, either in the algorithm or the problem, means that there will always be a level of uncertainty in the results.

Hypothesis testing is a useful technique that can help determine whether the apparent difference between experimental results is statistically significant or a chance occurrence. Two hypotheses are usually formulated. The null hypothesis, represented by $H_0$,
is the default assumption that there is no difference between the results. The alternate hypothesis denoted by $H_a$ states that the results are significantly different. A statistical hypothesis test is performed to see whether the null hypothesis can be rejected, at some predetermined level of significance. A successful test means that the null hypothesis can be rejected and the alternative hypothesis tentatively accepted. Two types of errors can occur in statistical testing, type I and type II errors. A type I error occurs when the null hypothesis is rejected, when in fact it is true. This could lead to an erroneous conclusion that one algorithm is superior to another. A type II error occurs when the test fails to reject the null hypothesis, when the alternative hypothesis is actually true. A type I error is generally considered to be more serious, because the default assumption of “no difference” is incorrectly rejected, making further analysis based on this result flawed. A type II error simply means that no difference was detected, even though it was present.

Many different statistical tests can be used to reject the null hypothesis. They all have their own strengths and weaknesses, as well as assumptions that must be met before they can be used. A parametric test assumes that data come from a probability distribution with certain parameters, such as a normal distribution. A non-parametric test does not make such assumptions about the data. When the parametric test assumptions about the data hold, the statistical power is greater than that of a corresponding non-parametric test and should therefore be preferred. On the other hand, when those assumptions do not hold, a parametric test will be unreliable. In those cases, a non-parametric test should be used instead. For the experiments in this research, the normality assumption cannot be sufficiently guaranteed. For this reason, a non-parametric test will be used instead. For comparing two groups, one of the many parametric or non-parametric tests can be used without further steps. However, comparing more than two groups is more difficult. A common approach is to first perform analysis of variance (ANOVA), to test the null hypothesis that the groups are the same. Following a successful rejection of the null hypothesis, a post hoc test is performed that identifies groups that are significantly different.

3.2.4 INTERNAL AND EXTERNAL VALIDITY

The internal validity of metaheuristic research can be assured by the fact that most of the factors that affect the outcome of the experiment are controlled for. Factors, such as randomness, that are outside of the experimenter’s control can be mitigated by proper statistical testing. That, of course, hinges on the assumption that the experiment is designed to support any subsequent conclusions. The external validity is measured by how well the results can be generalized and applied to other scenarios. This is more difficult to achieve. By performing the experiments on a wide selection of problems, the chance that the results are applicable to other problems should increase. Also, describing the experiments in such detail that they are easily replicated would help others reach similar conclusions.
FRAMEWORK AND SOFTWARE ARCHITECTURE FOR PARAMETER TUNING
CHAPTER 4
FRAMEWORK AND SOFTWARE ARCHITECTURE FOR PARAMETER TUNING

This chapter starts with an overview of papers relevant to solving the parameter tuning problem. A framework and software architecture is then presented for solving parameter tuning problems using a bilevel optimization approach. This framework and software architecture builds on existing research, with an emphasis on scalability.

4.1 EXISTING STUDIES OF PARAMETER TUNING

One of the earliest examples of using evolutionary algorithms to solve the parameter tuning problem can be found in Mercer and Sampson (1978). The authors used only a single run of the evolutionary algorithm, but the approach was later extended to multiple runs by Grefenstette (1986). Although the study succeeded in finding good control parameter values, the computational complexity limited the scope of the experiments and also their usefulness.

An early attempt to utilize parallel computing resources for parameter tuning was presented by Bäck (1994). A master-slave approach was used to run the lower-level optimization on separate processors. There is, however, no indication that the system was distributed over more than one computer.

Myers and Hancock (2001) used a Graeco-Latin square design with a wide range for each parameter setting. This was followed by a fully crossed factorial design with narrower ranges. This allowed interactions between the algorithm performance and the control parameters (specifically, population size) to be found. The exponential nature of the factorial design limited the number of interactions that could be modeled.

Bartz-Beielstein, Lasarczyk, and Preuss (2005) used sequential parameter optimization (SPO) in three different parameter tuning scenarios: a new algorithm on well-known problems, a well-known algorithm on well-known problems, and well-known algorithms on complex real-world problems. SPO consists of three iterative stages: the first stage selects new design points; the second stage estimates the algorithm performance by running an optimization with the selected parameter configurations; and the final stage builds a model that estimates the algorithm performance for untested parameter configurations. The authors conclude that SPO found parameter configurations that led to improved performance in all three scenarios.

Adenso-Diaz and Laguna (2006) proposed CALIBRA, a parameter tuning method that uses Taguchi’s fractional factorial experimental designs with a local search to find the best parameter configuration for algorithms with up to five control parameters. The authors tested their method on six different problems, and found that it could find better parameter configurations for some, but not all, of them.
Table 4.1: Literature overview.

<table>
<thead>
<tr>
<th></th>
<th>Bilevel tuning framework</th>
<th>Meta-EA approach</th>
<th>Other iterative approach</th>
<th>Non-iterative approach</th>
<th>Distributed system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mercer and Sampson, 1978</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grefenstette, 1986</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bäck, 1994</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myers and Hancock, 2001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Bartz-Beielstein, Lasarczyk, and Preuss, 2005</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adenso-Diaz and Laguna, 2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hutter, Hoos, and Stützle, 2007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nannen and Eiben, 2007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smit and Eiben, 2009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hutter, Hoos, and Leyton-Brown, 2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branke and Elomari, 2012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sinha et al., 2014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ugolotti and Cagnoni, 2014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>López-Ibáñez et al., 2016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kakuguchi et al., 2016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dymond, Kok, and Heyns, 2017</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This thesis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ParamILS is an iterated local search method (Hutter, Hoos, and Stützle, 2007) that uses a combination of default and random settings for initialization; these settings are then perturbed by an iterative local search method with a fixed number of random moves. It also has random restarts and neighboring exchange mechanisms to avoid converging to suboptimal parameters.

Relevance Estimation and Value Calibration (REVAC) (Nannen and Eiben, 2007) is a method based on information theory and used to measure parameter relevance. It is an estimation of distribution algorithm that estimates the expected performance of the parameter values chosen from a probability density distribution. A distribution with a narrow peak indicates a highly relevant control parameter, whereas a broad plateau indicates a moderately performing control parameter. Smit and Eiben (2010a) used REVAC to improve the CEC 2005 competition winner.

A comparison of different parameter tuning methods and the effects of two add-ons, rac-
ing and sharpening, were presented by Smit and Eiben (2009). The considered parameter tuning methods were: Meta-EA with CMA-ES, REVAC, and SPO. The two add-ons are opposing forces, since racing reduces the number of tests and sharpening increases them. In this paper, the add-ons are also combined, in order to obtain the benefits of both. The authors concluded that Meta-EA with CMA-ES is preferred if the purpose is to find the best parameter configuration.

Sequential model-based optimization (SMBO) is a model-based regression method. Hutter, Hoos, and Leyton-Brown (2011) identified three key limitations of SMBO for parameter tuning: (i) it only supports numerical parameters; (ii) it can only tune specialists; and (iii) it cannot terminate poorly performing parameter configurations early on. The main contribution of the present paper is to remove the first two constraints. Hutter, Hoos, and Leyton-Brown (2011) tested their improvements by tuning CPLEX on 17 configurations, and found that it could find equal or better parameter configurations than all other considered parameter tuning approaches.

Branke and Elomari (2012) proposed a parameter tuning method that could find the best parameter configurations for multiple computational budgets in a single run. It uses a rank-based approach inspired by the non-dominated sorting mechanism of NSGA-II. The authors used this method to tune the parent and offspring population size of CMA-ES on eight test problems. The authors concluded that their flexible budget method could find parameter configurations as good as those found using a fixed budget method, but with substantial time savings.

Sinha et al. (2014) used a bilevel optimization approach to solve the parameter tuning problem. Quadratic approximations were used to reduce the number of lower-level optimizations, thereby reducing the overall computational complexity. The optimization algorithms they tuned were DE (i.e. differential weight and crossover probability) and Nelder-Mead simplex search (i.e. reflection coefficient, expansion coefficient, and contraction coefficient). The results indicate that their algorithm was able to converge close to the best parameter values in most cases.

Ugolotti and Cagnoni (2014) used NSGA-II to solve a bi-objective parameter tuning problem that took into consideration both the precision and speed when tuning DE and particle swarm optimization (PSO). Precision was defined as the best result obtained after a certain number of fitness evaluations. Speed was defined as the number of function evaluations needed to reach a minimum fitness threshold. From the non-dominated front of these two objectives, the authors extracted knowledge of the effects of control parameters. For example, they found that the population size mainly influenced the location of a configuration on the Pareto-optimal front.

Kakuguchi et al. (2016) used a multi-objective evolutionary algorithm (NSGA-III) to find robust control parameters for multi-objective optimization algorithms. Finding robust or generalist control parameters requires that the control parameter values be tested on multiple problems. In the present study the approach was to use the hypervolume from each problem as a separate upper-level objective. Although only two problems were used, the study found that the optimal parameter configurations differed between the problems.

López-Ibáñez et al. (2016) proposed irace, which is an iterated racing algorithm for automatic configuration. Bezerra, López-Ibáñez, and Stützle (2016) used irace to design optimization algorithms from a template of various components commonly used in evolutionary algorithms.

Iterated racing consists of three steps. The first step is to sample new configurations ac-
cording to a distribution. In the second step, the best configurations are selected from the newly sampled ones by means of racing. The final step is to update the sampling distribution in order to bias it towards the best configurations. These steps are repeated until a stopping criterion is met. The input to irace is one or more problem instances used during each race. The order in which the instances are used is randomized. Racing refers to the process of systematically eliminating statistically worse configurations in a series of races or iterations. Each race starts by running a minimum number of instances (the default is five), after which the parameter configurations that perform statistically worse are discarded. Additional instances are then run for each parameter configuration, so that more parameter configurations can be discarded. This process is repeated until only a few parameter configurations are left and the race is completed.

Each parameter has its own independent sampling distribution. Numerical and ordinal parameters use a truncated normal distribution, whereas categorical parameters use a discrete distribution. The statistical test can be any appropriate test, but the default choice in irace is Friedman’s non-parametric two-way analysis of variance by ranks.

The Many-Objective Tuning Algorithm (MOTA) was proposed by Dymond, Kok, and Heyns (2017). In its problem formulation, it uses speed as both an objective and a decision variable. Each CPV tuple is assessed against multiple function evaluation budgets. It also uses a preemptive terminating strategy for noise handling. The results indicate that MOTA was able to outperform other parameter tuning methods.

For a good and detailed discussion of the fundamentals of parameter setting for evolutionary algorithms, see Lobo, Lima, and Michalewicz (2007).

Table 4.1 provides an overview of the papers discussed above. “Bilevel tuning framework” refers to whether the paper explicitly states that at least some important parts of the software are replaceable, for example, the ability to use different upper-level algorithms or to exchange the upper-level objectives. None of the papers describes the software architecture or implementation details regarding parallelization. Most of the bilevel optimization approaches probably utilize parallel processors on a single computer. However, since they do not mention whether the experiments were run on more than one computer, they are not classified as having a distributed system. The irace package has support for distributing evaluations using the Message Passing Interface (MPI) or by writing custom functions to distribute evaluations. To the best of this author’s knowledge, no bilevel optimization approaches have had the explicit aim of supporting large scale parameter tuning experiments using distributed parallel computing resources. The following sections will present a framework and software architecture that has been designed and implemented to achieve that specific aim.

4.2 FRAMEWORK

The thesis uses a bilevel optimization approach to solve the parameter tuning problem. This section presents the underlying framework used to support this approach. An overview of the framework is shown in Figure 4.1. The highlights of the framework are:

- Parallel evaluations of CPV tuples by running multiple lower-level optimizations, including replications, simultaneously.
- Ability to tune both specialist and generalist CPV tuples.
- Support for both static and dynamic upper-level resampling strategies.
- Handling of single- and multi-objective parameter tuning formulations at the upper level.
- Support for single- and multi-objective optimization problems at the lower level.
- Support for the use of different optimization performance indicators.

\[ U_1(p^*) = \sum_{t=1}^{q_{post}} U_{1,t}(p^*) \quad \cdots \quad U_{M}(p^*) = \sum_{t=1}^{q_{post}} U_{M,t}(p^*) \]

Post-replications of the best parameter sets

Using \( p \)

Minimize \( \{ U_{1}(p), \ldots, U_{M}(p) \} \)

Subject to \( p^{l} \leq p \leq p^{u} \)

Stop when \( P_{1}(r) = P_{1,target} \) or \( F_{E_{1}}(r) = F_{E_{1},max} \)

Replication 1

Problem 1

Using \( p \)

Minimize \( \{ f_{1}(x_{1}), \ldots, f_{M_{1}}(x_{1}) \} \)

Subject to \( g_{j}(x_{1}) \geq 0 \; \forall \; j \)

\( h_{k}(x_{1}) = 0 \; \forall \; k \)

\( x_{1}^{l} \leq x_{1} \leq x_{1}^{u} \)

Stop when \( P_{1}(1) = P_{1,target} \) or \( F_{E_{1}}(1) = F_{E_{1},max} \)

Using \( p \)

Minimize \( \{ f_{1}(x_{1}), \ldots, f_{M_{1}}(x_{1}) \} \)

Subject to \( g_{j}(x_{1}) \geq 0 \; \forall \; j \)

\( h_{k}(x_{1}) = 0 \; \forall \; k \)

\( x_{1}^{l} \leq x_{1} \leq x_{1}^{u} \)

Stop when \( P_{1}(r) = P_{1,target} \) or \( F_{E_{1}}(r) = F_{E_{1},max} \)

Replication r

Problem 1

Using \( p \)

Minimize \( \{ f_{1}(x_{1}), \ldots, f_{M_{1}}(x_{1}) \} \)

Subject to \( g_{j}(x_{1}) \geq 0 \; \forall \; j \)

\( h_{k}(x_{1}) = 0 \; \forall \; k \)

\( x_{1}^{l} \leq x_{1} \leq x_{1}^{u} \)

Stop when \( P_{1}(r) = P_{1,target} \) or \( F_{E_{1}}(r) = F_{E_{1},max} \)

Replication 1

Problem q

Using \( p \)

Minimize \( \{ f_{1}(x_{q}), \ldots, f_{M_{q}}(x_{q}) \} \)

Subject to \( g_{j}(x_{q}) \geq 0 \; \forall \; j \)

\( h_{k}(x_{q}) = 0 \; \forall \; k \)

\( x_{q}^{l} \leq x_{q} \leq x_{q}^{u} \)

Stop when \( P_{q}(1) = P_{q,target} \) or \( F_{E_{q}}(1) = F_{E_{q},max} \)

Using \( p \)

Minimize \( \{ f_{1}(x_{q}), \ldots, f_{M_{q}}(x_{q}) \} \)

Subject to \( g_{j}(x_{q}) \geq 0 \; \forall \; j \)

\( h_{k}(x_{q}) = 0 \; \forall \; k \)

\( x_{q}^{l} \leq x_{q} \leq x_{q}^{u} \)

Stop when \( P_{q}(r) = P_{q,target} \) or \( F_{E_{q}}(r) = F_{E_{q},max} \)

Replication r

Problem q

Figure 4.1: Parameter tuning framework.

The main part of the framework is the design layer, where the upper-level tuner is located. In this thesis, only evolutionary algorithms are used as tuners, but in principle any optimization algorithms can be used. The parameter tuning problems on the upper level can be either single or multiobjective, formulated to find the best CPV tuple,
p, or, for multi-objective problems, the non-dominated set of CPV tuples. The framework uses minimization problems without loss of generality, since maximization problems can readily be converted to minimization problems. Each objective, \( \{U_1, \ldots, U_M\} \), is one of several performance indicators.

### 4.2.1 UPPER LEVEL PERFORMANCE INDICATORS

The choice of how to assess the utility of a CPV tuple determines the utility landscape and therefore also what the best CPV tuple is. There are several different ways to assess the performance of an optimization algorithm, for example, the number of function evaluations of a performance target, the fitness after a certain number of function evaluations, and the success rate. When these methods are aggregated over a number of runs, they are denoted as follows:

- Mean best fitness (MBF)
- Average number of solutions until a target is reached (AES)
- Success rate (SR)

All these methods require at least one predetermined value. For MBF, it is the budget given to the optimization algorithm and for AES it is the performance target, and possibly also a maximum budget to avoid spending too much computational effort on CPV tuples that may never reach the target. Selecting appropriate values for both MBF and AES can be difficult and will also affect the optimal CPV tuple. One advantage of AES over MBF is that it is easier to interpret the relative difference between the numbers of function evaluations than it is to understand the relative difference between fitness values.

MBF and AES cannot be used simultaneously since they use two different stopping criteria. They can, however, be used together with other indicators such as the runtime of the lower-level optimization, the number of generations, or other similar indicators.

### 4.2.2 EVALUATION OF A CPV TUPLE

The upper-level tuner evaluates the utility of a CPV tuple, p, by running a lower-level optimization with those parameters. This lower-level optimization can be a single- or multi-objective optimization problem, but it has to return a single performance measure. IGD and hypervolume are unary performance indicators that can be used for this purpose when the lower-level problem has multiple objectives. The lower-level optimization is run until the stopping criterion is met.

When AES is used, the optimization algorithm stops when the performance measure reaches the target (\( P_{q, \text{target}} \) for problem \( q \)) or when the maximum number of function evaluations (FEs) has been reached. Both the performance target and maximum FEs can be set individually for each problem. When the lower-level optimization stops, it determines the performance as the number of FEs used. If the lower-level optimization does not reach the target even after a predefined maximum number of evaluations (\( F_{E_{q, \text{max}}} \) for problem \( q \)), the difference between the performance and the performance target is assigned as a constraint violation. This allows the upper-level tuner to differentiate between two CPV tuples that both fail to reach the target performance.
In the case of MBF, the optimization simply stop when the maximum number of FEs has been reached and the performance of the CPV tuple is determined as the fitness value of the lower-level optimization.

Each CPV tuple is evaluated against all lower-level problems (Problem 1 to Problem q). Specialist CPV tuples are obtained when there is only a single lower-level problem, and the more problems the CPV tuples are tuned against, the more generalist they become. Each problem is replicated r times. Every problem that a CPV tuple is evaluated on receives the same number of replications. However, different CPV tuples might be replicated different numbers of times. For example, in the exploration phase of the upper-level optimization, a small number of replications could be used, while the number of replications increases later in the exploitation phase. Alternatively, heuristics together with statistical testing can be used to decide how to allocate replications over the entire set of CPV tuples. Common Random Numbers (CRNs) are used for all replications. This reduces the variance by using the same set of seeds for every CPV tuple. The drawback is that the control parameter values can be overfitted to a specific sequence of seeds.

The average performance, $U^+_i(p)$, of a CPV tuple is calculated for each upper-level objective when all replications for a problem instance are completed. The replication averages are then summed over all problem instances to form the final utility of each objective, $U_i(p)$. Those results are then returned to the tuner so that the CPV tuples can be compared with each other and the next generation of CPV tuples can be created. This cycle continues until the stopping criterion for the upper-level optimization is met. After the upper-level optimization is complete, the best CPV tuple(s), $p^*$, is(are) replicated on a set of post-replication seeds that differ from the seeds used during the tuning. The purpose of the post-replication step is to compare the best CPV tuples that may have emerged from different resampling strategies. Post-replication also helps identify CPV tuples that have been too specialized on the seeds used during experiments.

### 4.2.3 Parallelization Potential

DE and GAs, the two types of evolutionary algorithms used as tuners in this thesis, are both population-based algorithms. For population-based algorithms, the candidate solutions created in each generation are independent and can therefore also be evaluated in parallel. The size of each generation is determined by the offspring population size, so the parallelization potential of a generational population-based algorithm is therefore capped by it. However, every replication is independent of every other replication, meaning that all replications belonging to the same generation can also be performed in parallel. The parallelization capability in each generation is therefore,

$$max_{parallelism} = \lambda \ast r \ast q$$  \hspace{1cm} (4.1)$$

where $\lambda$ is the offspring population size, $r$ is the number of replications, and $q$ is the number of problems. Even for moderate values, the maximum parallelization can be quite substantial.

### 4.3 Software Architecture

The software architecture for implementing framework described above is shown in Figure 4.2. The main architecture and the accompanying design decisions made are explained in this section. The main design goals of the architecture are flexibility, scala-
bility, and efficiency. The architecture is flexible in the sense that its components can be combined in various ways to suit different requirements. The components can be combined into a single module that can easily be embedded in an application. Or alternatively, the components can be distributed over many different computers for increased computational power. The architecture can therefore also scale from running on a single computer to running on several hundred computers without any modifications. Inefficient design decisions might not be noticeable if the architecture is used only for small deployments. However, for large deployments, it is important that the amount of information being sent between different components be minimized and that the workload be distributed evenly over all computers to ensure that computational resources are maximally utilized.

Each solid box in Figure 4.2 represents one indivisible component. These components can be run independently of one another on different computers when needed. The dashed boxes each represent a logical grouping of components for the instantiation of the architecture used in this thesis. Each opt-manager therefore only manages the optimizations running on the same computer and, similarly, each worker-manager, manages only the workers running on the same computer. Another possibility would be to have a global opt-manager that handles every optimization instance in the network. However, such a configuration was not used as it is too centralized and the opt-manager could become a bottleneck in the system. The outermost dashed box represents an instance run on one computer. Each computer can have a different number of cores, $N_i$, where $i$
is the computer number. $\sum N_i$ determines the number of optimization and worker jobs that can be running simultaneously.

### 4.3.1 OPTIMIZATION TYPES

There are two types of optimizations in this architecture, direct evaluation and indirect evaluation. A direct evaluation optimization never sends any evaluations to the proxy-manager, but instead evaluates them directly in the same thread, for example, an algorithm being tuned on a test problem whose objective(s) is(are) extremely cheap to evaluate. Such test problems are much more time consuming to send to a worker for evaluation than they are simply to evaluate directly. Even if the problem is evaluated on the same computer (but in a different thread), the computational cost of serializing and sending the message through the different components would outweigh any asynchronous benefits. However, for more time-consuming objective function evaluations, it is better to send the job for evaluation asynchronously so that multiple jobs can be evaluated at the same time. This distinction has no impact on the framework or the experiment results. It is strictly a computational optimization that reduces the amount of work needed when the lower-level objective function(s) is(are) cheap to evaluate.

### 4.3.2 IMPLEMENTATION AND SOFTWARE LIBRARIES

The entire system is implemented in C++ using various software libraries. Communication between components is performed by message passing. Having well defined interfaces between components also helps reduce coupling between them. The two core external libraries that facilitate the message passing design are ZeroMQ (Distributed Messaging 2018) for the actual message passing and Protocol Buffers (Protocol Buffers 2018) for the serialization of messages. ZeroMQ is a lightweight and portable messaging library. It has native implementations in many different programming languages and wrappers for many more. It also runs on many different operating systems. It provides a thin abstraction layer over TCP/IP that helps it achieve a small overhead while simultaneously providing useful features such as different messaging patterns. ZeroMQ does not define a serialization protocol; its only job is to send bytes over a socket. Messages are instead defined and serialized with Protocol Buffers. Protocol Buffers started as an internal Google project but is now run as an open source project. Its goal is to provide a portable, language-neutral, and extensible way of serializing data. It is also available in many different programming languages and operating systems. Messages are defined in a special format that is fed to a compiler that creates code in a specified programming language. This code is then used to create, serialize, and deserialize messages. Since components communicate with one another by message passing, it does not matter whether the components are located in the same process space or half way around the world in two different data centers. This makes it possible for the system to be both flexible and scalable, and this would be much harder without libraries such as ZeroMQ and Protocol Buffers.

The system supports multiple operating systems and has been tested on Microsoft Windows and Linux. It works on most operating systems that have a C++ compiler and implementations of ZeroMQ and Protocol Buffers. The system also supports deploying components to different operating systems, so that computers running different operating systems can be combined into a single system.
4.3.3 INFORMATION FLOW

The arrows in Figure 4.2 represent all possible information flows and the labeled arrows represent an example information flow for a bilevel optimization tuning experiment. It starts with a user starting a bilevel optimization experiment by sending a start optimization request to the controller (A). The controller selects a suitable compute instance where the upper-level optimization will run and relays the request to that instance (B). The compute then relays the request to its opt-manager (C). The opt-manager will spawn a thread and start an optimization of the requested type on it (D). The upper-level optimization will create a CPV tuple and send it to the proxy-manager for evaluation (E). The proxy-manager will gather a list of all opt-managers that can start at least one more optimization and randomly select one where the lower-level optimization will run (F). The opt-manager will spawn an optimization of the correct type and start the optimization on it (G). The lower-level optimization will create a candidate solution for the problem in the application layer. For a direct evaluation optimization, the candidate solution is immediately evaluated and the lower-level optimization will continue to run until a stopping criterion met and the utility of the CPV tuple is returned to the upper-level optimization. For an indirect optimization, the candidate solution is sent to the proxy-manager (H). The proxy-manager will gather a list of all worker-managers that can run at least one more evaluation job and randomly select one where the evaluation will be sent (J). The worker-manager selects a free worker and sends the candidate solution for evaluation (K). The worker evaluates it and returns the result to the worker-manager.

4.3.4 CONTROLLER

The controller and storage are the only two components that are visible externally. The controller’s main purpose is to serve as the entry point for starting new optimizations. However, optimizations created as part of bilevel optimization experiments are not sent through the controller so as to avoid a single bottleneck point. It is also the only component that is cognizant of all available compute instances in the system. When a compute instance starts for the first time, it registers itself with the controller and regularly checks back, so that the controller can detect and remove unresponsive compute instances.

4.3.5 STORAGE

The storage is the other externally visible component and it is responsible for storing and retrieving information about an optimization experiment. It uses SQLite, which is a file-based SQL library that is both fast and portable. The advantage of using SQLite is that it is both small and easy to embed in an application, while also providing the benefits of the structured storage of SQL. The drawback is that it has more difficulty handling multiple concurrent users than do standalone SQL databases such as MySQL or Postgres. However, relying on an external SQL database would make the system less flexible and portable.

4.3.6 OPTIMIZATION MANAGER

The role of the opt-manager is to keep track of all running and idle optimizations. Each optimization is run on its own thread, which is owned and managed by the opt-manager. The opt-manager communicates with the optimization only through message passing.
The optimization is put into an idle state when it is first started. It stays in the idle state until it receives a start optimization request that includes the CPV tuple to be used for that particular optimization experiment. When the optimization is finished, the opt-manager is not destroyed but instead goes back into the idle state. This removes the overhead of spawning and destroying threads. It also avoids creating and destroying optimization algorithm objects such as sockets.

When an opt-manager receives a start optimization request, it first checks whether the optimization type is direct or indirect evaluation. The reason for distinguishing between optimization types is that a direct evaluation optimization uses more resources than does an indirect evaluation optimization. A direct evaluation optimization is therefore strictly limited to the number of processor cores on a given machine, while an indirect evaluation optimizations is allowed to exceed the number of processor cores. The opt-manager then checks the number of optimizations of that type that are already running, and if the maximum has already been reached, the optimization is queued. If the optimization is allowed to start, it looks for idle optimizations of the same algorithm already running. When it finds an idle optimization of the correct type it sends the start optimization request and the CPV tuple to it. The optimization is then removed from the idle list. If it cannot find an idle optimization of the correct type, it instead spawns a new thread running an optimization algorithm of that type and sends the request to it. When the optimization is completed, it is put back on the idle list.

4.3.7 WORKER MANAGER

The worker-manager is responsible for all workers located on the same machine. The worker-manager only communicates with its workers using message passing. Thus, a worker can be run in a thread on the same machine or in a separate process on a different machine. The purpose of a worker is to perform expensive computations such as the evaluation of a simulation model. When a worker is first started, it registers itself with the worker-manager. A worker regularly checks in with its worker-manager so that it can be removed if it becomes unresponsive, for example, if the simulation software crashes or gets stuck due to a problem in the model. Before an optimization is started a new session is created and distributed to all workers. The session contains information that does not change during the optimization, such as the simulation model. This helps keep the information sent during an optimization to a minimum. When a worker-manager receives a worker job, it sends it to the first available worker. If there are no workers available, the job is put into a queue and is sent as soon as another worker job finishes.

4.3.8 PROXY MANAGER

The purpose of the proxy-manager is to relay worker jobs and optimization jobs to a suitable opt-manager and worker-manager. The proxy-manager keeps track of all instances of opt-managers and worker-managers in the system, including the ones located on the same machine. This means that when a proxy-manager receives a job, it can determine whether the job is best run on the same machine or whether it should be sent to another machine in the system, based on the free workers available on each machine. The proxy-manager decides where a job should be run, whereas the opt-manager and worker-manager cannot resend jobs elsewhere. That is also why the proxy-manager communicates directly with opt-managers and worker-managers and not with other proxy-managers.
A proxy-manager only knows about its own state, and it shares no information with any other proxy-manager in the system. This decentralized design is intended to reduce the amount of data sent throughout the system. A consequence of this design is that a proxy-manager only knows about the jobs that it has distributed to the other opt-managers and worker-managers in the system. A opt-manager or worker-manager can therefore be assigned more jobs than it has processor cores. Each manager therefore has a queue of jobs where jobs are put that cannot be started right away. This helps reduce networking overhead, since this queue is local in each instance. The drawback of this decentralized design is that there is the possibility that a particular instance may be overburdened with jobs while another instance has no jobs. However, the risk of this happening is small since jobs are distributed randomly; also, has not proved to be a practical problem.

4.3.9 COMPUTE

The compute is a logical grouping of components (see Figure 4.2) providing a single contact point to the controller. It manages the life cycle of its components and relays start optimization requests to its opt-manager.
EVOLUTIONARY ALGORITHMS AS PARAMETER TUNERS
CHAPTER 5
EVOLUTIONARY ALGORITHMS AS PARAMETER TUNERS

There are many different ways of approaching the parameter tuning problem. Manually constructing and evaluating CPV tuples in an ad hoc manner is only feasible when a small number of them is to be tested. For larger tuning budgets, a more systematic approach is preferable. As shown by the framework described in the previous chapter, the approach taken in this thesis is to formulate the parameter tuning problem itself as an optimization problem and solve it with an optimization algorithm. This is a challenging optimization problem because of nonlinear objective functions, interacting variables, multiple local optima, and noise. However, it is on precisely this kind of difficult optimization problem that evolutionary algorithms have been shown to be effective (Eiben and Smit, 2011). It is not unreasonable to assume that evolutionary algorithms might need certain modifications to perform optimally as parameter tuning algorithms, since they are often developed and tuned for test problems. The aim of this chapter is therefore to investigate how a genetic algorithm and differential evolution perform compared with a state-of-the-art tuning method (irace) and how they can be modified to better handle the characteristics of parameter tuning problems.

Solving a parameter tuning problem requires computationally expensive experiments, since evaluating a CPV tuple requires that a complete optimization run be performed. The performance of the tuner is therefore important for the problem to be solved to a satisfactory level within a relatively small tuning budget. This thesis investigates the search strategy and noise handling of evolutionary algorithms and how they relate to the performance of solving the parameter tuning problem. The search strategy is controlled by changing the control parameter values of the tuner. Different configurations of the control parameter values of the tuners are tested in order to find the one that works the best for parameter tuning problems.

The noise handling of DE and GAs is the second aspect that will be examined. Optimization algorithms are usually stochastic, which means that multiple optimization replications might be needed to find a good and robust CPV tuple. Here, robustness refers to minimal variation in performance over multiple optimization runs. Performing replications will, however, reduce the number of unique CPV tuples that can be evaluated within a given tuning budget, so there is a trade-off between the quantity and quality of the solutions. Various resampling strategies will be tested to assess the importance of optimization replications.

DE and GAs are used as optimization algorithms on both the upper and the lower levels. In the rest of the chapter, the lower-level algorithm will simply be referred to as DE or GA and the upper-level algorithm will be referred to as upper-level DE or upper-level GA.
5.1 GENETIC ALGORITHM

A GA is an evolutionary optimization algorithm that mimics the process of natural evolution. It starts with a random population of individuals that evolved over time by cross-breeding and mutation. In each generation, a fixed number of new individuals is added to the population. At the end of each generation, the population is trimmed by sorting the individuals by their fitness and removing the worst ones. The crossover operator is SBX and the mutation is polynomial.

These are the parameters that are tuned for the GA:

1. Population size ($\mu$): The number of individuals in the population; an integer in the range $[10, 300]$.
2. Offspring population size ($\lambda$): The number of children created in each generation; an integer in the range $[10, 300]$.
3. Tournament size ($ts$): The number of tournament rounds, with a value of 1 indicating that parents are selected randomly from the population; an integer in the range $[1, 20]$.
4. Mutation probability ($pm$): The probability of random changes in the decision variables as $pm / N$, where $N$ is the number of decision variables; a real value in the range $[0, 5]$.
5. Mutation distribution index ($\eta_m$): Index governing the proximity of the mutated child to its parent, with larger values indicating a smaller change in the original value; a real value in the range $[0, 500]$.
6. Crossover probability ($pc$): The probability of creating offspring from two parents; a real value in the range $[0, 1]$.
7. Crossover distribution index ($\eta_c$): Index governing the proximity of the children to their parents, with larger values indicating that the children are closer to their parents; a real value in the range $[0, 500]$.

5.2 DIFFERENTIAL EVOLUTION

DE is a population-based evolutionary algorithm (Storn and Price, 1997). It starts by filling the population with a random set of individuals. A new individual is then created by adding the weighted difference between two individuals in the population to a third individual. This new individual is then recombined with a fourth individual, and it replaces the parent if it has better fitness, otherwise it is discarded. All parents are selected randomly from the population.

These are the parameters that are tuned for DE:

1. Population size ($\mu$): The number of individuals in the population; an integer in the range $[10, 300]$.
2. Offspring population size ($\lambda$): The number of children created each generation; an integer in the range $[10, 300]$.
3. Differential weight ($F$): Controls the amplification of the differential variation; a real value in the range $[0, 5]$.
4. Crossover probability ($pc$): The probability of accepting the differential perturbation for each decision variable; a real value in the range $[0, 1]$. 
5.3 NOISE HANDLING IN EVOLUTIONARY ALGORITHMS

Since evolutionary algorithms contain stochastic factors, they will not produce the same results when run with different seeds. This is something that needs to be considered when they are tuned. There are multiple ways of handling this noise. The simplest method is to explicitly average a CPV tuple by running the optimization multiple times with different seeds. Population-based methods might also benefit from implicit averaging, since they sample solutions from a local area (Jin and Branke, 2005). However, resampling a CPV tuple with a fixed budget means that fewer unique tuples can be explored. Handling this trade-off between replications and testing new CPV tuples is an important aspect of a parameter tuner.

The simplest resampling strategy is always to replicate each CPV tuple the same number of times, in what is referred to as the static resampling strategy. If there are too few static replications, there is a possibility of selecting greedy parameters, whereas too many replications will limit the number of unique CPV tuples that can be explored, which could prevent the tuner from converging to the optimal CPV tuple. The advantage of this strategy is that it is simple to understand and implement. However, there are circumstances in which a more flexible strategy could be beneficial. For example, early in the optimization it might not be as important to conduct replications as it is in the later stages. Also, a flexible resampling strategy could detect and discard poorly performing CPV tuples before spending too much replication effort on them.

The bilevel approach used in this thesis always runs at least one replication for each problem instance. The resampling strategy determines only the number of replications, not what problem instances to run. For parameter tuning scenarios with many problem instances, this will result in a large number of minimum optimizations per parameter configuration. In such scenarios, it might be more efficient to remove poorly performing parameter configurations based on a subset of problem instances. That is the approach used by irace. Problem instances are selected, in order, from this list when additional samples are needed. When the end of the list is reached, irace starts over from the beginning of the list, but with a different seed. The drawback of this approach is that the order of the problem instances can cause otherwise good parameter configurations to be discarded. For example, in a parameter tuning scenario with 100 problem instances, a parameter configuration that performs well on 95 problems and poorly on the remaining 5 could be discarded if by chance the problem instances on which it performs poorly are selected first.

When comparing tuning experiments that use different resampling strategies, it is important that they be compared in a fair way. One way to do this is to perform post-replications of the CPV tuple found by the tuner and to use the performance value obtained from the post-replications as the utility of the selected CPV tuple. If the same seeds are used for all post-replications, it does not matter how many replications the tuner used during the search, since that value is not used as the final utility.

5.4 EXPERIMENTAL DESIGN

This section will present the experiments performed to evaluate DE and GA as upper-level tuners. See Paper VII for additional experimental details and the complete results. Each of these two algorithms has two different sets of CPV tuples. One set consists of
the recommended values for each algorithm, which tend to be more exploitative, so the other set is chosen to promote more exploration. These sets will be combined in two ways: (i) only exploitation, and (ii) exploration followed by exploitation. This allows analysis of the usefulness of having an explicit exploration strategy. The latter strategy switches from exploration to exploitation when 75% of the tuning budget has been used.

The trade-off between replications and testing new CPV tuples is investigated by running experiments with 1, 5, 10, 15, and 20 static replications. Two dynamic resampling strategies are also included. The first is a step-resampling strategy that starts with 5 replications and then switches to 20 replications after 75% of the tuning budget has been spent. The second dynamic resampling strategy is time based. It starts with 1 replication and then, as the optimization progresses, gradually increases the number so that the last generation uses 20 replications. The number of samples for each individual \( x_s \) is calculated for each generation, as shown in Equations 5.1 and 5.2, where \( p \) is the progress, \( B_t \) is the total budget, \( B \) is the used budget, \( b_{\text{min}} = 1 \) and \( b_{\text{max}} = 20 \) are the minimum and maximum numbers of replications, and \( \lambda \) is the population size.

\[
p = \frac{B}{B_t - (b_{\text{max}} \times \lambda)} \tag{5.1}
\]

\[
x_s = p \times (b_{\text{max}} - b_{\text{min}}) + b_{\text{min}} \tag{5.2}
\]

The tuners are evaluated on their ability to tune both specialists and generalists on standard test functions. For the specialists, these are DE on the Ackley test function and GA on the Ellipse test function, and for the generalists these are DE and GA on the Ackley, Ellipse, Sphere, Griewank, Rosenbrock, and Rastrigin test functions. A mixture of single- and multi-modal optimization problems is chosen to provide a variety of challenging tuning problems. The configuration for each problem is shown in Table 5.1. The combination of DE on the Ackley test function is chosen specifically to test the ability of the tuner to find robust CPV tuples. This combination is challenging because a greedily configured DE can sometimes solve the multi-modal Ackley test function problem very fast. However, for other starting conditions it can become stuck in local optima without the ability to escape. The combination of GA on the Ellipse test function is meant to provide a tuning problem that has multiple local optima. That is, there are CPV tuples that are relatively easy to find and result in good performance; however, there are other significantly different CPV tuples that perform even better.

Table 5.1: Problem configurations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Dimension</th>
<th>Range</th>
<th>Performance Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ackley</td>
<td>20</td>
<td>[-10, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Ellipse</td>
<td>20</td>
<td>[-10, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>20</td>
<td>[-5, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Rastrigin</td>
<td>10</td>
<td>[-10, 10]</td>
<td>0.1</td>
</tr>
<tr>
<td>Sphere</td>
<td>20</td>
<td>[-10, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Griewank</td>
<td>20</td>
<td>[-600, 600]</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 5.2: Experimental configuration for the upper-level GA.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explorative</th>
<th>Exploitative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Offspring population size</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Mutation distribution index</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>Crossover distribution index</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Tournament size</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.3: Experimental configuration for the upper-level DE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explorative</th>
<th>Exploitative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Offspring population size</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Differential weight</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.1</td>
<td>0.9</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td>Mutation distribution index</td>
<td>0</td>
<td>20</td>
</tr>
</tbody>
</table>

The two algorithms (DE and GA) are combined with the seven resampling strategies (i.e. step, time, static 1, static 5, static 10, static 15, and static 20) and two sets of upper-level CPV tuples (i.e. exploitation only and ii. exploration 75% + exploitation 25%), plus irace, for a total of $2 \times 7 \times 2 \times 1 = 29$ configurations per experiment. Each configuration is replicated 20 times for each experiment. The two specialist parameter tuning problems are run with tuning budgets of 5000, 10,000, and 20,000, whereas the two generalist parameter tuning problems are run with tuning budgets of 30,000 and 60,000, for a grand total of $29 \times 20 \times (2 \times 3 + 2 \times 2) = 5800$ parameter tuning experiments. To compare experiments with different resampling strategies, the best CPV tuple from each experiment is replicated 100 times with a set of seeds that was not used during the experiments.

The two CPV tuples for the upper-level GA are shown in Table 5.2. The corresponding CPV tuples for upper-level DE are shown in Table 5.3. The version of irace used in the experiments is 2.3.1806 and its parameters are kept unchanged at their default values. There was no attempt to tune irace, so any comparisons with it are only valid for this particular configuration.
Table 5.4: Combined ranking of upper-level tuners.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Algorithm</th>
<th>Replication</th>
<th>Configuration</th>
<th>Total Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DE</td>
<td>step</td>
<td>std</td>
<td>27315.5</td>
</tr>
<tr>
<td>2</td>
<td>DE</td>
<td>time</td>
<td>std</td>
<td>32921.5</td>
</tr>
<tr>
<td>3</td>
<td>DE</td>
<td>10</td>
<td>std</td>
<td>37866.5</td>
</tr>
<tr>
<td>4</td>
<td>DE</td>
<td>step</td>
<td>exp</td>
<td>39060.0</td>
</tr>
<tr>
<td>5</td>
<td>DE</td>
<td>5</td>
<td>exp</td>
<td>40359.5</td>
</tr>
<tr>
<td>6</td>
<td>DE</td>
<td>5</td>
<td>std</td>
<td>41902.5</td>
</tr>
<tr>
<td>7</td>
<td>GA</td>
<td>step</td>
<td>exp</td>
<td>45930.0</td>
</tr>
<tr>
<td>8</td>
<td>DE</td>
<td>10</td>
<td>exp</td>
<td>46926.0</td>
</tr>
<tr>
<td>9</td>
<td>DE</td>
<td>time</td>
<td>exp</td>
<td>47499.5</td>
</tr>
<tr>
<td>10</td>
<td>GA</td>
<td>5</td>
<td>std</td>
<td>50281.0</td>
</tr>
<tr>
<td>11</td>
<td>GA</td>
<td>step</td>
<td>std</td>
<td>53732.0</td>
</tr>
<tr>
<td>12</td>
<td>GA</td>
<td>10</td>
<td>exp</td>
<td>53874.5</td>
</tr>
<tr>
<td>13</td>
<td>GA</td>
<td>time</td>
<td>exp</td>
<td>55051.0</td>
</tr>
<tr>
<td>14</td>
<td>GA</td>
<td>1</td>
<td>exp</td>
<td>91443.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Algorithm</th>
<th>Replication</th>
<th>Configuration</th>
<th>Total Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>DE</td>
<td>20</td>
<td>std</td>
<td>59322.5</td>
</tr>
<tr>
<td>16</td>
<td>GA</td>
<td>time</td>
<td>std</td>
<td>60300.0</td>
</tr>
<tr>
<td>17</td>
<td>DE</td>
<td>15</td>
<td>exp</td>
<td>60530.0</td>
</tr>
<tr>
<td>18</td>
<td>GA</td>
<td>10</td>
<td>std</td>
<td>63002.0</td>
</tr>
<tr>
<td>19</td>
<td>GA</td>
<td>15</td>
<td>exp</td>
<td>67303.5</td>
</tr>
<tr>
<td>20</td>
<td>GA</td>
<td>15</td>
<td>std</td>
<td>68367.5</td>
</tr>
<tr>
<td>21</td>
<td>DE</td>
<td>20</td>
<td>exp</td>
<td>68596.0</td>
</tr>
<tr>
<td>22</td>
<td>DE</td>
<td>1</td>
<td>exp</td>
<td>69891.5</td>
</tr>
<tr>
<td>23</td>
<td>GA</td>
<td>20</td>
<td>std</td>
<td>70000.0</td>
</tr>
<tr>
<td>24</td>
<td>GA</td>
<td>20</td>
<td>exp</td>
<td>70914.0</td>
</tr>
<tr>
<td>25</td>
<td>DE</td>
<td>1</td>
<td>std</td>
<td>71178.5</td>
</tr>
<tr>
<td>26</td>
<td>GA</td>
<td>5</td>
<td>std</td>
<td>72216.0</td>
</tr>
<tr>
<td>27</td>
<td>GA</td>
<td>1</td>
<td>exp</td>
<td>72496.5</td>
</tr>
<tr>
<td>28</td>
<td>GA</td>
<td>1</td>
<td>std</td>
<td>91443.0</td>
</tr>
<tr>
<td>29</td>
<td>irace</td>
<td></td>
<td></td>
<td>93314.0</td>
</tr>
</tbody>
</table>

5.5 RESULTS

This section will present the experimental results of the evaluation of DE and GA as upper-level tuners.

Table 5.4 presents the combined ranking of all replications and experiments. DE stands out as the best overall tuner for the set of parameter tuning problems used in this study. The best-performing DE configurations are the ones with the dynamic strategy and standard parameters. The configurations with 5 and 10 static replications also perform well. DE does not seem to benefit from exploration parameters, since the configurations with exploration parameters either perform similarly to or worse than the configurations using purely exploitative parameters.

The best GA configuration is the one with the step-resampling strategy and the exploration parameters. As in the case of DE, the best numbers of static replications are 5 and 10. The explicit exploration and exploitation strategy is beneficial for the GA, since configurations with only exploitation either have similar or worse performance as the configurations with the exploration strategy.

For both DE and GA, the best resampling strategies are the dynamic ones, followed by static 5 and static 10. Using a single static replication is among the worst strategies,
likely because the parameters found are not robust enough when they are post replicated. On the other hand, when using 15 and 20 static replications the performance also declines, probably because there are not enough unique evaluations for the evolutionary algorithm to properly explore the parameter space. It is also clear that both DE and GA significantly outperform irace in most configurations.

Figure 5.1: The performance of each upper-level configuration when tuning DE on the Ackley test function.

Figures 5.1-5.4 show the performance of the upper-level configurations for each of the parameter tuning problems. The utilities of the best CPV tuple from each of the 20 experiment replications are shown as box plots, with the utility of each experiment replication being the average of the 100 post-replication experiments of the best CPV tuple found for each configuration. The configurations are sorted in ascending order from left to right by their mean performance. The configurations shown in bold are statistically indistinguishable from one another according to Kruskal-Wallis testing followed by Tukey pairwise comparison, both performed with significance of 0.05.

The first part of the configuration name is the upper-level algorithm used. For GA and DE, the following two values denote the resampling strategy and parameter configuration, respectively. The value following $F$ is the number of optimizations out of the 2000 optimizations (20 experiment replications $\times$ 100 post-replications) that failed to reach the target performance within the maximum budget. Finally, $E$ is the total number of unique CPV tuples that was evaluated.

The results of the experiments of tuning DE on the Ackley test function are shown in Figure 5.1. From the results it can be seen that on this parameter tuning problem, the configurations that favor replications over unique CPV tuples perform better than do the configurations that do the opposite. The dynamic resampling strategies perform well with
both tuning budgets, but especially so for the more limited budget of 5000. Using only 1 or 5 static replications is clearly not enough for this parameter tuning problem, since all of those configurations have a high failure rate. However, even the best-performing configurations have some failed runs. It can also be noted that good results can be obtained on this parameter tuning problem using only a few hundred unique evaluations.

Figure 5.2: The performance of each upper-level configuration when tuning the GA on the Ellipse test function.

The results of the experiments of tuning GA on the Ellipse test function are shown in Figure 5.2. The results of this parameter tuning problem are quite different than from the previous problem. The best configurations from these experiments are the ones that use few replications. For the smaller tuning budget of 5000, the best configurations use 1 and 5 static replications. More configurations perform well with the higher tuning budget, especially the dynamic resampling strategies and static 5 and 10. The results indicate that the evolutionary algorithms need at least 1000 unique evaluations to have a good chance of finding the optimal CPV tuple. On this particular parameter tuning problem, there is little to no benefit of exceeding 5 static replications, and with the smaller tuning budget, 1 optimization replication is even sufficient. Another observation is that the control parameters of the GA are important for this problem, since the configurations that use the exploration strategy significantly outperform the configurations that do not. The cause is that the standard parameters more often lead to local optima. DE, however, is neither helped nor hindered by the exploration strategy.

The results of the experiments of tuning GA as a generalist are shown in Figure 5.3. The results of tuning GA as a generalist are similar, but not identical, to the results of tuning GA as a specialist. Configurations that favor evaluating more unique CPV tuples over more replications perform better. One difference from the specialist experiment
Figure 5.3: The performance of each upper-level configuration when tuning the GA as a generalist.

is that 1 static replication is not enough, since those configurations have a significantly higher failure rate. This is true for both the smaller and larger tuning budgets. Another similarity is that the GA configurations with the exploration strategy perform better than those with only standard control parameters.

The results of the experiments of tuning DE as a generalist are shown in Figure 5.4. As with tuning DE as a specialist, the configurations that favor replications over testing new CPV tuples perform better. However, the performance of 20 static replications is worse than in the specialist experiments. This is probably because it only has 500 unique CPV tuples, which seems to be too few for this parameter tuning problem.
Figure 5.4: The performance of each upper-level configuration when tuning DE as a generalist.
USING PARAMETER TUNING TO IMPROVE OPTIMIZATION ALGORITHMS
CHAPTER 6
USING PARAMETER TUNING TO IMPROVE OPTIMIZATION ALGORITHMS

This chapter presents how bilevel optimization for parameter tuning can be used to improve different optimization algorithms. The knowledge gained from these experiments casts light on how various algorithms should be configured for specific and general sets of problems.

6.1 SPECIALIST VERSUS GENERALIST PARAMETERS

Extracting algorithmic design decisions as control parameters does not circumvent the no free lunch theorem. It is merely shifted from the trade-off between optimization algorithms to the trade-off between CPV tuples, since no CPV tuple can be better than all other CPV tuples on all problems. The aim of this section is to explore the performance difference between the specialist and generalist CPV tuples of the CMA-ES algorithm and to evaluate the control parameter heuristics of CMA-ES versus those found with tuning (see Paper I for more details).

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (Hansen and Ostermeier, 2001; Hansen, 2006) is a single-objective evolutionary optimization algorithm that for certain problems outperforms other evolutionary algorithms, such as genetic algorithms, differential evolution, and particle swarm optimization (Hansen and Kern, 2004; Hansen, Ros, et al., 2011; Hansen, Auger, et al., 2010). CMA-ES generates new search points by sampling them from a multi variate normal distribution. A multi variate normal distribution is determined by its mean, \( m \in \mathbb{R}^n \), standard deviation, \( \sigma \in \mathbb{R}^n \) and covariance matrix, \( C \in \mathbb{R}^{n \times n} \). By modifying the covariance matrix, the search distribution is made to fit the contour lines of the objective function, thereby increasing the probability of generating good solutions.

CMA-ES has heuristics for all of its parameters. The parameter values are set mostly based on the number of dimensions of the problem. The heuristics are derived from both empirical studies and the inherent properties of CMA-ES. They are also designed to be effective over a diverse set of problems. The aim of this section is to investigate how good these heuristics are at selecting the optimal parameter values for the CEC'15 expensive problems (Chen et al., 2014), a set of 15 single-objective optimization test problems to be solved for 10 and 30 dimensions.
6.1.1 RESULTS

Table 6.1 shows the best, median, and mean utilities from each experiment. CMAES-S and CMAES-G denote the specialized and generalized parameter experiments, respectively. The scores, as calculated by the objective function in Equation (6.1), are shown.

Table 6.2: The score as calculated by the objective function in Equation (6.1). The specialized parameter score (CMAES-S) is the summation of all individual results.

Equation (6.1)
in Table 6.2. For CMAES-S, the values are the summation of all individual problems. Therefore, the difference between CMAES-S and CMAES-G is the performance gain (or loss) from allowing each test problem to use specialized instead of generalized parameters.

The specialized parameters can improve the performance by 81% for the median and 128% for the best results. This indicates that the no free lunch theorem holds for the parameters of CMA-ES and the CEC'15 problems, because no set of parameters optimal across all problems could be found. From the results it is also clear that functions 1, 2, and 10 are the most difficult to optimize. It is reasonable to assume that those functions will influence the general parameters the most, because of the formulation of the objective function.

The optimized parameter values can improve on the parameter values selected by the heuristic in most of the CEC'15 problems. Functions 1, 2, and 10 are the ones that improved the most. For the total score of all problems, the optimized parameters improved the performance by a factor of almost 32.

Small values for the population size parameter, $\lambda$, lead to fast convergence and large values help in avoiding local optima. Compared with the value selected by the heuristic, the parameter tuning results indicate that a smaller $\lambda$ provides better performance on the CEC'15 problems.

### 6.2 Evaluating Heuristics Using Parameter Tuning

Selecting appropriate values for the control parameters without extensive knowledge of the optimization algorithm and problem can be difficult. This is why most control parameters have recommended values that can be used for new problems. Sometimes there is also a heuristic or rule of thumb that can be used to select values based on the characteristics of the problem at hand. The aim of this section is to use parameter tuning to evaluate a heuristic that says that the mutation probability in a GA should be set to $p_m = \frac{1}{N}$, where $N$ is the number of decision variables. For additional details, see Papers II and IV.

Many real-world optimization problems are designed to be scalable with respect to the variables. For example, consider a production line involving several machines in which their processing times have to be optimized to maximize the overall throughput and minimize the work in process in the line. Adding additional operations (machines) to such a line is equivalent to scaling the original optimization problem, since the objectives remain the same. In such a situation, it is beneficial to study how the optimal parameter values for the algorithm change depending on the problem size.

It is worth mentioning here that the goal of this section is not to find parameter settings that work across a range of problems, but to study how the optimal parameters vary for a given problem with the number of variables and budget size. To achieve this, several experiments will be performed on each problem to obtain multiple sets of optimized CPV tuples.

Each experiment was run with both NSGA-II and NSGA-III on the ZDT test problems (Zitzler, Deb, and Thiele, 2000) using five (i.e. 5, 10, 20, 30, and 40) different values for the number of decision variables. This was done to test the validity of the rule of thumb that the mutation rate should be inversely proportional to the number of decision variables. Since each experiment was also run with different function evaluation budgets, it is also
possible to study its effect on the mutation probability. The usefulness of this evaluation is limited by the small number of problems used in this section, so no generalization can be made about how this rule works for other problems. The mutation probabilities are also only taken from the best set of parameters found. Therefore, this evaluation does not test the validity of this rule of thumb for suboptimal sets of parameters.

6.2.1 RESULTS

The mutation probabilities of the best CPV tuples with respect to the number of decision variables are shown here for two problems, ZDT1 in Figure 6.1 and ZDT4 in Figure 6.2. Each line in the figures represents the best CPV tuple found with a particular function evaluation budget. The experimental results can be divided into two groups based on the relationship between the mutation probability and the number of variables. ZDT{1, 2, 3} are in one group and ZDT{4, 6} are in the other. The first group uses a relatively high mutation probability for two variables, which then significantly decreases and remains almost constant for 10, 20, 30, and 40 variables. The second group has a more gradual decrease in mutation probability that closely resembles the curve suggested by the mutation probability rule of thumb.

The rule of thumb suggests a proportional decrease in the mutation probability when the number of decision variables is increased. The results, however, indicate that this is only partly true for ZDT1. The mutation probability does decrease with an increase in the number of decision variables from 2 to 10. The rule of thumb slightly overestimates
the mutation probability for 10 and 20 decision variables, but is more accurate with 30 and 40 variables. The exception is with a budget size of 100, where the rule of thumb underestimates the mutation probability.

On ZDT4, the rule overestimates $p_m$ for two variables when the budget size is greater than 500; it also underestimates $p_m$ for all $N$ when the budget size is 100. For all other cases, the rule of thumb agrees well with the experimental results.

Based on these results, it can be concluded that the rule of thumb can estimate good values for the mutation probability, especially for larger budgets, on the ZDT test problems.

6.3 MULTIPLE PARAMETER SETS

The parameter values of evolutionary optimization algorithms are generally kept constant throughout the optimization run. However, studies have demonstrated that this is suboptimal and that the efficiency of the optimization algorithm can be increased by adapting the parameters during the course of the optimization (Bäck, 1992; Hesser and Männer, 1990). It is difficult to theoretically derive the optimal adaption of the parameter values for anything but the simplest problems, since the parameters can interact in complex ways. The control mechanisms are therefore usually inspired by concepts in statistics and nature, in the same way as many optimization algorithms are. Since it is difficult to understand the interactions between the parameters, parameter control mechanisms usually only consider a few parameters at most; they will therefore not use or exploit the more complex beneficial interactions between parameters.

Instead of finding a control mechanism that directly adapts the parameters, the novel empirical approach in this section is to estimate the optimal adaption of the parameters by tuning multiple sets of parameters. In other words, the optimization run is divided into stages, each of which has separate, static parameter settings. These multiple parameter sets, together with the stage separation points, are then formulated into an optimization problem and solved using an evolutionary algorithm.

This study has two main objectives. The first is to estimate the potential performance improvement with the use of adapting parameters, by tuning one, two, and three multiple parameter sets. This will give a lower bound for an optimal parameter control mechanism on each problem that is tuned. The second is to analyze the optimized parameter sets for patterns and other knowledge that can be used to improve the optimization algorithm, or its parameter values, when applied to a particular problem. It is also important to note that the aim is not to find generally applicable parameters (see Paper III for more details).

The experiments presented in this section will tune the parameters of NSGA-II when applied to the first four problems in the ZDT (Zitzler, Deb, and Thiele, 2000), DTLZ (Deb, Thiele, et al., 2002), and WFG (Huband et al., 2005) test suites. These problems provide a good range of difficulty, with and without local fronts. The ZDT and WFG problems are configured as two-objective problems and the DTLZ problems are configured as three-objective problems.

There are several possible strategies for determining when to switch between parameter sets. The simplest is to keep the switching points static, by dividing the optimization run into equally sized parts. This means that the switching points would be the same across all problems. However, it is unlikely that the optimal switching points will be the same for all problems. Another option is to switch after a certain number of function evaluations have been performed. The drawback of that approach is that it forces the
switching to happen at the same time for every replication. The approach taken in this section is instead to switch parameter sets when the IGD falls below a certain value. This threshold IGD value is added as a variable to the upper-level optimization, so that it can differ from problem to problem. This also means that there are \( s - 1 \) switching variables for \( s \) number of parameter sets.

### 6.3.1 RESULTS

The numbers of FEs needed to reach the target IGD for each of the 100 optimization replications are shown as box plots. Figures 6.3 and 6.4 show the results for the ZDT and DTLZ problems, respectively. In all problems, there is a significant reduction in the number of FEs from the default CPV tuple to the tuned CPV tuples. The reduction ranges from 31% to 95%, partially explained by fact that tuned parameters are specialized for each problem. Nevertheless, it is further evidence that the right parameter setting can greatly affect the performance. The default CPV tuple is unable to reach the target IGD for ZDT4 in any of the 100 replications, as can be seen in Figure 6.3.

However, it is also clear that not all problems benefit from the use of an additional parameter set, which is even more evident in the results for three parameter sets. Only three of the problems, i.e. ZDT1, WFG2, and WFG4, have significant reductions. The other problems are either worse or not statistically different from either one or two parameter set(s). In theory, the performance should never worsen when increasing the number of parameter sets, because a larger number of parameter sets could always exactly mimic a smaller number. In practice, worsening can happen because there is a stochastic algorithm in the upper layer that can fail to find the optimal parameter setting. In fact, this is more likely to happen with more parameter sets, since the upper-level problem becomes harder. This can be observed in the results.

The results indicate that significant improvement can be achieved with the use of two
parameter sets on most problems. However, there were also problems that failed to display any improvement, indicating that for these algorithm-problem combinations, there are no discernible distinctions between different phases of optimization. In other words, parameters that are optimal at the beginning of the search are also optimal towards the end. However, for problems that displayed an improvement with two parameter sets, the first set of parameters was observed to promote exploration while the second set favored exploitation. For example, the first parameter sets had relatively high population sizes, high mutation probabilities, and low mutation distribution indices. Conversely, the second sets had lower population sizes, lower mutation probabilities, and higher mutation distribution indices. Only two problems displayed a significant advantage of using three parameter sets, i.e., WFG2 and WFG3. This could mean that either there were better parameter settings that upper-level optimization failed to find, or that there was limited use for three or more parameter sets. More experiments are needed to empirically demonstrate the latter. However, the studies in this section indicate that most of the advantage of adapting parameters can be had with the use of only two parameter sets.

6.4 THE TRADE-OFF BETWEEN RUNTIME AND EVALUATION EFFICIENCY

The performance of an optimization algorithm is usually measured in terms of the number of FEs needed to reach a predetermined performance target or, alternatively, the quality of the solution(s) after a maximum number of FEs. Here, this performance aspect is referred to as the function evaluation efficiency, or simply the efficiency, of the optimization algorithm. Another performance aspect is the runtime of the optimization. The runtime and the number of FEs are proportional to each other when no parallelization is utilized, and when the computational overhead of the optimization is low compared with a function evaluation. However, the relationship between the runtime and efficiency of an optimization algorithm is more complex when parallel computing resources are utilized. The aim of this section is to investigate this relationship for two EAs. Additional details and results can be found in Paper V.

The runtime of a population-based EA that only evaluates a single individual at a time can be estimated as a product of the CPU time used by each FE ($t_{FE}$), the size of the offspring population ($\mu$) (i.e., the number of FEs in each generation), and the number of generations ($G$):

$$RT_s = t_{FE} \times \mu \times G$$

(6.2)

In a parallel computing environment, if the number of available parallel computing nodes
is set to match the size of one generation, then the runtime of the optimization is estimated to be only the product of how long an FE takes and the number of generations:

$$RT_p = t_{FE} \times G$$  \hspace{1cm} (6.3)$$

In other words, given a fixed number of generations, the runtime of a population-based EA in a parallel computing environment can be reduced by a factor of $\mu$. Furthermore, increasing the offspring population size (from $\mu$ to $\mu'$ where $\mu' > \mu$) will allow more FEs to be performed in parallel in each generation. A higher parallelization potential, i.e. more computing nodes running in parallel in each generation, can further reduce the runtime ($RT_p$) because the total number of generations (i.e. from $G$ to $G'$ where $G' < G$) can be reduced. However, increasing the offspring population size could also have a negative impact on the efficiency. In other words, an algorithm configured to be optimal in terms of efficiency is not guaranteed to be optimal in terms of runtime. This gives rise to an unknown, complex relationship between total runtime and FE efficiency in a parallel computing environment, because minimizing runtime by using more parallel runs in each generation does not guarantee that the efficiency will be maximized. At some point, the efficiency of the optimization is actually compromised when the parallelization potential to reduce runtime ($RT_p$) is increased.

The aim of this study is to find this trade-off front for NSGA-II and DE when applied to DTLZ1, DTLZ2, and discrete-event simulation problem. Each problem has two versions with different numbers of decision variables, so as to identify any differences in the trade-offs with different problem difficulties.

6.4.1 RESULTS

All non-dominated solutions from all experiments are presented in Figure 6.6. Each subfigure contains the front for both NSGA-II and DE on the respective problems. This allows for performance comparison between NSGA-II and DE with the best CPV tuples found for each algorithm. For each problem, the non-dominated solutions from the DE experiments all dominate the non-dominated solutions from the NSGA-II experiments. Thus, it is clear that DE is superior as both a specialist and a generalist, for the problems included in this section.

The results also indicate that there are clear knee regions on all problems for both NSGA-II and DE. Outside this region, small gains in one objective lead to large sacrifices in the other. The NSGA-II knee regions are less pronounced than those of DE. The efficiency of DE does not seem to be significantly affected by the offspring population size, at least up to the point where only a handful of generations are needed to reach the performance target. After this point, the efficiency drops off significantly.

Figure 6.7 shows all the non-dominated CPV tuples for DE on DTLZ1. As expected, it is clear that the offspring population size largely controls the trade-off between the number of FEs and the number of generations. The values for the other parameters are within a narrow range of their optimal values, with some exceptions. Some differences can also be observed between the experiments with 7 and 14 variables. The differential weight, $F$, is close to 0.5 for 7 variables, but is either 0 and 1 for 14 variables. The crossover probability is higher, but the mutation probability is lower, for the problem with 7 variables.

Figure 6.8 shows the optimal CPV tuples for NSGA-II on DTLZ2. There are two distinct groups of CPV tuples that split the non-dominated front in two parts, each of which has very different optimal CPV tuples. The first group has a higher $p_m$, a ts of one, a much
higher $\eta_c$ and low $\eta_m$. The first group also has both higher population sizes and higher offspring population sizes. The reason for these two distinct groups of CPV tuples is hard to determine without further experiments. There is also a correlation between the population size and offspring population size, in that the optimal population size also increases as the offspring population size increases, but not proportionally.

6.5 THE TRADE-OFF BETWEEN RUNTIME AND COST IN A CLOUD COMPUTING ENVIRONMENT

When the stopping criterion of a black-box optimization algorithm is a particular performance target, the runtime of the algorithm is simply the wall-clock time it takes for the optimization to obtain that performance target. For those cases, the runtime of the algorithm is strongly related to the number of solutions it can evaluate in parallel. For evolutionary algorithms, the parallelism is directly controlled by the offspring population size. Increasing the offspring population size will therefore generally reduce the runtime of the algorithm. However, increasing the offspring population size will also alter the search strategy, and that can cause the algorithm to become less efficient in terms of the number of function evaluations it requires. The reason the algorithm can
become less efficient is that the optimal value of the offspring population size is associated with the probability of creating solutions that lie on the trajectory towards the global optimum (Jansen, De Jong, and Wegener, 2005). When the offspring population is larger than the optimal value, the probability of creating solutions outside this trajectory is increased, which can in turn result in wasted computational effort.

A trade-off therefore arises between the runtime and the function evaluation efficiency for the different offspring population sizes of an optimization. Increasing the offspring population size will reduce the runtime but at the same time increase the number of function evaluations needed. However, this relationship is probably not linear, as there will be diminishing returns from increasing the offspring population size beyond a certain point.

Recent advances in cloud computing services have made cheap computing resources readily available for anyone to utilize. Resources can be created on demand and destroyed when no longer required. This reduces the need to own large clusters of physical computers to solve optimization problems quickly. The trade-off between runtime and cost is particularly relevant in this context, because the computing resources can be exactly matched to the offspring population size and, assuming per-second-billing, the cost is simply:

\[ \text{cost} = r \times i_n \times i_c \]  

(6.4)
Knowledge of the trade-off between runtime and cost allows the practitioner to make an informed decision on whether to pay more for quick results or wait longer and pay less. However, this trade-off is largely unknown for evolutionary algorithms. The aim of this section is to close some of that knowledge gap by experimentally investigating this trade-off for two evolutionary algorithms, i.e. NSGA-II (Deb, Pratap, et al., 2002) and DE (Storn and Price, 1997), using a master-slave architecture when applied to a discrete-event simulation (DES)-based optimization problem. Both steady-state asynchronous and generational versions of these algorithms are included in this study.

An important factor in these experiments is that the optimal values for most other control parameters might change for different offspring population size values. This issue will be addressed by performing parameter tuning experiments against the offspring population size together with the other control parameters. A bilevel optimization is used where the upper-level optimization decision variables are the control parameter...
values of the lower-level optimization algorithm and the upper-level objectives are to minimize the runtime and cost of the optimization. The lower-level objective is to minimize the number of function evaluations needed to reach a performance target.

The approach is to perform parameter tuning experiments using a simplified version of the DES problem. A subset of the solutions with the best control parameter values (including offspring population size) from those experiments will be replicated on Digital Ocean’s cloud computing platform (for additional details, see Paper VI).

### 6.5.1 RESULTS

The results of the parameter tuning experiments, for all four algorithms, are shown in Figure 6.9. Figure 6.9a shows the cost and time trade-off and Figure 6.9b shows the evaluations against time. Figure 6.9a shows that the steady-state asynchronous algorithms provide a better trade-off, since they completely dominate the generational algorithms. DE is also clearly better than NSGA-II for the chosen DES problem, as demonstrated by the fact that even DE-GEN dominates NSGA-II-SSA. The difference between DE-SSA and NSGA-II-SSA is significant. In terms of time, the fastest DE solution is twice as fast as the fastest NSGA-II solution (24 vs. 50 seconds), and in terms of cost DE is also almost twice as good (452 vs. 829).

![Figure 6.9: Non-dominated solutions from parameter tuned NSGA-II and DE.](image)

(a) Cost versus time trade-off  
(b) Evaluations versus time trade-off

A subset of solutions from Figure 6.9 is selected to be replicated on Digital Ocean’s cloud platform. The non-dominated solutions from those experiments are shown in Figure 6.10. Comparing Figure 6.9a and Figure 6.10a shows that the relative ordering of the algorithms is preserved in the cloud experiments. The steady-state asynchronous algorithms are better than the generational algorithms and DE is validated to be better than NSGA-II. Figure 6.10a also shows that the generational algorithms use fewer evaluations. However, this advantage is lost by their low utilization of the computing resources.

The parallel coordinate plot of all non-dominated solutions from the parameter tuning experiments for DE is shown in Figure 6.11. The most significant difference between DE-SSA and DE-GEN is that the optimal population size for DE-GEN is around 100, while the optimal population size for DE-SSA is below 300. The other parameter values are similar. There are some notable differences in the tuned parameter values when compared with other commonly recommended values, such as $\lambda = 100$, $p_c = 0.3$, and $F = 0.5$. The
polynomial mutation probability is low and $\eta_m$ is high, indicating that the polynomial mutation is not that important. The differential weight is around 1 and the crossover probability is around 0.9, which is high compared with the commonly recommended values.

There is also no linear relationship between the offspring population size and the other parameters, for any of the algorithms, indicating that there is no simple heuristic that can be used to adjust the other control parameters based on the value of the offspring population size.

Figure 6.10: Non-dominated solutions from cloud experiments.
Figure 6.11: Parallel coordinate plot of the objectives and parameter settings of all non-dominated CPV tuples for DE.
CONCLUSIONS AND FUTURE RESEARCH
CHAPTER 7
CONCLUSIONS AND FUTURE RESEARCH

The improvement of optimization algorithms can enable more complex optimization problems to be tackled, existing optimization problems to be solved faster, and optimization runs to be more reliable. The approach taken in this thesis is to improve optimization algorithms by understanding how control parameters affect the performance of an optimization algorithm. This is achieved by formulating the problem of setting the control parameters as an optimization problem itself.

The thesis uses a bilevel optimization approach to solve the parameter tuning problem. This approach is computationally expensive, since a complete optimization run has to be performed to evaluate a single set of control parameters. A framework and software architecture was therefore developed that could utilize parallel computing resources and efficiently scale to hundreds of computing nodes with thousands of CPU cores. The components of the software architecture can easily be distributed as a single executable file or embedded into another application, as well as deployed to hundreds of computers without other hardware or software requirements. The framework supports a variety of parameter tuning aspects, such as tuning both specialist and generalist CPV tuples, handling both single- and multi-objective parameter tuning problems, and supporting both static and dynamic upper-level resampling strategies. The framework and software architecture was tested by performing millions of individual optimizations spanning many thousands of parameter tuning experiments.

The parameter tuning problem is difficult because of nonlinear objective functions, interacting variables, multiple local optima, and the stochasticity of the optimizers. However, it is on precisely such difficult optimization problems that evolutionary algorithms have been shown to be effective. Previous work has demonstrated that evolutionary algorithms can be successfully used to solve the parameter tuning problem. However, the computational complexity of the parameter tuning problem has often limited existing studies to a few control parameters at a time or to a limited set of problems. In this work, two evolutionary algorithms, differential evolution and a genetic algorithm, are evaluated as parameter tuners using the above-mentioned framework and software architecture. The importance of replications was investigated, and it was found that different parameter tuning problems can require very different resampling strategies. However, a simple dynamic resampling strategy of using few replications at the beginning of the search and later increasing their number was shown to provide a good trade-off between the quantity of CPV tuples explored and the quality of CPV tuples obtained. When the GA was configured with commonly recommended control parameter values, it had a tendency to converge to local optima. This problem could be overcome by having explicit exploration and exploitation configurations for the control parameters.

The performance of both DE and GA as upper-level tuners was compared with that of irace, which is a state-of-the-art parameter tuning method. Their performance was evaluated by tuning both specialist and generalist CPV tuples. The parameter tuning problems were chosen to provide different types of difficulties that can be encountered when solving parameter tuning problems, specifically, the need to balance the trade-off
between the number of replications and testing new and unique CPV tuples. Both DE and GA were significantly better than irace on the parameter tuning problems tested. Overall, DE was found to be a better tuner than the GA, considering all configurations tested. This shows that evolutionary algorithms can be competitive with other parameter tuning methods.

The proposed framework and software architecture using evolutionary algorithms as tuners have been evaluated and tested by tuning the control parameters of a variety of optimization algorithms. Lower-level optimization algorithms have been tuned for both single- and multi-objective problems using both academic test problems and real-world discrete event simulation problems. Both single and multi-objective parameter tuning problems have been solved by the upper-level optimization. Optimization algorithms have been tuned as both specialists and generalists and it was found that there can be significant performance differences between them. Parameter tuning was also used in a novel way to estimate an adaptive parameter control mechanism. The results indicate that much of the benefit of an adaptive parameter control mechanism can be gained by simply switching between two or three different parameter sets. A two-objective parameter tuning problem was used to find the trade-off between increasing the parallelization potential and function evaluation efficiency in evolutionary algorithms. DE was shown to provide a better trade-off regarding these two objectives than did NSGA-II on a discrete-event simulation problem. This work was later extended to include steady-state asynchronous versions of both DE and NSGA-II. The different versions of DE and NSGA-II were first tuned against a simpler version of the DES model and the best configurations were then replicated on a cloud-computing platform to validate the results. These results indicate that the steady-state asynchronous versions were better than the generational versions and that DE was better than NSGA-II for that particular DES model.

7.1 CONTRIBUTIONS TO KNOWLEDGE

The main contributions of this thesis are: (i) a framework and software architecture that together constitute a bilevel optimization approach for solving the parameter tuning problem; (ii) an analysis of the viability of evolutionary algorithms as tuners; and (iii) consideration of how parameter tuning can be used to improve optimization algorithm performance. The following is a summary of the contributions to knowledge of this thesis and the research questions that they relate to:

- **RQ1.** A framework and software architecture for large-scale parameter tuning experiments was developed. The viability of the design was tested by running thousands of parameter tuning experiments and millions of optimizations.

- **RQ2.** DE performs well with standard parameters as a tuner algorithm. A GA performs better as a tuner when it is configured for an explicit exploration and exploitation strategy. Dynamic resampling strategies are better than static resampling strategies for both DE and GA. However, this is highly problem dependent.

- **RQ2.** Both DE and GA significantly outperform irace for the parameter tuning problems examined in this thesis. This indicates that with an appropriate configuration, evolutionary algorithms can perform better than other parameter tuning methods.

- **RQ3.** The mutation probability rule of thumb \( p_m = \frac{1}{N} \) for evolutionary algorithms was evaluated using parameter tuning experiments. The results indicate
that it is able to provide good estimates of the mutation probability for NSGA-II on a DES optimization problem and the ZDT test problems.

- RQ3. The experiments in this thesis provided more evidence that the performance difference between specialist and generalist parameter settings can be substantial. It is therefore always preferable to use specialist settings when solving a complex real-world optimization problem.

- RQ3. Parameter tuning experiments are not made obsolete by parameter control mechanisms. In fact, parameter tuning can be used to design such mechanisms. This was demonstrated by tuning multiple static parameter sets of NSGA-II in order to approximate an optimal parameter control mechanism. These results indicate that much of the performance benefit can be had with only two or three static parameter sets that switch during optimization.

- RQ3. The performance of DE was superior to that of NSGA-II for a DES problem when both algorithms were configured with large offspring population sizes. This indicates that DE is a better choice than NSGA-II when large parallel resources are available.

- RQ3. For parallel computing environments where utilization of the computing resources is a factor, it was shown that steady-state asynchronous versions of DE and NSGA-II outperform generational versions of the same algorithms on a DES optimization problem.

### 7.2 FUTURE RESEARCH

This section describes some areas that have been identified as interesting topics for further research.

#### 7.2.1 SOFTWARE ARCHITECTURE AND IMPLEMENTATION

The framework and software architecture presented in this thesis were deployed and tested on up to a hundred computers without any noticeable performance issues. However, inefficiencies in the design or implementation could manifest themselves if the framework and software architecture are ever deployed on a much larger scale. As the cost of cloud computing infrastructure decreases, it becomes even more feasible to perform even larger parameter tuning experiments than those carried out in here. Deploying the proposed system to thousands of computers would be a good stress test to explore its limitations. The work in this thesis regarding the design and implementation of the system did not include any empirical experiments on the merits of the different design decisions. Performing such experiments would help identify weak and strong points in the system and provide empirical data on how it could be improved further. It would also provide justification for decisions that complicate the design, but significantly improve the performance of the system.

Even though the bilevel optimization framework developed in this thesis was used to solve parameter tuning problems, it could with minor modifications also be used to solve other bilevel problems, since all bilevel problems are, essentially, two nested optimizations. Utilizing the bilevel optimization framework for this purpose could enable otherwise prohibitively computationally expensive bilevel problems to be solved.
7.2.2 EVOLUTIONARY ALGORITHMS AS PARAMETER TUNERS

Part of this thesis investigated the performance of evolutionary algorithms as they solved the parameter tuning problem using a bilevel optimization approach. Both the resampling strategy and the search strategy as they relate to solving the parameter tuning problem were studied. The resampling strategy plays an important role in both the efficiency and performance of an optimization algorithm, especially for parameter tuning problems since they are computationally demanding. There is also no single best strategy since different parameter tuning problems can have very different optimal resampling strategies, as is indicated by the results in this thesis. There are, however, other interesting resampling strategies in the literature whose performance in the parameter tuning domain would be worth studying, specifically, those that use statistical tests to determine where to allocate the replication budget and methods that detect and discard poorly performing CPV tuples early on in the search process.

Another aspect studied in this thesis was the performance of different search strategies controlled by the control parameters of the upper-level tuner. The parameter tuning problem is also present on the upper level, since upper-level optimization has control parameters of its own. This problem could also be formulated as an optimization problem and solved using an optimization algorithm. Such a trilevel optimization approach will, of course, be even more computationally demanding to solve. The three-level approach was not implemented in this thesis. Instead, this aspect was investigated by testing two different search strategies for the upper-level tuner. However, further research is needed to establish in more detail how the optimization algorithms are best configured for parameter tuning problems. It is also not clear to what extent parameter tuning problems are similar to or different from the test problems found in the literature. A better understanding of their relationships can help us understand how to best use and configure them for parameter tuning problems. There is also potential for a positive feedback loop, since better understanding how to configure optimization algorithms on the lower level can also help to better configure the optimization algorithms on the upper level.

7.2.3 APPLICATIONS OF PARAMETER TUNING

One aspect of parameter tuning that warrants further research is how it can be utilized during the development of new optimization algorithms. Parameter tuning experiments during the early phases of algorithm development can help us understand how certain design decisions affect the performance of the algorithm. Such experiments can also guide what parameters to expose to the user, in addition to how they are best configured for various problems. That could help identify and eliminate control parameters that were initially thought to be important but are actually of little importance to the performance of the algorithm.

Parameter tuning experiments can also be used to develop and tune resampling strategies in the same way that the optimal adaption rate of a parameter control mechanism was estimated by tuning multiple parameter sets in Paper III. The trade-off between number of replications and testing new CPV tuples can be explored by tuning the number of replications to use in each stage of the optimization. For resampling strategies that use statistical testing, parameters such as the required significance and even which type of test to use could be tuned.

Parameter tuning can also be used to categorize optimization problems based on the type of control parameter values that best solve them. Each category would then contain only problems best solved by a single CPV tuple. Creating such categorization might be dif-
ficult, because if the tolerances in what are considered equal parameter values are too small, there would be very few problems in each category, whereas tolerances that are too large would put all the problems in one category. Nevertheless, such a categorization might shed some light on both the nature of optimization problems and the control parameters of an optimization algorithm.
INCLUDED PAPERS
Parameter Tuned CMA-ES on the CEC’15 Expensive Problems

Martin Andersson, Sunith Bandaru, Amos H.C. Ng, Anna Syberfeldt
School of Engineering Science,
University of Skövde, Skövde, Sweden
{martin.andersson,sunith.bandaru,amos.ng,anna.syberfeldt}@his.se

Abstract—Evolutionary optimization algorithms have parameters that are used to adapt the search strategy to suit different optimization problems. Selecting the optimal parameter values for a given problem is difficult without a-priori knowledge. Experimental studies can provide this knowledge by finding the best parameter values for a specific set of problems. This knowledge can also be constructed into heuristics (rule-of-thumbs) that can adapt the parameters for the problem. The aim of this paper is to assess the heuristics of the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) optimization algorithm. This is accomplished by tuning CMA-ES parameters so as to maximize its performance on the CEC’15 problems, using a bilevel optimization approach that searches for the optimal parameter values. The optimized parameter values are compared against the parameter values suggested by the heuristics. The difference between specialized and generalized parameter values are also investigated.

I. INTRODUCTION

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [1], [2] is a single objective evolutionary optimization algorithm that for certain problems outperforms other evolutionary algorithms, like genetic algorithms, differential evolution and particle swarm optimization [3], [4], [5]. CMA-ES generates new search points by sampling them from a multivariate normal distribution. The distribution is determined by its mean \( \mu \in \mathbb{R} \), standard deviation \( \sigma \in \mathbb{R} \) and its covariance matrix \( \mathbf{C} \in \mathbb{R}^{n \times n} \). By modifying the covariance matrix the search distribution is made to fit the contour lines of the objective function, thereby increasing the probability of generating good solutions.

The no free lunch theorem [6] states that no optimization algorithm can be better than all other algorithms on all problems. Optimization algorithms try to circumvent this fact by using parameters that can be tweaked to alter its search behaviour. Experimental studies can be used to find the appropriate parameter values for a given problem and optimization algorithm. However, this is both difficult and time-consuming, since many experiments are required to obtain reliable results. Another approach is to use existing heuristics or rules of thumb to estimate good parameters for new problems. This is less computationally expensive, since it does not require any experiments. There is, however, no guarantee that the selected parameters will actually work well for that problem.

CMA-ES has heuristics for all of its parameters. They estimate parameter values mostly based on the number of dimensions of the problem. The heuristics are derived from both empirical studies and inherent properties of CMA-ES. They are also designed to be effective over a diverse set of problems. The aim of this paper is to investigate how good these heuristics are when using CMA-ES parameters to maximize its performance on the CEC’15 expensive problems [7]. Table I provides a summary of the included functions. A bilevel optimization approach will be used to search for the parameters that maximize the performance of CMA-ES on the CEC’15 problems. This will allow for a comparison of the difference in performances between the optimized and default (heuristic suggested) parameter values.

Another important aspect of parameter tuning regards generalized and specialized parameter values. Generalized parameter values are those that are meant to work across many different problems, while specialized parameter values are fine-tuned against a small set of problems. This is important because the optimal parameter values of an optimization algorithm can be quite different between problems and also between specialized and generalized parameter values [8].

### TABLE I. SUMMARY OF CEC’15 EXPENSIVE TEST PROBLEMS

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Description</th>
<th>Parameter Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite Functions</td>
<td>Composite Function 1 (N=3)</td>
<td>Elliptic Function</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>Composite Function 2 (N=5)</td>
<td>Expanded Elliptic Function</td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>Composite Function 3 (N=5)</td>
<td>Rotated Bent Cigar Function</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>Composite Function 4 (N=5)</td>
<td>Shifted and Rotated Expanded Griewank's Function</td>
<td>2500</td>
</tr>
<tr>
<td></td>
<td>Composite Function 5 (N=5)</td>
<td>Shifted and Rotated HappyCat Function</td>
<td>3000</td>
</tr>
<tr>
<td></td>
<td>Composite Function 6 (N=5)</td>
<td>Shifted and Rotated Katsuura Function</td>
<td>3500</td>
</tr>
<tr>
<td></td>
<td>Composite Function 7 (N=5)</td>
<td>Shifted and Rotated Schwefel's Function</td>
<td>4000</td>
</tr>
<tr>
<td></td>
<td>Composite Function 8 (N=5)</td>
<td>Shifted and Rotated Rastrigin's Function</td>
<td>4500</td>
</tr>
<tr>
<td>Composite Functions</td>
<td>Composite Function 9 (N=5)</td>
<td>Shifted and Rotated Griewank's Function</td>
<td>5000</td>
</tr>
<tr>
<td>Composite Functions</td>
<td>Composite Function 10 (N=5)</td>
<td>Shifted and Rotated Rosenbrock's Function</td>
<td>5500</td>
</tr>
<tr>
<td></td>
<td>Composite Function 11 (N=5)</td>
<td>Shifted and Rotated Schwefel's Function</td>
<td>6000</td>
</tr>
<tr>
<td></td>
<td>Composite Function 12 (N=5)</td>
<td>Shifted and Rotated Rastrigin's Function</td>
<td>6500</td>
</tr>
<tr>
<td></td>
<td>Composite Function 13 (N=5)</td>
<td>Shifted and Rotated Griewank's Function</td>
<td>7000</td>
</tr>
<tr>
<td></td>
<td>Composite Function 14 (N=5)</td>
<td>Shifted and Rotated Rosenbrock's Function</td>
<td>7500</td>
</tr>
<tr>
<td></td>
<td>Composite Function 15 (N=5)</td>
<td>Shifted and Rotated Schwefel's Function</td>
<td>8000</td>
</tr>
<tr>
<td></td>
<td>Composite Function 16 (N=5)</td>
<td>Shifted and Rotated Rastrigin's Function</td>
<td>8500</td>
</tr>
</tbody>
</table>

This table provides a summary of the included functions. The bilevel optimization approach will be used to search for the parameters that maximize the performance of CMA-ES on the CEC’15 problems.
values are often more useful for the practitioner because they are designed to be applicable to a wide range of problems.

This paper analyzes both specialized and generalized CMA-ES parameter values, both in terms of performance and the differences of the optimal parameter values. To find specialized parameter values each problem and dimension is optimized individually, while generalized parameter values are obtained by searching for the optimal parameter values across all problems and dimensions.

It is possible to distinguish three layers in parameter tuning: The application layer, the algorithm (lower) layer and the design (upper) layer [9]. The problem to be solved is located on the application layer and the metaheuristic to solve that problem is on the algorithm layer. On the design layer is the parameter tuner that tests different parameters for the metaheuristic on the algorithm layer. To avoid confusion, the quality of solutions for the problem on the application layer is called fitness while the quality of the parameters in the design layer is called utility [9].

Parameter tuning can itself be viewed as an optimization problem in which the objective is to find the parameter values that give the best performance on a particular problem or a set of problems. This approach can be referred to as meta-ES [9] or bilevel optimization [10]. In this paper, the objective at the upper level optimization is the same as the CEC'15 expensive problems scoring method. The scoring method is the summation of the mean and median of the best function values over multiple runs of the 15 test problems, for both 10 and 30 dimensions, as shown below:

\[
\text{Minimize } \sum_{i}^{15} \text{mean}(f_i(p)) + \bar{\text{median}}(f_i(p)) \quad \text{subject to } x^l \leq x \leq x^u
\]

(1)

The algorithmic parameters of the lower-level optimization problem become the variables for the upper-level optimization problem. The objective for each test problem is calculated according to the following equation, where MaxFES is the maximum number of function evaluations allowed for each problem.

\[
f(x) = 0.5 \times (f_{\text{MaxFES}} + f_{0.5 \times \text{MaxFES}})
\]

(2)

The CEC'15 problems are called expensive problems because they only allow for small number of function evaluations, 500 and 1500 evaluations for 10 and 30 dimensions respectively. Most studies use far more function evaluations (in [3] the maximum was set at \(10^{17}\)) when comparing optimization algorithm performances. The size of the function evaluation budget will most probably affect the optimal parameters. This aspect is however not addressed in this paper.

The rest of the paper is organized as follows. Section II introduces CMA-ES and its parameters. In Section III a description of the experimental design is provided. The experimental results appear in Section IV. The conclusions are summarized in Section V.

II. CMA-ES

There are several variants of CMA-ES. The one used in this paper is the \((\mu/\mu, \lambda)\)-CMA-ES. Here \(\lambda\) is the population size, \(\mu\) is the number of selected search points and \(\mu, \lambda\) indicates that the new search points are weighted when updating the mean.

New search points are generated from a multi-variate normal distribution. They are evaluated and ranked according to their fitness. The best \(\mu\) of all \(\lambda\) points are weighted and summed to form the new mean. Instead of only using the selection information from a single generation, CMA-ES utilizes the path taken by the population over a number of generations. This is called the evolution path. The covariance matrix is updated by the evolution path and the \(\mu\) weighted difference vectors of previous and new search points. The step size \(\sigma\) is also updated using an evolution path. Reliably estimating the covariance matrix from a single generation is not always possible. That is why information from previous generations are also added, this is called the rank-\(\mu\)-update. For a complete description of CMA-ES, see [1]. The CMA-ES implementation used in this paper is based on Hansen's C code [11].

A. Parameters and Heuristics

CMA-ES has strategy parameters that can be used to control its search behavior. This section will provide a short description of them and present the heuristics that are used to set them. The heuristics are also based on those found in Hansen's C code [11]. In the following equations, \(N\) refers to the dimension of the problem.

1) \(\lambda\): The number of new search points generated in each generation.

\[
\lambda = 4 + \lfloor 3 \times \ln(N) \rfloor
\]

(3)

2) \(\mu\): The number of search points that will form the new mean. Better points are given more significance using weights given by Equation (5).

\[
\mu = \left[ \frac{\lambda}{2} \right]
\]

(4)

\[
w_i = \ln (\mu + 1) - \ln (i); \quad w_i = \frac{w_i}{\sum_{i=1}^{\mu} w_i}
\]

(5)
3) $c_σ$: The length of the evolution path horizon. This controls the learning rate for the cumulation of the step size.

$$c_σ = \frac{\mu_{eff} + 2}{N + \mu_{eff} + 3}$$  \hspace{1cm} (6)

$$\mu_{eff} = \left(\sum_{i=1}^{N} w_i^2\right)^{-1}$$  \hspace{1cm} (7)

4) $c_c$: The length of the evolution path horizon. This controls the learning rate for the cumulation of the rank-one update of the covariance matrix.

$$c_c = \frac{4}{N + 4}$$  \hspace{1cm} (8)

5) $c_{cav}$: The learning rate for the covariance matrix.

$$c_{cav} = \frac{1}{\mu_{cav}} \times t_1 + \left(1 - \frac{1}{\mu_{cav}}\right) \times t_2$$  \hspace{1cm} (9)

$$\mu_{cav} = \mu_{eff}$$  \hspace{1cm} (10)

$$t_1 = \frac{2}{(N + \sqrt{2})^2}$$  \hspace{1cm} (11)

$$t_2 = \min\left(\frac{2 + \mu_{eff} - 1}{(N + 2) + (N + 2) + \mu_{eff}}\right)$$  \hspace{1cm} (12)

6) $σ^{(0)}$: The initial step size. The step size is problem dependent, but the optimum of the optimized function should fall within $w^{(0)} \pm 2σ^{(0)}$. With decision variables scaled between $[0, 10]$ and random $w^{(0)}$, the initial step size is chosen to be $2$.

$$σ^{(0)} = 2$$  \hspace{1cm} (13)

7) $d_{σ}$: Dampening for the step size update.

$$d_{σ} = t_1 \times \max\left(0, 3.1 - \frac{N}{2t_{m} + N \frac{w_{σ}}{λ \cdot w_{σ}}}\right) + c_σ$$  \hspace{1cm} (14)

$$t_1 = 1 + 2 \times \max\left(0, \left(\frac{\mu_{eff} - 1}{N + 1}\right) - 1\right)$$  \hspace{1cm} (15)

III. EXPERIMENTAL DESIGN

Each experiment in this study has a set $t$ that either include one or all test problems. Experiments with only one test problem find specialized parameter values and experiments with all problems find generalized parameter values. In total, there are 31 experiments, 15 test problems $\times$ 2 dimensions for the specialized experiments and 1 experiment for the generalized experiment.

The bilevel optimization approach to parameter tuning only provides a single optimized parameter set $p^*$ for each experiment. To get a larger sample size to draw conclusions from, each experiment is independently replicated 20 times. The outcome of each replication is the set of parameter values with the best objective value, as measured by Equation (2). Therefore, each experiment produces 20 sets of optimized parameter values. Within each experiment, every evaluated parameter set $p$ is independently replicated 20 times. The average fitness from these optimizations is then used as the utility of that parameter set.

The experiments were run on three Dell PowerEdge R420 servers, each with two Intel Xeon ES-2400 V2 processors for a total of 72 logical cores. The optimization runs were distributed across the servers using an optimization framework written in C++. The framework has the capability of distributing and running independent optimizations in parallel, allowing for an efficient use of all the available computing resources. The experiments took 85 hours to complete.

A. Design Layer

CMA-ES is used to solve the single objective minimization problem, Equation (1), at the design layer. There are two main reasons for choosing CMA-ES. It is known to be an effective single objective optimization algorithm and it has default values for most parameters [3], [4]. $λ$ is set to 10 which is slightly larger than the default. All parameters at the design layer are scaled so they fall in the range $[0, 10]$ and because of that $σ^{(0)}$ is set to 2. All other parameters use their default values as described previously.

The maximum iterations allowed at the design layer is 6000, with restarts at iteration 2000 and 4000. Restarts are used to reduce the probability of CMA-ES getting trapped in a local optimum. Initial experiments showed that improvements had plateaued by 2000 iterations, which is why each restart was separated by that number of iterations.

B. Algorithm Layer

The algorithm (lower) layer is where the utility of a parameter set $p$ is evaluated. The evaluation is performed by starting instances of CMA-ES against a set $t$ containing one or more of the CEC 15 problems. The composition of the set $t$ is different for each experiment. For every problem in the set $t$ 20 instances of CMA-ES are started with the parameter set that is being evaluated. The utility of the parameter set $p$ is then calculated as the sum of all mean and median fitnesses obtained by these optimizations, as shown in the objective function in Equation (1).

There are 15 functions included in the CEC'15 expensive problems. Each function is optimized for 10 and 30
dimensions, which are allowed a maximum of 500 and 1500 function evaluations. The decision variables are scaled so that they fall in the range \([-100, 100]\). The fitness for a test problem is calculated according to Equation (2).

C. Starting Positions

Each experiment replication is assigned a set of 20 (one for each lower level replication) random starting positions for the initial mean. That means that CMA-ES instances within a replication use the same set of starting positions for evaluating any given parameter set \( p \), and that the starting positions are different between replications. Every parameter set \( p \) is evaluated against all starting positions in the assigned set.

By evaluating each parameter set \( p \) against different starting positions, the probability of specifically optimizing the parameters for a particular starting position is reduced. Using the same set of starting positions for all evaluations within a replication reduces the effect of the starting position from the performance of a particular parameter set. An alternative to statically assigning starting positions would be to randomly generate them at the start of each optimization. That approach avoids the issue of sub-optimizing for a particular starting position, but it also makes the comparison between parameters sets unfair since they do not have the same starting conditions.

D. Parameters

The following CMA-ES parameters are tuned in this study.

1) \( \lambda \): The population size: an integer in the range \([2, 300]\).
2) \( \mu \): The number of selected search points as a percentage of the population size: a real-value in the range \([0, 1]\).
3) \( c_{c} \): Learning rate for the cumulation of the size: a real-value in the range \([0, 1]\).
4) \( c_{c} \): Learning rate for the cumulation of the rank-one update: a real-value in the range \([0, 1]\).
5) \( c_{c} \): Learning rate for the covariance matrix update: a real-value in the range \([0, 1]\).
6) \( \sigma^{(0)} \): The initial step size: a real-value in the range \([0, 10]\).
7) \( \delta \): Damping parameter for step size update: a real-value in the range \([0, 10]\).

The parameters of the optimization on the algorithm layer in Equation (1) become variables for the optimization on the design layer. Thus the variable vector \( p \) in Equation (1) is \( p = \{\lambda, \mu, c_{c}, c_{c}, c_{c}, \sigma^{(0)}, \delta\} \).

IV. EXPERIMENTAL RESULTS

The results are divided into two sections, the first section consists of the results of the parameter tuning and it is followed by a comparison of the optimized versus the default parameter values. The parameter tuning experiments use Equation (2) to calculate the fitness, while the replicated generalized and default parameter values use Equation (16).

\[
f(a) = f_{Opt} PE
\]  
(16)

The results of four different variants of CMA-ES are presented in this section. CMAES-S and CMAES-G denote the specialized and generalized parameters experiments respectively. CMAES-R is the best parameter set \( p^* \) from the generalized parameter experiment replicated on a new set of starting positions. The same set of starting position is used to get the results for the default parameters, CMAES-D.

The computational complexity for the default and optimized parameter values are shown in Table II, calculated according to the guidelines given in [7].

A. Parameter Tuning Results

Since the same formula is used for the design layer problem as the scoring method in the CEC'15 expensive problems, the results for the parameter tuning experiments are representative of the final score.

Table III shows the best, median and mean utility from each experiment. The scores, as calculated by the objective function in Equation (1), are shown in Table IV. For CMAES-S the values are the summation of all individual problems. Thus, the difference of CMAES-S and CMAES-G is the performance gain (or loss) of allowing each test problem to use specialized instead of generalized parameters.

<table>
<thead>
<tr>
<th>Table II</th>
<th>Computational Complexity for Default (T1) and Optimized (T2) Parameter Values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>T0</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
</tr>
<tr>
<td>T1/T0</td>
<td></td>
</tr>
<tr>
<td>T2/T0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>100.6</td>
<td></td>
</tr>
<tr>
<td>200.0</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>35.8</td>
<td></td>
</tr>
<tr>
<td>0.09</td>
<td></td>
</tr>
<tr>
<td>113.3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table III</th>
<th>Best, Median and Mean Results from 20 Parameter Tuning Experiments for Specialized Parameters (CMAES-S) and Generalized Parameters (CMAES-G).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Best</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>Parameter1</td>
<td></td>
</tr>
<tr>
<td>Parameter2</td>
<td></td>
</tr>
<tr>
<td>Parameter3</td>
<td></td>
</tr>
<tr>
<td>Parameter4</td>
<td></td>
</tr>
<tr>
<td>Parameter5</td>
<td></td>
</tr>
<tr>
<td>Parameter6</td>
<td></td>
</tr>
<tr>
<td>Parameter7</td>
<td></td>
</tr>
<tr>
<td>Parameter8</td>
<td></td>
</tr>
<tr>
<td>Parameter9</td>
<td></td>
</tr>
<tr>
<td>Parameter10</td>
<td></td>
</tr>
<tr>
<td>Parameter11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table IV</th>
<th>The Score as Calculated by the Objective Function in Equation (1). The Specialized Parameters Score (CMAES-S) is the Summation of All Individual Results.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Score</td>
</tr>
<tr>
<td>Parameter1</td>
<td></td>
</tr>
<tr>
<td>Parameter2</td>
<td></td>
</tr>
<tr>
<td>Parameter3</td>
<td></td>
</tr>
<tr>
<td>Parameter4</td>
<td></td>
</tr>
<tr>
<td>Parameter5</td>
<td></td>
</tr>
<tr>
<td>Parameter6</td>
<td></td>
</tr>
<tr>
<td>Parameter7</td>
<td></td>
</tr>
<tr>
<td>Parameter8</td>
<td></td>
</tr>
<tr>
<td>Parameter9</td>
<td></td>
</tr>
<tr>
<td>Parameter10</td>
<td></td>
</tr>
<tr>
<td>Parameter11</td>
<td></td>
</tr>
</tbody>
</table>
The specialized parameters are able to improve the performance by 81% for the median and 128% for the best result. This shows that the no free lunch theorem holds for CMA-ES and the CEC’15 problems, because no set of parameters could be found that are optimal across all problems. From the results it is also clear that functions 1, 2, 10 are the most difficult ones. It is reasonable to assume that these functions will influence the general parameters the most, because of the formulation of the scoring method.

The optimized parameter values from all 20 replications, for both 10 and 30 dimensions, are shown as boxplots in Figures 1-7. The function labeled G in the plots uses the generalized parameters.

1) Figure 1: The median values for \( \sigma^{(1)} \) vary between problems. The variation pattern is roughly the same for both 10 and 30 dimensions. However, 30 dimensions have lower values. For 10 dimensions all median values are below 4, while for 30 dimensions the median values are below 2.

2) Figure 2: The median values for \( \mu \) do not significantly change between problems. The exceptions are functions 9 and 15, which have lower values than the rest. Another observation is that 30 dimensions have in general higher values than 10 dimensions.

3) Figures 3, 4 and 5: No clear patterns can be observed with \( b_{c_1}, c_2 \) and \( c_6 \). Although the significance of \( b_{c_1} \) seems to be higher for 30 dimensions, especially for function 1. More experiments are needed to determine if they do not influence the performance in a significant way or if they have dependencies on other parameters which allow them to have a wide range of optimal values.

4) Figure 6: The median values for \( c_{\text{cov}} \) vary between problems. One trend that can be observed is that the values for 30 dimensions are smaller than for 10. Smaller values are clearly preferred for the generalized parameters.

5) Figure 7: In general, optimized \( \lambda \) values are around 10, with exceptions for functions 5, 9 and 15 with 10 dimensions and 2, 5 and 9 with 30 dimensions. Apart from those exceptions, there are no discernible differences between 10 and 30 dimensions.

**B. Tuned Parameters vs Default Parameters**

Table V shows the parameters with the best utility from each experiment. The row labeled Generalized are the best generalized parameters and Default are the parameters as suggested by the heuristics. The optimized \( \lambda \) and \( \mu \) parameters are smaller than the default. This will lead to faster convergence at the cost of reduced global search capability. It is difficult to draw any conclusions about the other optimized parameters as they are similar to the default values.

Tables VII and IX shows the performance of the generalized parameters on the CEC’15 problems, for 10 and 30 dimensions. For comparison, the performance using default values are shown in Table VI and VIII. The fitness values for those experiments are calculated using Equation (16). Note that this is different from the parameter tuning experiments which used Equation (2).
TABLE IX. CMAES-R, RESULTS FOR 30D

<table>
<thead>
<tr>
<th>Func</th>
<th>Best</th>
<th>Worst</th>
<th>Median</th>
<th>Mean</th>
<th>Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.691E+03</td>
<td>1.476E+07</td>
<td>2.264E+05</td>
<td>1.289E+06</td>
<td>...</td>
</tr>
</tbody>
</table>


The optimized parameter values are able to improve on the default parameter values in most of the CEC 15 problems, see Table X. Functions 1, 2 and 10 are the ones that did improve the most. For the total score over all problems, the optimized parameters increased the performance by a factor of almost 32.

V. CONCLUSIONS

This paper tuned the parameters of CMA-ES with the aim of maximizing its performance on the CEC 15 expensive problems. A bilevel optimization approach was used to search for both generalized and specialized parameters. The results show that generalized parameters have lower performance than specialized parameters, and that the general parameters can have values that are different from any of the specialized parameters.

The optimized parameter values were also compared against the values suggested by the heuristics, both in terms of performance on CEC 15 problems and how similar they were. Two notable differences have been seen in the λ and µ parameters. Small values for the population size parameter λ leads to fast convergence and large values help in avoiding local optima. Compared to the default parameters, the parameter tuning results show that a smaller λ provide a better trade-off for the CEC 15 problems. Larger values of µ increase the explorative behavior. The default heuristic for µ, Equation (4), from [11] suggested a value of 0.5. The results show that the optimized value of 0.26 is closer to the proposed value of 0.27 in [1]. Even though λ and µ are highlighted here, that does not mean that they are solely responsible for the performance increase. Further work needs to be done to determine the significance of each parameter and how they interact with each other.

No parameters have been found that are optimal across all problems, which is in agreement with the no free lunch theorem. The parameters found in this study are influenced by the CEC 15 scoring method and the fact that a relatively limited function evaluation budget was used. How well these parameters work on other problems with different function evaluation budgets is difficult to estimate, without doing further experiments.

The heuristics are designed to scale the parameters with the dimension of the problem. This aspect of the optimized parameters are not studied in this paper. Experiments that find optimal parameters for different dimensions are needed to address this issue.

REFERENCES

Fig. 1. Parameter tuning results for parameter $\sigma(0)$.

Fig. 2. Parameter tuning results for parameter $\mu$.

Fig. 3. Parameter tuning results for parameter $d\sigma$.

Fig. 4. Parameter tuning results for parameter $c\sigma$. 
Fig. 5. Parameter tuning results for parameter $c$.

Fig. 6. Parameter tuning results for parameter $c_{cov}$.

Fig. 7. Parameter tuning results for parameter $\lambda$. 
Paper II
Parameter Tuning of MOEAs Using a Bilevel Optimization Approach

Martin Andersson(B), Sunith Bandaru, Amos Ng, and Anna Syberfeldt

Virtual Systems Research Centre, University of Skövde, Skövde, Sweden
{martin.andersson,sunith.bandaru,amos.ng,anna.syberfeldt}@his.se

Abstract. The performance of an Evolutionary Algorithm (EA) can be greatly influenced by its parameters. The optimal parameter settings are also not necessarily the same across different problems. Finding the optimal set of parameters is therefore a difficult and often time-consuming task. This paper presents results of parameter tuning experiments on the NSGA-II and NSGA-III algorithms using the ZDT test problems. The aim is to gain new insights on the characteristics of the optimal parameter settings and to study if the parameters impose the same effect on both NSGA-II and NSGA-III. The experiments also aim at testing if the rule of thumb that the mutation probability should be set to one divided by the number of decision variables is a good heuristic on the ZDT problems. A comparison of the performance of NSGA-II and NSGA-III on the ZDT problems is also made.

Keywords: Parameter tuning · NSGA-II · NSGA-III · ZDT · Bilevel optimization · Multi-objective problems

1 Introduction

Real-world optimization problems are often formulated with multiple objectives and are therefore preferably solved using multi-objective evolutionary algorithms (MOEAs). Metaheuristics such as EAs involve a set of user-defined parameters that control various aspects of the algorithm. It is well-known [1, 10] that these settings can greatly affect the search process and the overall performance of the algorithm. However, setting them for a particular problem is not always intuitive. A strategy that is often used is to choose parameter values that have been shown to be effective on similar problems. Some metaheuristics, such as evolutionary strategies (ES), come with their own heuristics or recommendations for choosing the parameters. Neither method guarantees maximal performance from the algorithm. This paper addresses this issue by using the idea of optimal parameters, similar in principle to the one proposed in [9]. The parameter-setting problem can itself be viewed as an optimization problem in which the objective is to maximize the performance of the algorithm used on a particular problem. For single-objective problems, this performance indicator could be directly related to the best function value attained by the algorithm. Since this work considers
multi-objective problems, a commonly used performance indicator is the hypervolume [12]. Thus, our formulation contains a multi-objective problem nested within a single-objective problem and resembles the following,

\[
\begin{align*}
\text{Maximize} & \quad \mathcal{H}(p) \\
\text{where,} & \quad \mathcal{H}(p) \text{ is the hypervolume of the non-dominated set} \\
\text{obtained by solving the following problem with parameters } p \\
\text{Minimize} & \quad \{f_1(x), f_2(x), \ldots, f_M(x)\} \\
\text{Subject to} & \quad g_j(x) \geq 0 \quad \forall \ j \in \{1, 2, \ldots, J\} \\
& \quad h_k(x) = 0 \quad \forall \ k \in \{1, 2, \ldots, K\} \\
& \quad x_l \leq x \leq x_u
\end{align*}
\]

The algorithmic parameters of the lower-level optimization problem become the variables for the upper-level optimization problem.

Many real-world optimization problems are also designed to be scalable with respect to the variables. For example, consider a production line involving several machines in which their processing times have to be optimized for maximizing the overall throughput and minimizing the work in process of the line. Adding additional operations (machines) to such a line is equivalent to scaling the original optimization problem since the objectives remain the same. In such a situation, it is beneficial to study how the optimal parameter values for the algorithm change with respect to the problem size. Another important aspect is the computational cost. Objective functions in the real world are rarely analytical. In other words, evaluation of the objective functions may involve computationally expensive simulations. Studying the impact of the available computational budget on the optimal parameter values can lead to considerable savings in time and cost.

In order to illustrate the above ideas, two MOEAs, namely NSGA-II [4] and NSGA-III [3], are chosen with the ZDT test suite [13] to experimentally study the effects of problem size and available budget. NSGA-II and the ZDT test problems is a combination that is commonly used to assess the performance of a new metaheuristic. Finding the optimal parameters and the corresponding hypervolume for this combination will also allow a new metaheuristic to be compared against the NSGA-II best-case performance on the ZDT problems. Other test problems are not included in this study because that would reduce the number of experiments performed on each problem. This trade-off will allow for a more in-depth analysis of the ZDT problems.

It is worthwhile to mention here that the goal of this paper is not to find parameter settings that work across a range of problems, but to study how the optimal parameters vary for a given problem with the number of variables and budget size. In order to achieve this, several experiments will be performed on each problem to get multiple sets of optimized parameters. A secondary aim of this paper is to see how NSGA-III compares to NSGA-II in terms of performance and whether they use similar optimal parameter settings. Though NSGA-III was originally designed to handle many (\(> 3\)) objective problems, this paper will
address how it performs against NSGA-II on the ZDT problems. Testing against problems with three or more objectives would be interesting but is out of scope of this paper and left for future work.

The rest of the paper is organized as follows. Section 2 introduces the parameter setting problem and related work. In Section 3 a description of the experimental design is provided. The experimental results appear in Section 4. The conclusions are summarized in Section 5.

2 Background

The problem of finding the optimal set of parameters for a particular problem can itself be formulated as an optimization problem and solved by an EA. This bilevel optimization approach is called a meta-EA [5]. Though computationally intensive, the approach is highly parallelizable since replications of the optimizations, at both the upper and lower-level, are independent. A software framework was developed as part of this work that could efficiently distribute and run optimizations in parallel. This software together with a cluster of homogeneous commodity computers enabled the scope of the experiments to be extended well beyond what would have been feasible on a single computer.

One issue that has to be considered when testing different parameters is that they are usually not independent. This means that changing the parameters one by one may lead to sub-optimal settings. Changing them simultaneously on the other hand will require a large number of experiments to be performed. It is therefore impractical to perform parameter tuning manually, even though there exist different techniques to overcome this problem to some extent. A detailed description and taxonomy of the available techniques can be found in [5].

2.1 Classification and Terminology

It is possible to distinguish three layers in parameter tuning: The application layer, the algorithm (lower) layer and the design (upper) layer [5]. The problem to be solved is located on the application layer and the metaheuristic to solve that problem is on the algorithm layer. On the design layer is the parameter tuner that tests different parameters for the metaheuristic on the algorithm layer. To avoid confusion, the quality of solutions for the problem on the application layer is called fitness while the quality of the parameters is called utility [5]. The classification that was proposed in [6] distinguishes between parameter tuning where the parameters are static and parameter control where they can change during the optimization.

Tuners can be divided into two main categories: iterative and non-iterative [5]. Non-iterative tuners generate all parameters at the start, usually in a systematic fashion. This allows the utility landscape to be modeled from the utility of the evaluated parameters. Iterative tuners, on the other hand, generate the parameters iteratively as the tuner progresses. This makes them more suitable for finding the (near-)optimal parameter vectors, because they can perform a search of the utility landscape.
2.2 Related Work

Using a bilevel optimization approach to do parameter tuning has been done before in the literature. In [1] a Genetic Algorithm (GA) was tuned on single objective sphere problem. The authors found that the GA using the optimized parameters to be significantly better than a GA with “standard” parameters.

Another example is [7] which used a GA to tune the parameters for a GA on a set of numerical test functions. The result were then validated on a image registration task, showing a small but statistical significant advantage to the tuned GA against a “standard” GA.

A more recent example is [9] in which the authors used NSGA-II to tune the parameters of Partial Swarm Optimization (PSO) and Differential Evolution (DE). The algorithms were tuned against both the precision and speed of convergence. It was found that in addition to finding good parameters, the approach could also extract relationships between parameters and the impact of a parameter on the quality criteria.

3 Experimental Design

The meta-EA approach only provides a single optimal parameter set $p^*$ for each experiment, meaning that it does not provide much insight into the utility landscape. This paper will address this issue by running several different experiments on the same test problem. Two things will be varied for all test problems: the function evaluation budget and the number of decision variables ($N$).

3.1 Experimental Setup

The experiments involve four aspects that this paper studies, these are listed in Table 1. Each experimental setting is combination of these different aspects. Thus, in total there are 350 (2 MOEAs $\times$ 5 test problems $\times$ 7 budget sizes $\times$ 5 problem sizes) different experimental settings each of which is independently replicated 20 times. The outcome of each replication is the set of parameters with the best hypervolume. Therefore, each experimental setting produced 20 different sets of parameters.

Each experimental setting was a bilevel optimization with a function evaluation budget of 1000, using the parameters shown in the third column of Table 2. The budget was based on manually analyzing a small number of bilevel optimizations and identifying the fact that most of them stopped improving after about 500 evaluations. The MOEA being optimized, at the algorithm layer, was also independently replicated 20 times for each set of parameters being evaluated. The average hypervolume from these optimizations was then used as the utility of that set of parameters.

3.2 Experimental Settings

The following paragraphs explain each row of Table 1.
Table 1. Experimental settings and corresponding choices

<table>
<thead>
<tr>
<th>Experimental setting</th>
<th>Experimental choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOEA</td>
<td>NSGA-II, NSGA-III</td>
</tr>
<tr>
<td>Test problem</td>
<td>ZDT1, ZDT2, ZDT3, ZDT4, ZDT6</td>
</tr>
<tr>
<td>Function evaluations</td>
<td>100, 500, 2000, 3500, 5000, 6500, 8000</td>
</tr>
<tr>
<td>Number of decision variables</td>
<td>2, 10, 20, 30, 40</td>
</tr>
</tbody>
</table>

MOEAs and Test Problems. The two tuned MOEAs, NSGA-II and NSGA-III, on the algorithm layer are both real-value coded. That is why the binary coded test problem ZDT5 was excluded from the experiments. All other ZDT test problems are used in this study. The reference point for the hypervolume calculations for ZDT\{1, 2, 3, 6\} is (11, 11) and (11, 1000) for ZDT4. The reason for the higher reference point on ZDT4 is that some of the optimizations failed to reach any solution within the (11, 11) reference point.

Both NSGA-II and NSGA-III use the SBX crossover operator and a polynomial mutation.

Function Evaluations. It has been argued that keeping the parameters static during an optimization is not optimal [6]. This would also indicate that it is advantageous to use different parameters for different function evaluation budgets, even though the parameters are static during the run. In order to test this, each experiment will be performed with different budget sizes.

Number of Decision Variables. A number of different rules of thumb have been proposed in the literature. For example, in a binary coded GA, the mutation rate should be proportional to the length of the chromosome [8], \( p_m = 1/l \). For a real-value coded GA the length is substituted with the number of decision variables. Previous work has found this rule to be accurate on a single objective sphere problem [1]. This rule will be tested by varying the number of decision variables for each problem.

3.3 Meta-EA Parameters

At the design layer is a real-value coded meta-EA using the SBX crossover and a polynomial mutation. This introduces the problem of choosing a good set of parameters at the design layer as well. To avoid using yet another meta-EA to solve this problem, a full factorial experimental design was performed instead. The values for each parameter is shown in Table 2. To limit the runtime of these experiments only one test problem, ZDT1, was selected as the test problem on the application layer. To further limit the scope only the NSGA-II algorithm was used at the algorithm layer. The function evaluation budget for NSGA-II
on the algorithm layer was 1000 and the number of replications were 10. On
the design layer the function evaluation budget for the meta-EA was 250 with
10 replications. The parameters with the highest average hypervolume was then
chosen as the set of parameters to use at the design layer for the rest of the
experiments. The chosen parameters are shown in the third column in Table 2.

Table 2. Full factorial experimental design for meta-EA parameter settings

<table>
<thead>
<tr>
<th>Meta-EA Parameter</th>
<th>Possible Values</th>
<th>Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>4, 8, 16</td>
<td>8</td>
</tr>
<tr>
<td>Mutation Probability</td>
<td>0.2, 0.4, 0.6, 0.8</td>
<td>0.4</td>
</tr>
<tr>
<td>Mutation Distribution Index</td>
<td>1, 2, 5, 10, 20, 40</td>
<td>1</td>
</tr>
<tr>
<td>Crossover Probability</td>
<td>0.2, 0.4, 0.6, 0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>Crossover Distribution Index</td>
<td>1, 2, 5, 10, 20, 40</td>
<td>40</td>
</tr>
</tbody>
</table>

3.4 Parameters

NSGA-II and NSGA-III have very similar parameters, the only difference is that
NSGA-III does not directly specify the population size. It is instead based on
the number of reference points. The reference points are systematically created
by placing them on a normalized hyperplane as described in [2]. To obtain the
number of reference points created by this method the following equation is used:

\[ H = \binom{M-1+\text{divisions}}{\text{divisions}} \]

1. Population size for NSGA-II (\( \text{pop} \)): An integer in the range \([2, 300]\). Upper
   bound determined by small scale experiments that showed all optimizations
   used a population size less than 300.
2. Divisions for NSGA-III (\( \text{divisions} \)): The number of divisions along each
   objective. The population size is set to exactly the number of reference points
   created by the divisions. An integer in the range \([1, 299]\). Upper bound set
to 299 to get the same population size limits as for NSGA-II.
3. Mutation probability (\( p_m \)): The probability of random changes to the deci-
   sion variables. A real-value in the range \([0, 1]\).
4. Mutation Distribution Index (\( \eta_m \)): Index governing the proximity of the
   mutated child to its parent. A real-value in the range \([0, 300]\). Upper bound determined by small scale experiments that showed all optimizations used a
   \( \eta_m \) less than 300.
5. Crossover probability (\( p_c \)): The probability of creating offspring from par-
   ents. A real-value in the range \([0, 1]\).
6. Crossover Distribution Index (\( \eta_c \)): Index governing the proximity of the
   mutated children to the parents. A real-value in the range \([0, 300]\). Upper bound determined by small scale experiments that showed all optimizations
   used a \( \eta_c \) less than 300.
The parameters of the optimization on the algorithm layer in Equation (1) become variables for the optimization on the design layer. Thus the variable vector $p$ in Equation (1) is $p = \{pop, p_m, \eta_m, p_c, \eta_c\}$ for NSGA-II and $p = \{divisions, p_m, \eta_m, p_c, \eta_c\}$ for NSGA-III.

### 3.5 Performance Measure

The hypervolume measure is used to assess the performance of the EA at the algorithm layer. The hypervolume is the volume in objective space formed by a reference point and the Pareto front. The hypervolume is calculated using the technique described in [11], which also discusses the hypervolume measure in more detail. The advantage of the hypervolume measure is that provides a single measure for both the convergence and spread of the solutions. The drawback is that it can be computationally expensive and that it can be sensitive to inclusion, or exclusion, of extremal points. Each EA keeps a Pareto archive of unlimited size that is used to calculate the hypervolume at the end of the optimization. So even though no limit was set for the archive size it is of course limited in practice by the available memory and running complexity of the hypervolume calculation. Neither proved to be a problem for the experiments in this study.

### 4 Experimental Results

This section will present the results from the experiments. Due to the large number of experiments conducted, totally 350, only a subset of all results can fit in this paper. The results for the most common problem size, 30, are shown in Table 3 and Table 4 for NSGA-II and NSGA-III respectively. The values are the median together with the standard deviation.

The experiments were run on a heterogeneous cluster of commodity hardware. In total there were 91 computers and the experiments took approximately 170 hours to complete.

#### 4.1 Population Size

Most of the experiments found that a small population size was most optimal. Many found the smallest possible size, two, to be the best. Having a small population size increases the selection pressure since only a small amount of solutions survive each generation. Thus, allowing the optimization to advance more quickly. However, this comes at the cost of diversity among the solutions, but based on the results, the ZDT problems do not seem to require much diversity among the solutions. One reason the population size can be kept small is the fact that the hypervolume is calculated from the, unlimited, Pareto archive. Using the last generation to calculate the hypervolume would in most cases result in a smaller hypervolume, since fewer solutions would be used in the calculation. An exception to the small population size is experiments with $N = 2$. This is especially true when the budget size is 100. One reason for this might be that
### Table 3. Optimal parameter values for NSGA-II with N = 30

<table>
<thead>
<tr>
<th>Budget</th>
<th>HV</th>
<th>Pop</th>
<th>pm</th>
<th>ηm</th>
<th>PC</th>
<th>ηc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZDT1</td>
<td>100</td>
<td>270</td>
<td>310</td>
<td>0</td>
<td>0</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>1164</td>
<td>1479</td>
<td>0</td>
<td>0</td>
<td>1479</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>7442</td>
<td>9660</td>
<td>0</td>
<td>0</td>
<td>9660</td>
</tr>
</tbody>
</table>

Table 4. Optimal parameter values for NSGA-III with N = 30

<table>
<thead>
<tr>
<th>Budget</th>
<th>HV</th>
<th>m, n</th>
<th>pm</th>
<th>ηm</th>
<th>PC</th>
<th>ηc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZDT1</td>
<td>100</td>
<td>270</td>
<td>310</td>
<td>0</td>
<td>0</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>1164</td>
<td>1479</td>
<td>0</td>
<td>0</td>
<td>1479</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>7442</td>
<td>9660</td>
<td>0</td>
<td>0</td>
<td>9660</td>
</tr>
</tbody>
</table>
a random search, which both NSGA-II and NSGA-III degenerates to when the population size is greater or equal to the budget, has about the same performance when the problem is easy to solve and the function evaluation budget is very limited.

The median values for the population size parameter using NSGA-II are shown in Table 5. The same observations can be made for the NSGA-III and are not shown for that reason.

4.2 Mutation Probability: 1/N Rule of Thumb

Each experiment was run with five different values for the number of decision variables. This was done to test the accuracy of the rule of thumb that the mutation rate should be set to one divided by the number of decision variables. Since each experiment was also run with different function evaluation budgets, it is also possible to see if that had any affect on the mutation probability. The usefulness of this evaluation is limited by the small number of problems used in this paper and no generalization can be made how this rule works on other problems. The mutation probabilities are also only from the best set of parameters found. Therefore, this evaluation does not test the accuracy of this rule of thumb for sub-optimal sets of parameters.

The experiment results can be divided into two groups based on the relationship between the mutation probability and the number of variables. ZDT\{1, 2, 3\} is in one group and ZDT\{4, 6\} is in the other. The first group start with a relatively high mutation probability for two variables, which then decreases and is kept almost constant for 10, 20, 30 and 40 variables. The second group has a more gradual decrease in the mutation probability. The median values from two problems are shown here, ZDT1 in Figure 1 and ZDT4 in Figure 2.

On ZDT1 the rule slightly overestimates the mutation probability for budget sizes greater than 500 when N is 10 and 20 because the optimized mutation probability does not change much between N 10 and 40. It also underestimates for all N when the budget size is less than 2000. On ZDT4 the rule overestimates when N is 2 and the budget size is greater than 500. It also underestimates for all N when the budget size is 100. For all other cases the rule matches well with the data.

Based on these results the rule of thumb is able to estimate good values for the mutation probability, especially for larger budgets, on the ZDT test problems.

4.3 Mutation Probability vs. Budget Sizes

A trend that can be observed throughout all experiments is that mutation probability decreases as the function evaluation budget increases. The trend is most prominent on ZDT\{1, 2, 3\} and less so on ZDT\{4, 6\}. Figure 5 and 6 shows the
Table 5. Experimental results for population sizes in NSGA-II

<table>
<thead>
<tr>
<th>N</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>3500</th>
<th>5000</th>
<th>6500</th>
<th>7000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZDT1</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
</tr>
<tr>
<td>ZDT2</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
</tr>
<tr>
<td>ZDT3</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
</tr>
<tr>
<td>ZDT4</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
<td>0.1 - 0.5</td>
</tr>
</tbody>
</table>

Fig. 1. Applicability of 1/N rule of thumb for pm on ZDT1

Fig. 2. Applicability of 1/N rule of thumb for pm on ZDT4
median values for NSGA-II on ZDT1 and ZDT6. The results for NSGA-III are similar, but they are not shown here because of space limitations.

4.4 Budget Size, $p_c$ and $\eta_c$

A trend how $p_c$ changes with respect to the budget size can be observed on all problems except ZDT4, although it is most clear on ZDT\{1, 2, 3\}. For small budgets $p_c$ is relatively high. As the budget size increases $p_c$ first falls and then rises, approaching a value of one. The point at which it starts to rise is related to the number of decision variables. Another trend is that there is a point at which an increase of the budget size causes a sharp fall of $\eta_c$. One explanation for why $p_c$ is high and $\eta_c$ is low for large budget sizes is based on the fact that most experiments use a population size of two. The two individuals are pushed apart by the crowding distance and with a large enough budget they will end up at each of the two extreme values. The rest of the non-dominated front is then filled by crossing these two solutions, and since they are at opposite extremes a low $\eta_c$ is preferred. These trends are shown for NSGA-II on ZDT1, ZDT2 and ZDT6 in Figure 3 and Figure 4, the values are the median.
Fig. 5. Mutation probabilities with varying budget sizes for NSGA-II on ZDT1

Fig. 6. Mutation probabilities with varying budget sizes for NSGA-II on ZDT6
Table 6. Hypervolume results for NSGA-II and NSGA-III

<table>
<thead>
<tr>
<th>Method</th>
<th>N</th>
<th>Hypervolume (10^5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>119.35 119.43 118.58 118.73 126.41 125.58</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>104.92 103.35 93.39 90.65 110.66 104.97</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>103.06 101.24 87.85 86.87 108.01 102.54</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>102.73 101.69 90.68 89.65 110.43 103.84</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>102.52 101.50 89.68 88.65 110.23 103.74</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>102.32 101.30 88.68 87.65 110.03 103.64</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>102.12 101.20 87.68 86.65 110.84 103.54</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>102.01 101.10 86.68 85.65 110.74 103.44</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>101.90 100.99 85.68 84.65 110.64 103.34</td>
</tr>
</tbody>
</table>

NSGA-II

<table>
<thead>
<tr>
<th>Method</th>
<th>N</th>
<th>Hypervolume (10^5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>119.35 119.43 118.58 118.73 126.41 125.58</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>104.92 103.35 93.39 90.65 110.66 104.97</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>103.06 101.24 87.85 86.87 108.01 102.54</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>102.73 101.69 90.68 89.65 110.43 103.84</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>102.52 101.50 89.68 88.65 110.23 103.74</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>102.32 101.30 88.68 87.65 110.03 103.64</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>102.12 101.20 87.68 86.65 110.84 103.54</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>102.01 101.10 86.68 85.65 110.74 103.44</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>101.90 100.99 85.68 84.65 110.64 103.34</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>101.80 100.89 84.68 83.65 110.54 103.24</td>
</tr>
</tbody>
</table>

NSGA-III

<table>
<thead>
<tr>
<th>Method</th>
<th>N</th>
<th>Hypervolume (10^5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>119.35 119.43 118.58 118.73 126.41 125.58</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>104.92 103.35 93.39 90.65 110.66 104.97</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>103.06 101.24 87.85 86.87 108.01 102.54</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>102.73 101.69 90.68 89.65 110.43 103.84</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>102.52 101.50 89.68 88.65 110.23 103.74</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>102.32 101.30 88.68 87.65 110.03 103.64</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>102.12 101.20 87.68 86.65 110.84 103.54</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>102.01 101.10 86.68 85.65 110.74 103.44</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>101.90 100.99 85.68 84.65 110.64 103.34</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>101.80 100.89 84.68 83.65 110.54 103.24</td>
</tr>
</tbody>
</table>
4.5 Hypervolume Comparisons Between NSGA-II and NSGA-III

The mean hypervolume values for both NSGA-II and NSGA-III are shown in Table 6. It is not possible, due to the number of experiments performed, to include all parameter settings used to obtain the hypervolume results. A subset of all the parameter settings and their corresponding hypervolumes are presented in Table 3 and Table 4. A Welch-t test with a significance of 5% is performed to determine if the two samples, NSGA-II and NSGA-III, are statistical different. If the null hypothesis can be rejected, the greater hypervolume is shown in bold.

The difference in hypervolume between NSGA-II and NSGA-III is for the most part small. However, for some of the experiments, NSGA-II is slightly better. NSGA-III is statistically better on some experiments but the difference is too small to be concluded as significant.

To summarize, NSGA-II is found to be marginally better than NSGA-III on the ZDT problems. Both NSGA-II and NSGA-III can find solutions very close to the Pareto front for ZDT\{1, 2, 3, 6\}. The most difficult problem is ZDT4, for which with \(N > 10\) none of algorithms could reach the maximum theoretical hypervolume within 8000 evaluations.

5 Conclusions and Further Work

This paper utilized a bilevel optimization framework to find optimal parameter values for two different MOEAs, namely NSGA-II and NSGA-III, for maximal performance on the ZDT test suite. Both the number of decision variables and the function evaluation budgets were simultaneously varied to determine how they affect the optimal parameter settings for the respective algorithm. This made it possible to test the rule of thumb that the mutation probability should be set to \(1/N\). The results show that, on the ZDT test problems, this rule is a good heuristic.

The experiments also made it possible to see what affect the different function evaluation budgets has on the optimized parameters. An important observation was that the optimal mutation probability is not only dependent on the number of decision variables but also on the available budget size. Specifically, it was observed that the optimal mutation probability decreases with increasing budget.

It was also clear from the results that the ZDT test problems do not require much diversity in the population because most experiments found the optimal population size to be less than 10, often close to the minimum of just two individuals. This also indicates that a parameter controlling the elitism should have been included in the experiments.

Another aim of this paper was to compare the performance between NSGA-II and NSGA-III on the ZDT test problems. From the results, it is possible to discern a slight advantage with NSGA-II over NSGA-III on the ZDT problems. As far as the optimal parameter values are concerned, it was observed that the differences are small.

Extending these experiments to scale the number of objectives instead of the number of decision variables would be interesting, and is intended as future work.
Since these results, as well as other earlier work, indicate that it is sub-optimal to keep parameter settings static during the run, it would be worthwhile to modify an EA, on the algorithm layer, to be able to use multiple sets of parameters during an optimization. This would allow a meta-EA to tune multiple sets of parameters at different intervals of the optimization, instead of being limited to a single set throughout the optimization run.

References

Tuning of Multiple Parameter Sets in Evolutionary Algorithms

Martin Andersson
University of Skövde
P.O. Box 408
Skövde, Sweden
martin.andersson@his.se

Sunith Bandaru
University of Skövde
P.O. Box 408
Skövde, Sweden
sunith.bandaru@his.se

Amos H.C. Ng
University of Skövde
P.O. Box 408
Skövde, Sweden
amos.ng@his.se

ABSTRACT
Evolutionary optimization algorithms typically use one or more parameters that control their behavior. These parameters, which are often kept constant, can be tuned to improve the performance of the algorithm on specific problems. However, past studies have indicated that the performance can be further improved by adapting the parameters during runtime. A limitation of these studies is that they only control, at most, a few parameters, thereby missing potentially beneficial interactions between them. Instead of finding a direct control mechanism, the novel approach in this paper is to use different parameter sets in different stages of an optimization. These multiple parameter sets, which remain static within each stage, are tuned through extensive bi-level optimization experiments that approximate the optimal adaptation of the parameters. The algorithmic performance obtained with tuned multiple parameter sets is compared against that obtained with a single parameter set. For the experiments in this paper, the parameters of NSGA-II are tuned when applied to the ZDT, DTLZ and WFG test problems. The results show that using multiple parameter sets can significantly increase the performance over a single parameter set.

CCS Concepts
• Mathematics of computing → Evolutionary algorithms;

Keywords
evolutionary algorithms; parameter tuning; multiple parameters; multi-objective optimization

1. INTRODUCTION
Many evolutionary optimization algorithms have parameters (population size, mutation probability, etc) that can be used to change the search behavior of the algorithm. This allows the algorithm to be adapted to different problems. However, it is often difficult to know the optimal parameter settings without performing tuning experiments. These experiments are computationally expensive, since for each parameter setting, a complete optimization has to be performed. It is therefore not realistic to expect parameter tuning experiments to be performed for every new algorithm and problem combination. Further, the optimal parameter settings will be different from problem to problem. Instead, each algorithm has certain recommended (default) parameter values and heuristics that can be used for new problems. There is however no guarantee that the default parameter values will be effective on any particular problem.

The parameters of evolutionary optimization algorithms are generally kept constant during runtime. However, studies have shown that this is sub-optimal and the efficiency of the optimization algorithm can be increased by adapting the parameters during the course of optimization [4, 8]. It is difficult to theoretically derive the optimal adaption of parameters for anything but the simplest problems, since the parameters can have complex interactions among them. The control mechanisms are therefore usually inspired by concepts in statistics and nature, in the same way as many optimization algorithms are. Since it is difficult to understand the interactions between the parameters, parameter control mechanisms usually only consider at most a few parameters. That will however make it difficult to find and exploit any beneficial interactions between them.

Instead of finding a control mechanism that directly adapts the parameters, the novel empirical approach in this paper is to estimate the optimal adaption of the parameters by tuning multiple sets of parameters. In other words, the optimization run is divided into stages, where each stage has separate, static, parameter settings. These multiple parameter sets, together with the stage separation points, are then formulated into an optimization problem and solved using an evolutionary algorithm. This approach is referred to as bilevel optimization, where the upper level optimization searches for the best parameter setting and the lower level optimization evaluates them. The idea is to let this upper level optimization find the optimal parameter setting for each stage and the points where the switch between them occurs. The idea of splitting the optimization into different stages with different parameter settings is not new [15, 1, 2]. The difference is that previous studies used online parameters that self adapted, while the parameter sets in this paper are found offline, using parameter tuning experiments.

There are two main objectives with this study. First, estimate the potential performance improvement with adapting
parameters, by tuning one, two and three multiple parameter sets. This will give a lower bound for an optimal parameter control mechanism on each problem that is tuned. Second, analyze the optimized parameter sets for patterns and other information that can be used to improve the optimization algorithm, or its parameter values, when applied to a particular problem. It is also important to note that the aim is not to find generally applicable parameters.

The experiments in this paper will tune the parameters of NSGA-II when applied to the first four problems in the ZDT [19], DTLZ [6] and WFG [9] test suites. These problems will provide a mixture of easy and hard problems, with and without local fronts. Both two objective (ZDT and WFG) and three objective problems (DTLZ) are considered.

The rest of the paper is structured as follows. Section 2 introduces the parameter setting problem. In Section 3 a description of bilevel optimization is provided. The idea behind multiple parameter sets are given in Section 4. The experimental design is explained in Section 5. The results appear in Section 6 and the conclusions are summarized in Section 7.

2. OPTIMIZATION ALGORITHM PARAMETERS

The no free lunch theorem states that no optimization algorithm can be better than all other algorithms on all problems [14]. This also applies to the parameters of an optimization algorithm. That is, no parameter setting can be better than all other parameter settings on all problems. For example, one problem might benefit from parameter settings that promote a greedy search, while the same settings on another problem may lead the algorithm to a local optima. This can be mitigated by tuning the parameters against many different problems, yielding a parameter setting that is generalist.

Whilst parameter settings tuned for a small set of problems would become specialists. The difference in performance between a generalist and a specialist for a given problem can be significant [3, 11].

The problem of setting the parameters can be broadly divided into two types:

- Parameter tuning: The parameters are set at the beginning and are kept constant during the optimization.
- Parameter control: The parameters are initialized at the beginning and are then adapted according to some rules or heuristics during the optimization.

There are a number of different parameter control mechanisms proposed for various evolutionary algorithm components; see [30] for a comprehensive overview. The mechanisms can be divided into three categories, parameter dependent, control ensembles and parameter independent methods. Parameter dependent mechanism are those that control a specific component, such as population size or variation. For simple problems, it is possible to derive strategies based on theoretical studies. Most however, draw inspiration from other areas, such as statistics or nature. Control ensembles combine multiple mechanisms, in order to balance the trade-offs between exploring and exploiting the search space.

The third category is that of parameter independent methods. They adapt the parameters based on simple rules and heuristics and can also use feedback on how well a particular parameter is performing. Parameter control mechanisms can also be considered specialists or generalists, in the sense that they do not violate the no free lunch theorem. However, it is reasonable to assume that because of their greater flexibility they can provide better trade-offs than static parameters.

Dividing the optimization in multiple stages, where each stage uses independent parameters is not a new concept. In [15], three parameter values are used to adapt a control parameter \(p_x\) or \(p_y\). The optimization run is divided into epochs and each epoch is further divided into two periods. In the first period, crossover and mutation are performed according to one of three parameters that is picked randomly. In the second period, the parameters are proportionally chosen based on their performance in the first period. Another approach is to use time series prediction to dynamically adjust values during the optimization [1]. Each parameter is divided into intervals and new values are sampled from these intervals. At the end of each iteration, the success probability of each parameter value is calculated. These probabilities are then used to adjust the parameter values in the next iteration. This method was extended in [2] with dynamically adjusted interval ranges using a meta-algorithm.

All of these studies use self-adapting parameters that determine what values the parameters should take in each stage. The approach adopted in this paper is different in that, each stage has static values that are set by an upper level optimization. In this regard it is similar to the method used in [2], which used what the authors call a meta-algorithm to adjust the interval ranges for the parameter values. The advantage of not imposing any direct control mechanism on the parameters is that they are free from any assumptions of the algorithm developer. By increasing the number of stages, the approximation of the optimal adaptation is improved. The disadvantage is that the upper level optimization becomes harder when the number of stages is increased, because it becomes even more difficult to find the optimal mix of parameter sets.

3. PARAMETER TUNING USING BILEVEL OPTIMIZATION

Parameter tuning can be divided into three layers: The application layer, the algorithm (lower) layer and the design (upper) layer [7]. On the application layer is the problem that is being solved and the optimization algorithm used to solve it is on the algorithm layer. The tuner that tunes different parameters for the optimization algorithm is on the design layer. The quality of solutions on the application layer is called fitness and the quality of the parameters is called utility [7].

The parameter tuning problem can itself be viewed as an optimization problem, in which the objective is to maximize the performance of the algorithm when used to solve a particular problem. For single-objective problems, the performance indicator could be directly related to the best function value attained by the algorithm. For multi-objective problems there is no single best solution, instead the output is a set of non-dominated solutions. The quality of the non-dominated set is usually measured in terms of its diversity and convergence. In order to avoid a multi-objective problem at the design layer, only unary performance indicators are considered. Two such indicators are inverted generational distance (\(I_{IGD}\)) [16] and hypervolume (\(I_H\)) [17].
1. The inverted generational distance of a non-dominated solution set $S$ obtained from an MOEA is defined as,
\[
I_{IGD}(S,P) = \frac{1}{|P|} \sum_{x \in S} \min_{x' \in P} |F(x) - F(x')|,
\]
where $P$ is a reference set of Pareto-optimal solutions that represent the desired density and distribution of points on the Pareto front, $\parallel \cdot \parallel$ denotes the Euclidean distance, and $|S|$ denotes the size of a set. Solution sets with lower $I_{IGD}$ values are preferred since they closely resemble the reference set $P$. Thus, the best possible value for $I_{IGD}$ is 0.

2. The hypervolume of a non-dominated solution set $S$ obtained from an MOEA is defined as,
\[
H(S,r) = \lambda \left( \bigcup_{x \in S} H(P(x), r) \right),
\]
where $r$ is a reference point that is dominated by all solutions in $S$, $\lambda$ is the Lebesgue measure, and $H(P(x), r)$ represents the hypercube with body-diagonal $ab$. Solution sets with higher $H$ values are better because they dominate a larger region of the objective space. Hypervolume is the only known unary quality indicator that is Pareto-compliant [18], i.e., for any two solution sets $S_1$ and $S_2$ ($\neq S_1$), $S_1 \subseteq S_2 \Rightarrow I_{IGD}(S_1) > I_{IGD}(S_2)$.

The IGD indicator will be used in this paper because it is faster to compute than the hypervolume, which is important because of the extensive parameter tuning experiments planned.

The utility of a parameter setting is the number of function evaluations the optimization algorithm, on the algorithm layer, uses to reach a pre-specified target IGD value. An alternative would be to measure the IGD value after a maximum number of function evaluations. Both approaches have their advantages and disadvantages and both require a value (the budget or target IGD) to be specified before the optimization is started. The maximum budget value is crucial because the number of function evaluations to reach a particular IGD value will be different for different parameter settings. A very high maximum budget value, in addition to increasing computational time, also enables more parameter settings to reach the target IGD value. This in turn makes it difficult to differentiate between them. In other words, a very high budget washes out any differences between good parameter settings. This is the reason the utility is chosen as the number of function evaluations to reach a target IGD value. However, the need for a maximum budget is not completely eliminated, since some parameter settings may never reach the target IGD.

There are many different parameter tuning methods, see [7] for an overview. The approach in this paper is to formulate the tuning of the multiple parameter sets as a single objective optimization problem and solve it using an evolutionary algorithm. This is referred to as a Meta-EA [7] or bilevel optimization [12].

Minimize $\mathcal{FE}(p)$

where, $\mathcal{FE}(p)$ is the mean number of function evaluations needed to reach a target IGD value when the following problem is solved with parameters $p$.

Minimize $\{F(x) = f_1(x), f_2(x), \ldots, f_n(x)\}$

Subject to $g_j(x) \geq 0 \forall j \in \{1, 2, \ldots, J\}$

$h_k(x) = 0 \forall k \in \{1, 2, \ldots, K\}$

$x_k \leq x \leq u_k$

(3)

The algorithmic parameters of the lower-level optimization problem become the variables for the upper-level optimization problem. Each additional set of parameters adds one variable for each parameter, plus one switching point.

4. MULTIPLE PARAMETER SETS

The main idea of this paper is to search for the best sequential combination of different parameter sets. The parameters provide a mechanism to balance the trade-off between exploration and exploitation of the search space. The hypothesis is that more parameter sets would allow the optimization to start with a parameter setting that would suit the exploration phase and then switch to another parameter setting when it enters the exploitation phase. A failure to find any improvement of two or more parameter sets would indicate that an optimization algorithm and problem combination might only have one phase. In other words, parameters that are the most effective in the beginning are also the most effective in the end. Such conclusion would, of course, be tied to a particular problem and algorithm combination. Nevertheless, that combination would probably not be a good choice for evaluating a new parameter control mechanism. Since it is not certain that there will be an advantage in using adaptive parameters on it. On the other hand, if an improvement is found with two or more parameter sets, the performance increase would be a lower bound for an optimal parameter control mechanism. Further, analyzing the changes in the parameter values between sets could provide enough information to generalize it into a parameter control mechanism.

There are several possible strategies for determining when to switch between parameter sets. The simplest is to keep the switching points static, by dividing the optimization run into equally sized parts. This means that the switching points would be the same across all problems. It is however unlikely that the optimal switching points will be the same for all problems. Another option is to switch after a certain number of function evaluations have been performed. The drawback of that approach is that it forces the switch to happen at the same time for every replication. The approach taken in this paper is instead to switch parameter sets when the IGD falls below a certain value. This threshold IGD value is added as a variable to the upper level optimization, so that it can be different from problem to problem. This also means that there are $s - 1$ switching variables for $s$ number of parameter sets.

The experiments in this paper use a version of the NSGA-II [5] algorithm that is modified to be used with multiple parameter sets. The crossover operator is SBX and the mutation is polynomial. Six different parameters of NSGA-II are included in the experiments. The following section pro-
vides a short description of them and their respective variable ranges.

1. Population size (pop): The number of individuals in the population. An integer in the range [2, 200].
2. Child population (child): The number of children created each generation. Specified as a percentage of the population size. An integer in the range [2, 200].
3. Mutation probability (pm): The probability of random changes to the decision variables. A real-value in the range [0, 1].
4. Mutation Distribution Index (eta): Index governing the proximity of the mutated child to its parent. Larger values mean a smaller change to the original value. A real-value in the range [0, 1].
5. Crossover probability (pc): The probability of creating offspring from two parents. A real-value in the range [0, 1].
6. Crossover Distribution Index (eta): Index governing the proximity of the children to the their parents. Larger values mean the children are closer to its parents. A real-value in the range [0, 1].

Thus the variable vector p in Equation (3) is,

\[ p = (\text{pop}^{(1)}, \text{child}^{(1)}, \eta^{(1)}, \eta^{(2)}, \text{pc}^{(1)}, \text{pc}^{(2)}, \ldots, \text{pop}^{(k)}, \text{child}^{(k)}, \eta^{(k)}, \eta^{(k)}, \text{pc}^{(k)}, \text{pc}^{(k)}) \]

where \( k \) is the number of parameter sets and \( \text{pop}^{(k)}, \text{child}^{(k)}, \eta^{(k)}, \eta^{(k)}, \), \( \text{pc}^{(k)}, \text{pc}^{(k)} \) is the \( k \)-th parameter set to the \( k \)-th. The switch variables are normalized in the range [0, 1] as described later in the following section.

5. EXPERIMENT DESIGN

The tuning experiments in this paper are performed against NSGA-II on the algorithm layer and the first four problems in the ZDT, DTLZ and WFG test suites at the application layer. ZDT and WFG are configured for two objectives, while DTLZ have three objectives. The number of decision variables for ZDT is 30, 7 for DTLZ1, 12 for DTLZ3, and 4 with 4 positional variables for WFG. The parameters of NSGA-II will be tuned against all these problems with one, two and three parameter sets, for a total of 36 parameter tuning experiments. As a reference, the following recommended (default) parameters are also evaluated against the same problems: \( \text{pop} = 100, \text{pc} = 1/N, \eta = 0.9, \text{pop} = 20, \eta = 0.9, \text{pop} = 20 \) and \( \text{child} = 1 \).

Since NSGA-II is a stochastic algorithm, each parameter setting evaluation have to be replicated in order to accurately estimate its utility. The experiments in this study use the estimated mean, \( \overline{IGD}(\text{pop}) \), from 10 replications as the utility of a parameter setting p. Each replication uses a unique random seed, but the set of seeds are the same for different parameter setting evaluations within the same design layer experiment. This means that two parameter settings with the same population size will always generate the same initial population. Thus, reducing the impact (which can be substantial) of the starting conditions on the utility, making it easier to determine if any difference in utility is due to a parameter setting change or by random chance. The drawback is that the parameters will be tuned for a specific set of seeds. Hence, the number of replications must be high enough so that the tuned parameters do not overfit the seeds.

Initial experiments showed that 10 replications provide a good trade-off between computational effort and risk of overfitting. Each design layer optimization is also replicated 5 times and the best parameter setting from each experiment is replicated 500 times on a unique set of random seeds. This further ensures that the utility of the best parameter setting found through bilevel tuning does not overfit a particular set of random seeds. The default parameter setting as well as the tuning experiments are also replicated with the same random seeds, so that the comparison between them is as fair as possible. The maximum budget is also raised for each problem by a factor of ten. This reduces the risk that a particular optimization terminates because of the limit on the maximum budget and not because the target IGD value was reached.

The difficulty of reaching a target IGD value depends on the actual target itself and the IGD reference set. The more points in the reference set, the better the Pareto front approximation is. The problem is to reduce the runtime of the algorithm while still achieving a good diversity and convergence of the non-dominated solutions. The IGD targets cannot be too difficult to reach, because it would make the runtime of the parameter tuning experiments prohibitively expensive. The difficulty of reaching a particular target IGD also depends on the problem. That is why for each problem we use a different number of IGD reference points, different IGD targets and different maximum function evaluations as shown in Table 1. Even though the IGD target is the primary stopping criteria, each experiment has a maximum budget so that it is guaranteed that the optimization algorithm will eventually stop. As mentioned before, the switch variables are normalized in the range [0, 1], where 1 corresponds to switch occurring just as the target IGD is reached and 0 corresponds to the IGD at initialization of the population. In order to estimate a value for the latter, Latin hypercube sampling is used to generate 100 parameter settings, from which the initial IGD values are calculated. Problems ZDT4, DTLZ3 and WFG1, have a high initial IGD in relation to the target IGD. This causes the switch variable to have a very low precision close to the target IGD, because the switch variable is assumed to have a linear mapping whereas practically the IGD decreases nonlinearly. The standard GA operators have difficulty making such small changes to the switch variable. For this reason the initial IGD value is capped at 5 for these problems as shown in Table 1. This implies that no switch can occur until the IGD value is less than 5. The values used for each problem are also shown in Table 1.

The single objective optimization problem at the design layer, shown in Equation 3, is noisy (the lower level optimizations are stochastic), has interacting variables (such as, population size and crossover probability), and a non-linear objective function with many local optima. These characteristics make the upper level optimization challenging. However, EAs are known to be robust enough to handle such problems. Therefore, the upper level algorithm used in
Table 1: IGD and maximum budget experiment settings for the included test problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Target IGD</th>
<th>Initial IGD</th>
<th>IGD points</th>
<th>Maximum budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTLZ1</td>
<td>0.001</td>
<td>2.54</td>
<td>100</td>
<td>200000</td>
</tr>
<tr>
<td>WFG1</td>
<td>0.001</td>
<td>3.32</td>
<td>200</td>
<td>500000</td>
</tr>
<tr>
<td>WFG2</td>
<td>0.001</td>
<td>2.87</td>
<td>100</td>
<td>200000</td>
</tr>
<tr>
<td>WFG3</td>
<td>0.001</td>
<td>3.87</td>
<td>500</td>
<td>1000000</td>
</tr>
<tr>
<td>WFG4</td>
<td>0.001</td>
<td>2.07</td>
<td>100</td>
<td>200000</td>
</tr>
<tr>
<td>ZDT1</td>
<td>0.001</td>
<td>3.07</td>
<td>500</td>
<td>1000000</td>
</tr>
<tr>
<td>ZDT2</td>
<td>0.001</td>
<td>3.32</td>
<td>500</td>
<td>1000000</td>
</tr>
<tr>
<td>ZDT3</td>
<td>0.001</td>
<td>3.32</td>
<td>500</td>
<td>1000000</td>
</tr>
<tr>
<td>ZDT4</td>
<td>0.001</td>
<td>3.32</td>
<td>500</td>
<td>1000000</td>
</tr>
<tr>
<td>ZDT5</td>
<td>0.001</td>
<td>3.32</td>
<td>500</td>
<td>1000000</td>
</tr>
</tbody>
</table>

Table 2: The median number of FEs to reach the target IGD value in 100 optimization runs.

<table>
<thead>
<tr>
<th>Problem</th>
<th>ZDT1</th>
<th>ZDT2</th>
<th>ZDT3</th>
<th>ZDT4</th>
<th>ZDT5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>3.76E+05</td>
<td>5.33E+04</td>
<td>3.51E+04</td>
<td>5.33E+04</td>
<td>3.51E+04</td>
</tr>
<tr>
<td>OneSet</td>
<td>2.33E+03</td>
<td>8.51E+02</td>
<td>7.92E+02</td>
<td>8.18E+02</td>
<td>3.15E+02</td>
</tr>
<tr>
<td>TwoSets</td>
<td>2.31E+03</td>
<td>8.50E+02</td>
<td>7.92E+02</td>
<td>8.18E+02</td>
<td>3.15E+02</td>
</tr>
<tr>
<td>ThreeSets</td>
<td>2.31E+03</td>
<td>8.50E+02</td>
<td>7.92E+02</td>
<td>8.18E+02</td>
<td>3.15E+02</td>
</tr>
</tbody>
</table>

In this section, we present the performance outcomes and the best parameter settings. The default parameter setting is unable to reach the target IGD in any of the 100 replications, for ZDT4, as seen in the bottom right of Figure 1, and WFG2, in top right of Figure 3. Increasing the number of parameter sets from one to two, further reduces the number of FEs for most problems. However, it is also clear that not all problems benefit from an additional parameter set, which is even more evident in the results of three parameter sets. Only three of the problems, ZDT1, WFG2 and WFG4, have significant reductions. The other problems are either worse or not statistically different from either one or two parameter set(s). In theory, the performance should never get worse when increasing the number of parameter sets, because a larger number of parameter sets could always exactly mimic a smaller number. In practice this can happen because there is a stochastic algorithm at the upper layer that can fail to find the optimal parameter setting. In fact, this is more likely to happen with more parameter settings, since the upper level problem becomes harder. This can be observed in the results.

In this section, we present the performance outcomes and the best parameter settings. Four parameter settings are presented, the default (recommended), one, two and three parameter sets. The column preceding each setting shows the reduction in function evaluations in percentage with respect to the setting on the left. A Kruskal-Wallis test at 95% confidence in performed to determine if the settings are statistically different. The rejection of the null hypothesis is followed by a post-hoc Nemenyi test [11]. The settings that are statistically indistinguishable from the best median values are shown in bold in Table 2.
Figure 1: Box-and-whisker plots showing the number of FEs to the target IGD value in 100 optimization runs, for ZDT1, ZDT2, ZDT3 and ZDT4.

Table 3: The best parameter configurations using one parameter set.

<table>
<thead>
<tr>
<th>Function</th>
<th>pop</th>
<th>p</th>
<th>m</th>
<th>ηm</th>
<th>pc</th>
<th>ηc</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZDT1</td>
<td>2.00E+00</td>
<td>2.37E-02</td>
<td>5.65E-02</td>
<td>9.98E-01</td>
<td>9.73E-02</td>
<td>7.94E-01</td>
<td></td>
</tr>
<tr>
<td>ZDT2</td>
<td>2.00E+00</td>
<td>2.40E-02</td>
<td>1.04E+00</td>
<td>1.00E+00</td>
<td>2.05E-02</td>
<td>4.77E-01</td>
<td></td>
</tr>
<tr>
<td>ZDT3</td>
<td>1.20E+01</td>
<td>1.84E-02</td>
<td>3.94E-01</td>
<td>9.72E-01</td>
<td>1.54E+01</td>
<td>3.62E-01</td>
<td></td>
</tr>
<tr>
<td>ZDT4</td>
<td>1.80E+01</td>
<td>3.98E-02</td>
<td>2.32E-03</td>
<td>9.93E-01</td>
<td>1.75E+01</td>
<td>1.01E-01</td>
<td></td>
</tr>
<tr>
<td>DTLZ1</td>
<td>7.70E+01</td>
<td>5.50E-02</td>
<td>1.08E+01</td>
<td>9.53E-01</td>
<td>1.38E+01</td>
<td>9.47E-01</td>
<td></td>
</tr>
<tr>
<td>DTLZ2</td>
<td>4.20E+01</td>
<td>3.22E-03</td>
<td>1.97E+02</td>
<td>1.00E+00</td>
<td>1.60E+00</td>
<td>1.94E-01</td>
<td></td>
</tr>
<tr>
<td>DTLZ3</td>
<td>5.20E+01</td>
<td>2.33E-02</td>
<td>8.90E+00</td>
<td>9.99E-01</td>
<td>5.42E+00</td>
<td>5.74E-01</td>
<td></td>
</tr>
<tr>
<td>DTLZ4</td>
<td>6.20E+01</td>
<td>6.11E-02</td>
<td>1.91E+02</td>
<td>9.91E-01</td>
<td>2.93E+00</td>
<td>1.93E-01</td>
<td></td>
</tr>
<tr>
<td>WFG1</td>
<td>2.90E+01</td>
<td>3.66E-02</td>
<td>7.45E+01</td>
<td>8.43E-01</td>
<td>4.04E+01</td>
<td>2.62E-01</td>
<td></td>
</tr>
<tr>
<td>WFG2</td>
<td>2.00E+00</td>
<td>1.99E-01</td>
<td>5.26E-01</td>
<td>1.48E-01</td>
<td>1.05E+02</td>
<td>9.19E-01</td>
<td></td>
</tr>
<tr>
<td>WFG3</td>
<td>2.00E+02</td>
<td>1.58E-02</td>
<td>1.10E+02</td>
<td>9.97E-01</td>
<td>2.91E+00</td>
<td>6.88E-01</td>
<td></td>
</tr>
<tr>
<td>WFG4</td>
<td>8.00E+00</td>
<td>2.16E-02</td>
<td>1.97E+01</td>
<td>7.59E-01</td>
<td>1.22E+02</td>
<td>1.80E-01</td>
<td></td>
</tr>
</tbody>
</table>
The best parameter configuration for two parameter sets is shown in Table 4. The top row is the first parameter set and bottom row is the last. The switch column displays the dominant IGD threshold at which the optimization shall switch parameter values. The population size tends to decrease from the first to second parameter set. For example, in WFG2 it starts at 194 in the first set and goes down to 48 in the second set. In those problems, both the mutation probability and population size are lower for the second parameter set than the first. The crossover probability is close to 1 for all problems, except WFG(1, 2, 4).

For every problem the mutation probability decreases in the second parameter set. Also, for most problems the mutation distribution index goes from smaller to larger values. This is consistent with the idea of exploration and exploitation phases, where more and stronger mutations are preferred in the beginning of the search. It is difficult to estimate the number of function evaluations before the parameter set switch occurs, because the actual initial IGD values are not known and the IGD does not decrease linearly. This makes it difficult to analyze the switch variable. However, the switch does not occur very close to any of the target IGDs, indicating that the switch is not towards the end of the optimization. The switch values also differ from problem to problem, showing that there is no single IGD threshold value that is optimal for all problems.

In Table 5 the best parameter settings for three parameter sets are shown. The order in which the parameter set is used is fixed. This means that a search with switch value 1 is lower for the second parameter set than the third, the second was only in effect for one generation. In other words, the optimization was effectively only using two sets. The problems that showed the best improvement with three sets was WFG(2, 4). In those problems, both the mutation probability and population sizes decreased (except for the second mutation probability in WFG2) from the first to last set. The starting and ending values, especially population size and mutation probability, resemble those from the best two parameter sets experiment, but it is difficult to draw any firm conclusions from that.

It is difficult to generalize the results from the parameter tuning experiments into one or more parameter control mechanisms, because there are no discernible patterns that are present in all problems. There are however some patterns, like decreasing mutation probability, seen across the parameter sets that warrant further investigation.
7. CONCLUSIONS
This paper used multiple parameter sets to estimate the optimal adaption rates of the parameters of the NSGA-II algorithm. One, two and three parameter sets were tuned against the first four problems in the ZDT, DTLZ and WFG test suites. The aim is to estimate the potential performance improvement with adapting parameters and to analyze the best parameter settings for patterns that could be generalized to a control mechanism, which would be addressed in a future work.

The results show that significant improvement can be achieved with two parameter sets on most problems. However, there were also problems that failed to show an improvement. This indicates that for those algorithm-problem combinations, there are no discernible distinctions between different phases of optimization. In other words, parameters that are optimal in the beginning of the search are also optimal towards the end. However, for problems that showed an improvement with two parameter sets, it was observed that the first set of parameters promoted exploration while the second set favored exploitation. For example, the first parameter sets had relatively high population sizes, high mutation probabilities and low mutation distribution indices. Conversely, the second set had lower population sizes, lower mutation probabilities and higher mutation distribution indices. Only two problems showed a significant advantage of using three parameter sets, WFG2 and WFG3. This could mean that either there are better parameter settings that upper level optimization failed to find, or there is limited use for three or more parameter sets. More experiments are needed to empirically show the latter. However, the studies in this paper indicate that most of the advantage of adapting parameters can be had with just two parameter sets.

The tuned parameter sets were also compared against a default (recommended) parameter setting. There are significant performance improvements to be had with even a single set of tuned parameters, as expected with specialist parameter settings. For future work, a comparison between tuned generalist and specialist parameter settings, would reveal the trade-off between performance and generality.

8. REFERENCES
Towards Optimal Algorithmic Parameters for Simulation-Based Multi-Objective Optimization

Martin Andersson, Sunith Bandaru, Amos H.C. Ng

School of Engineering Science
University of Skövde
PO Box 408, 541 28 Skövde, SWEDEN
Email: firstname.lastname@his.se

Abstract—The use of optimization to solve a simulation-based multi-objective problem produces a set of solutions that provide information about the trade-offs that have to be considered by the decision maker. An incomplete or sub-optimal set of solutions will negatively affect the quality of any subsequent decisions. The parameters that control the search behavior of an optimization algorithm can be used to minimize this risk. However, choosing good parameter settings for a given optimization algorithm and problem combination is difficult. The aim of this paper is to take a step towards optimal parameter settings for optimization of simulation-based problems. Two parameter tuning methods, Latin Hypercube Sampling and Genetic Algorithms, are used to maximize the performance of NSGA-II applied to a simulation-based problem with discrete variables. The strengths and weaknesses of both methods are analyzed. The effect of the number of decision variables and the function budget on the optimal parameter settings is also studied.

I. INTRODUCTION

Simulation models of real-world processes allow for rapid experimentation with minimal disturbance to the production. However, manually constructing and running experiments becomes infeasible when the scope of the experimentation grows. An alternative to manual experimentation is to formulate the question to be answered as an optimization problem and solve it with an optimization algorithm. Since the optimization results are the foundation of any subsequent decisions, it is crucial that optimization algorithm is able to find solutions that allow the decision maker to come to the correct conclusions. One way of improving the performance of optimization algorithms is to use an optimal set of parameters.

A. Parameter Settings

The no free lunch theorem states that no optimization algorithm can be better than all other optimization algorithms on all problems [1]. This means that, in practice, there can be significant performance differences between different optimization algorithms applied to the same problem, or even between the same optimization algorithm applied to different problems. Furthermore, most optimization algorithms have parameters that are used to alter its search strategy. The values of these parameters can greatly affect the performance of the optimization algorithm [2], [3]. The parameters allow the optimization algorithm to be adapted to suit different problems. For example, an algorithm could have a parameter that modifies the trade-off between exploration and exploitation of the search space. More exploration could be beneficial on a problem with many local optima, while an exploitative behavior would be preferable on a problem with few local optima.

Thus, the no free lunch theorem also applies to the selection of the parameter settings of an optimization algorithm. That is, no parameter settings can be better than all other parameter settings for all problems. Each optimization algorithm has its own set of parameters, some of which have only a minor effect on the search behavior, while others can drastically alter it. Selecting inappropriate parameter settings can mean that the search space is not properly explored, either because of time constraints or some inherent property of the algorithm. This leaves the decision maker with incomplete and sub-optimal information, which could negatively affect the quality of any decisions based on that information. It is therefore important to choose the optimization algorithm and its parameter settings wisely. However, without a priori knowledge, this is a difficult task.

Most optimization algorithms have recommended (default) settings for their parameters. The default settings are designed to work across a range of problems and are usually a good starting point for further experimentation. There are also heuristics (rules of thumb) that can be used to choose appropriate parameter settings for new problems. These heuristics often derive the settings from some characteristic of the problem, for example, the number of decision variables. Even though the default settings are meant to perform well over a wide range of problems, there are no guarantees that they will work well for any given problem.

Most default parameter settings are derived from experiments on real-valued problems. As an example, NSGA-II [4] used only real-valued problems, ZDT [5] among others, as benchmarks. No discrete problems were included in those experiments. However, discrete decision variables are common in certain domains, such as production system simulations, inventory management and queuing.

It is difficult to know how transferable the performance of a particular set of parameter settings are from one problem to another. Each problem has its own characteristics that affect the optimal parameter settings. Although there exist well-defined categories of problems, there exist no definitive guidelines on what type of parameter settings are most effective for a
particular category. This is further complicated by the fact that this knowledge is specific for each optimization algorithm.

B. Multi-Objective Optimization Problems

An optimization problem is multi-objective when it has two or more conflicting objectives that are simultaneously optimized. This type of problem arises in many contexts. For example, in optimization of production systems, maximizing the throughput is often in conflict of minimizing the work in process. This paper considers multi-objective optimization problems as described in Equation (1), where \( M \) is the number of objectives, \( g \) represents inequality constraints, and \( h \) represents equality constraints. \( x \in \mathbb{R}^N \) is vector of \( N \) decision variables and \( f \) represent objective functions which map a solution in the decision space to the objective space as \( f : \mathbb{R}^N \rightarrow \mathbb{R}^M \).

\[
\begin{align*}
\text{Minimize} & \quad \{ f_1(x), f_2(x), \ldots, f_M(x) \} \\
\text{Subject to} & \quad g_j(x) \geq 0 \forall j \in \{1, 2, \ldots, J\} \\
& \quad h_k(x) = 0 \forall k \in \{1, 2, \ldots, K\} \\
& \quad x_i \leq x \leq x_u,
\end{align*}
\]

(1)

With conflicting objectives there is no single best solution. The concept of domination is used instead to differentiate solutions in multi-objective problems. A solution \( u \) is said to dominate another solution \( v \) if \( u_i \leq v_i \) for all \( i = 1, \ldots, M \) and \( u \neq v \). When neither of two solutions dominate each other, they are said to be non-dominating. The result of a multi-objective optimization is a set of solutions called the non-dominating set and the set of all feasible non-dominated solutions is called the Pareto-optimal set. There are two main properties that are desired of a set of non-dominated solutions. The convergence of a non-dominated set is a measure how close it is to the Pareto-optimal set and the spread measures the diversity of the solutions within the set. The hypervolume [6] combines both convergence and spread into a single measure. It measures the volume formed by a reference point and the non-dominated set. The main advantage of the hypervolume metric is that it does not require the knowledge of the Pareto-optimal set. The drawback of the hypervolume metric is that it is sensitive to the inclusion or exclusion of extremal points and the relative ranges of the objectives.

C. Factors That Affect the Optimal Parameter Settings

There are also other factors beside the problem itself that can affect the optimal parameter settings. For example, the number of decision variables (dimensions) of a problem and the maximum function evaluation budget [7]. For certain optimization algorithms there are rules of thumb that estimate optimal parameter settings based on some of these factors. For genetic algorithms there is a rule of thumb that scales the mutation probability by the number of decision variables \((1/N)\). This rule of thumb was shown to work well for NSGA-II applied to the ZDT problems [7]. It is also relevant for simulation-based models to know how the optimal parameter settings are affected by the number of decision variables.

For example, consider a simulation model that has some decision variables associated with each machine. Knowing how the optimal parameter settings change with the number of machines would allow appropriate parameter settings to be estimated directly from the problem, without doing extensive parameter tuning experiments.

D. Aim

Based on the previously discussed problems of choosing parameter settings, it is unlikely that the default parameter settings and heuristics for a given optimization algorithm are optimal for simulation-based problems with discrete variables. The main aim of this paper is to take a step towards a better understanding of how to find the optimal parameter settings for a optimization algorithm when it is used to solve a simulation-based optimization problem. This is approached from two perspectives. First, there are multiple parameter tuning methods and it is not clear which of them is the most effective. Therefore, two different methods, sampling and bilevel optimization [7] are tested in this paper. The sampling method used is Latin Hypercube Sampling (LHS). The bilevel optimization approach uses a genetic algorithm at the upper level. Both methods are used to tune a well known optimization algorithm (NSGA-II) when applied to a simulation-based problem with discrete variables. The secondary aim is to analyze the optimal parameter settings from these experiments and identify parameters that are important and look for new heuristics for simulation-based problems.

The effect of the number of decision variables and the function budget on the optimal parameter settings is also studied. The optimized parameter settings are compared against the default parameter settings, to determine the performance gain that can be obtained by tuning the parameters.

The rest of the paper is organized as follows: Section II introduces parameter tuning. In Section III, a description of the simulation model is provided. Section IV gives a short description of NSGA-II and its parameters. The experimental design appears in Section V. The experimental results are presented in Section VI and the conclusions are summarized in Section VII.

II. CHOOSING OPTIMAL PARAMETER SETTINGS

There are two approaches for choosing parameter settings:

- Parameter tuning: The settings are chosen before the optimization is started and are static for the entire run.
- Parameter control: The settings are first initialized and then adapted as the optimization progresses.

This paper focuses exclusively on parameter tuning, since the original version of NSGA-II does not have any support for parameter control. Parameter tuning experiments are computationally expensive, since they must perform a complete optimization run for each parameter setting that is being tested. Moreover, since evolutionary optimization algorithms are stochastic by nature, multiple replications must be performed to draw conclusions that are statistically sound. This is further complicated by the fact that a single simulation
can take several minutes or even hours to complete. It is difficult to perform extensive parameter tuning experiments on such expensive problems. However, when optimization of other similar models is to be performed in the future, it may be worthwhile to invest in a complete parameter tuning study on the base model to learn about the optimal parameter settings. In this paper, we consider one such base simulation model.

A. Parameter Tuning Layers

Parameter tuning can be thought of as consisting of three different layers. The problem that is being solved is on the application layer. The optimization algorithm used to solve it is on the algorithm layer and on the design layer is the tuner that tests different parameter settings for the optimization algorithm on algorithm layer. The quality of the solutions on the algorithm layer is called fitness, while the quality of the parameter settings on the design layer is called utility [8]. This paper focuses on the results of the design layer experiments (LHS vs. Bilevel optimization) and how the parameters affect the optimization algorithm at the algorithm layer (NSGA-II).

B. Generalist vs. Specialist

Another aspect of parameter tuning is that of generalized and specialized parameter settings. Generalized parameter settings are those settings that are tuned to work well over a diverse set of problems, while specialized parameters are only tuned for a specific problem. Optimization algorithms with generalized parameter settings and specialized parameter settings are also called generalist and specialist respectively [9]. There is no hard boundary between a specialist and a generalist. However, the distinction is still useful for denoting the purpose of the parameter tuning. This paper only considers specialized parameter settings.

C. Performance Measures

When performing parameter tuning experiments there must be a method of evaluating the utility of a parameter set. For single objective problems this is straightforward, since the objective value can used directly. Multi-objective problems must use some other measure since solutions (at the algorithmic layer) can be non-dominated. The advantage of single measure is that it makes it easier (at the design layer) to directly compare different parameter sets. On the other hand, using multiple measures, such as both convergence and spread, makes it possible to distinguish the trade-offs between parameter sets. In order to avoid the complexity of handling a multi-objective problem at the design layer, this paper only considers the performance as measured by the unary hypervolume metric.

D. Parameter Tuning Methods

There are several methods available for parameter tuning, two of which are discussed here, see [8] for a more complete list. Parameter tuning methods can be divided into two main categories: non-iterative and iterative tuners. A non-iterative tuner is divided into two sequentially executed steps. The first step generates all parameter settings and the second step evaluates them. Iterative tuners start with a small set of parameters which are evaluated and based on the result of those evaluations a new set of parameter settings are generated. This process is repeated until some stopping criteria is met. Iterative methods are usually better at estimating the utility landscape, while iterative methods are better at finding optimal parameter settings.

This paper evaluates two methods for parameter tuning. The first is Latin hypercube sampling [10], which is a non-iterative method that generates all samples up front. A Latin square is a two-dimensional statistical sampling technique that divides the decision space into a grid and ensures that every row and column have at least one sample. The distribution of samples within a square is random. Latin hypercube sampling is an extension of this method for an arbitrary amount of dimensions.

E. Design Layer Optimization Problem

The second method is a bilevel optimization approach [7], [11]. In this approach, the parameter tuning task is formulated as an optimization problem and solved with an evolutionary optimization algorithm. This method is also called Meta-EA [8]. The formulation of the parameter tuning problem at the design layer is shown in Equation (2) and this paper uses a genetic algorithm to solve it. Genetic algorithms are modeled based on biological evolution. They work by maintaining a population of solutions from which new solutions are generated by crossbreeding and mutating existing solutions. Selection pressure is applied to make it more probable for successful solutions to generate offspring.

Maximize \( p \)

where, \( HV(p) \) is the hypervolume of the non-dominated set obtained by solving the following problem with parameters \( p \):

Minimize \( \{f_1(x), f_2(x), \ldots, f_M(x)\} \)

Subject to

\( g_j(x) \geq 0 \quad \forall \, j \in \{1, 2, \ldots, J\} \)

\( h_k(x) = 0 \quad \forall \, k \in \{1, 2, \ldots, K\} \)

\( x \leq x_S \leq x_L \)

(2)

The algorithmic parameters of the lower level optimization problem become the variables for the upper level optimization problem. The simulation-based problem, on the application layer, solved by the lower level optimization (NSGA-II) is described in the next section.

III. A SCALABLE SIMULATION MODEL

The simple stochastic simulation model considered in the experiments of this paper represents an un-paced flow line, consisting of \( s \) workstations with \( s-1 \) inter-station buffers. For the sake of simplicity, we consider the number of machines in each workstation as one, so that the terms ‘workstation’ and ‘machine’ are interchangeable in the following discussions.
The productivity of each machine \( i \) is governed by its availability \( \alpha_i \), processing time \( \beta_i \) and repair time \( \gamma_i \). In the initial state, the workloads of all the workstations are perfectly balanced, each having a processing time of \( \beta^{orig} = 80 \) seconds per job. All machines have an availability of \( \alpha^{orig} = 0.95 \) and repair time of \( \gamma^{orig} = 300 \) seconds. The processing times are assumed to be constant, which is realistic for automated machining processes. The times to failure of the workstations are modeled with exponential distributions and the randomness of the repair times \( \gamma_i \) is modeled using Erlang distributions.

As an example, Figure 1 shows the model with \( s = 6 \) workstations/machines.

Fig. 1. A simple model representing an un-paced flow line with 6 workstations and 5 buffers.

In a complex flow line with unbalanced workloads, the detection of bottlenecks is essential for a subsequent improvement of the production rate or throughput. The location of bottlenecks depends on many factors, including the job flow logic, variability and disturbance of the machines and the buffer allocations. Even for a simple, straight flow line with balanced workloads as described above, detecting which workstation(s) to improve in order to increase the overall throughput was first proposed in [12] and later further elaborated in [13]. In such an optimization formulation, the system throughput \( TH \) is the primary objective for improvement, so that \( f_1(x) = \max_i (TH(x)) \). The total number of changes, i.e. improvement actions, can be defined as the secondary objective function, \( f_2(x) \). We consider three discrete, multi-level improvement variables \( \{ \alpha_i, \beta_i, \gamma_i \} \) that can each be either set to their original value or to an improved value.

The available improvement actions for availability, processing times and repair times and their corresponding step-sizes are:

\[
\begin{align*}
\alpha &= \{ 90, 92, 94, 96, 98 \} \\
\beta &= \{ 60, 65, 70, 75, 80 \} \\
\gamma &= \{ 180, 210, 240, 270, 300 \}
\end{align*}
\]

The second objective, \( f_2(x) \), can then be written as a summation of improvements [13]:

\[
f_2(x) = \min \left( \sum_{i=1}^{s} \alpha_i + \sum_{i=1}^{s} \beta_i + \sum_{i=1}^{s} \gamma_i \right)
\]

where

\[
\begin{align*}
\hat{\alpha}_i &= \frac{\alpha^{orig} - \alpha^{orig}}{\Delta \alpha} \\
\hat{\beta}_i &= \frac{\beta^{orig} - \beta^{orig}}{\Delta \beta} \\
\hat{\gamma}_i &= \frac{\gamma^{orig} - \gamma^{orig}}{\Delta \gamma}
\end{align*}
\]

Additionally, in order to simultaneously solve the lean buffer problem [14], we optimize the capacity of inter-station buffer spaces, \( B_i = \{ 1, 2, \ldots, 10 \} \forall i \in \{ 1, \ldots, s - 1 \} \), by adding a third objective of minimizing the total number of buffers, i.e. \( f_3(x) = \min \{ \sum_{i=1}^{s-1} B_i \} \). It is to be noted that the worst objective values for \( f_2(x) \) and \( f_3(x) \) are 12s and 10(s - 1) respectively.

The model described in this section is developed and simulated using Facts Analyzer 2.0 [15]. The decision variables related to improvement actions are implemented as multiple choice sets in the software, as described in [16].

IV. NSGA-II AND ITS PARAMETERS

The optimization algorithm that is tuned, on the algorithm layer, is NSGA-II [4]. NSGA-II is an evolutionary multi-objective optimization algorithm that mimics the process of natural evolution. It starts with a random population of individuals, where each individual represent a solution to the problem. In each generation a number of new individuals are created. A new individual is created by picking two parent individuals from the population and mixing them with a crossover operator. After the crossover a random mutation is applied to children. When all children have been created they are added to the population. To trim the population, the individuals are sorted by the non-dominated relationship as a first criteria and how crowded the individuals are as a second criteria. The crossover operator is SBX and the mutation is polynomial.

Six different parameters of NSGA-II are studied in this paper. The following section provides a short description of them and their respective variable ranges.

1) Population size \( (pop) \): The amount of individuals in the population. An integer in the range \([2, 300]\).
2) Child population \( (child) \): The number of children created each generation. Specified as a percentage of the population size. A real-value in the range \([0, 1]\).
3) Mutation probability \( (\eta_m) \): The probability of random changes to the decision variables. A real-value in the range \([0, 1]\).
4) Mutation Distribution Index \( (\eta_m) \): Index governing the proximity of the mutated child to its parent. Larger values mean a smaller change to the original value. A real-value in the range \([0, 300]\).
5) Crossover probability \( (\eta_c) \): The probability of creating offspring from two different parents. A real-value in the range \([0, 1]\).
6) Crossover Distribution Index \( (\eta_c) \): Index governing the proximity of the children to the parents. Larger values mean the children are closer to its parents. A real-value in the range \([0, 300]\).

Thus the variable vector \( p \) in Equation (2) is \( p = \{ pop, child, \eta_m, \eta_c, \eta_m, \eta_c \} \).

V. EXPERIMENTAL DESIGN

As previously mentioned, there are two aims of this paper. The first is to compare LHS and bilevel optimization as parameter tuning methods, primarily with respect to their ability of finding the parameter settings that maximize the hypervolume.
The second aim is to analyze the optimal parameter settings to determine how they scale with the number of decision variables and the function evaluation budget. This section describes what experiments are performed to achieve these aims. All experiments are performed on the previously described simulation model. In order to test the effect of decision variable scaling, four variants of this model are created. The variants have 5, 10, 15 and 20 machines, with 19, 39, 59 and 79 decision variables respectively, as described in Section III. Two function evaluation budgets at the algorithm layer are tested, 4000 and 8000. The extent of these experiments had to be limited to two different budgets because of the computationally expensive simulation. Each parameter set $p$ is replicated 3 times and the average hypervolume is the utility of that parameter set. Since the simulation model on the application layer is also stochastic, it is replicated 3 times from which the mean value of each output is calculated.

The optimization algorithm used at the design layer for the bilevel optimization approach is a genetic algorithm. This introduces the problem of choosing optimal parameter settings for the design layer optimization as well. However, this problem is out of scope and will not be addressed by this paper. The GA uses the SBX crossover operator and a polynomial mutation, with the following parameters: $pop = 100$, $p_{m} = 1/N$, $\eta_{m} = 5$, $p_{c} = 0.9$, $\eta_{c} = 20$ and $child = 0.5$. In this paper, the term GA refers to the use of a bilevel optimization approach with a genetic algorithm at the design layer.

The bilevel optimization approach is computationally expensive since for every upper level evaluation many lower level optimizations have to be performed. The number of parameter sets that are feasible to evaluate is directly related to the runtime of the optimization on the algorithm layer. Which in turn is related to the amount of simulations and the duration of a single simulation. This makes parameter tuning of real-world simulation-based problems challenging, since a single simulation can take minutes or even hours to run. Furthermore, a GA is stochastic, so replication of the experiments are needed to get statistical significant results. The number of replications have to be kept small as a consequence of the relatively expensive simulation model. The GA is replicated 3 times and the best parameter set $p^{*}$ is taken as the output. For each experiment the budget of the GA at the design layer is 10000, and with 3 replications this results in 30000 unique optimizations at the algorithmic layer. The LHS is given the same budget of 3000 evaluations as the GA. Both the GA and LHS are applied to the four different variants of the model.

### VI. Experimental Results

This section presents the results from the parameter tuning experiments. In addition to the LHS and GA tuning results, experiments with parameter settings suggested by [4] are also included as a reference to which the GA and LHS results are compared, where $pop = 100$, $p_{m} = 1/N$, $\eta_{m} = 20$, $p_{c} = 0.9$, $\eta_{c} = 20$ and $child = 1$. Both LHS and GA are applied to all of the four variants of the model, using both 4000 and 8000 as the maximum function evaluations. This results in 16 different experiments for the LHS vs. GA comparison. The best set of parameter settings from each experiment is independently replicated 10 times. The hypervolume improvements (in percent) of LHS and GA compared to the suggested default settings are shown in Figure 2. Comparing hypervolume results between experiments is not meaningful since they use different reference points. Instead it is the difference between the default, LHS and GA results that are important. For all experiments, the GA is able to find parameter settings with higher hypervolumes than LHS. Further, the gap increases as the number of machines increases. LHS also found parameter settings that performed better than the default. However, the improvements were smaller and did not increase with the number of machines.

Figure 4 (GA) and 5 (LHS) show a parallel coordinates plot of the hypervolume and all decision variables for the experiments on the model with five machines and a budget of 8000. Only those solutions that fall in the top 5% of the total range of the hypervolume are included. From these figures it is clear that the LHS method have failed to find the optimal parameter settings. All solutions from the GA have small $p_{m}$, $p_{c}$, and $\eta_{c}$ values, while LHS only found those solutions with small $p_{m}$. This also indicates that $p_{m}$ is the most important of those variables to keep small.

Figure 3 illustrates the difference in search strategy between the optimized and default parameter settings, the example uses five machines and a 4000 budget. Solutions from experiments using both optimized and default parameter settings are combined and all dominated solutions removed. The blue points are from the optimized parameters and the red points are from the default parameters. The optimized parameters are better at finding extreme values, while the default are better in a region in the middle. This pattern is observed for other experiments as well. This is partially explained by the fact that solutions at the extremes contribute more to the hypervolume than solutions in the middle. This mean that parameter settings that are good at finding the extremes will have a relatively high utility and will therefore be preferred by the GA.

The optimized parameter settings from the GA experiments are shown in Table I and II, for budgets of 4000 and 8000 respectively. Each row represents one set of optimized parameter settings, $p_{i}$, where $i$ is the number of machines and $j$ is the replication. The optimized values for $\eta_{c}$ is, with a few exceptions, small. In fact, most $\eta_{m}$ are close to zero which is less than the default value of 20. Values for $p_{c}$ with a 4000 budget are high, mostly above 90%. The results for the 8000 budget are similar, but with a few smaller values. Overall the optimized $p_{c}$ values match well with default value of 0.9. The variance of $\eta_{c}$, $pop$ and $child$ parameters are too high to draw any conclusions from. Although, $\eta_{c}$ values are in general above 100. From the hypervolume results it is also clear that the GA has for some replications found sub-optimal parameter settings. This indicates that the parameter tuning problems have many local optima and that further tuning of
For evolutionary optimization algorithms, there is a rule of thumb that the mutation probability should be set to $1/N$, where $N$ is the number of decision variables. Figure 6 and 7 show how good the rule is at estimating the optimal mutation probability for a simulation-based problem with discrete variables. Optimized mutation probabilities for each of the three GA replications as well as the LHS results are included. The results show that the optimized values for $p_m$, in general, do in fact decrease as the number decision variables increase. However, the rule consistently underestimates the optimal mutation probability by approximately a factor of two.

The function evaluation budget does not appear to have a large effect on the optimal parameter settings, based on the results in Table I and II. However, some experiments show a greater range of values. This is shown in Figure 8 (4000) and 9 (8000) for experiments with 15 machines. Only those solutions that fall in the top 5% of the total range of the hypervolume are included. Here $pop$, $p_c$, $\eta$, and $\text{child}$ have larger ranges with a budget of 8000 compared to 4000.

VII. CONCLUSIONS

The main aim of this paper is to take a step towards more optimal parameter settings for simulation-based optimization. The approach is to perform parameter tuning experiments on four different variants of a simulation-based model with discrete variables. The only difference between the models are the number of machines, allowing for an analysis of the effect of decision variable scaling on the optimal parameter settings.

A sampling method, LHS, is compared with a bilevel optimization approach, for their ability of finding parameter sets that maximizes the hypervolume of NSGA-II applied to a simulation-based problem. The results show that for both 4000 and 8000 evaluations, bilevel optimization is better than LHS at finding optimal parameter settings. However, both
methods improved upon the hypervolume values obtained by the default parameter settings. The results also show that the performance gap between the two methods widen as the number of machines in the simulation model increases. This indicates that the bilevel approach is better at finding optimal parameter settings for difficult problems. However, the experiments also show there are local optima in the parameter tuning problem, where the bilevel optimization can get stuck.

More experiments are needed to identify methods and their parameters that make the upper level optimization more robust.

The optimized parameter settings are better than the recommended settings at finding extreme solutions, as shown in Figure 3. One possible explanation for this can be the small parameters that make the upper level optimization more robust.
difficult when the solutions get close to the Pareto-optimal set, when smaller instead of larger changes are needed.

To study the effect of the function evaluation budget on the optimal parameter settings, budgets of 4000 and 8000 are used on the algorithm layer. The results show that the budget does not have a large effect on the optimal parameter settings. This indicates that the parameter settings that are good in the beginning of the search also work well in later stages of the search.

For evolutionary optimization algorithms there is a rule of thumb that the mutation probability should be set to \( \frac{1}{N} \), where \( N \) is the number of decision variables. This rule is evaluated by performing parameter tuning experiments on four different simulation models, with different number of decision variables. The results show that this heuristic is reasonably good at estimating the optimal mutation probability for the simulation-based problem used in this paper. It does, however, underestimate the mutation probability for both the 4000 and 8000 budgets. Further experiments are needed to verify whether the heuristic holds for other models with more decision variables and higher budgets.

REFERENCES


Paper V
On the Trade-off Between Runtime and Evaluation Efficiency In Evolutionary Algorithms

Martin Andersson  
School of Engineering Science, University of Skövde, Skövde, 54128, Sweden  
martin.andersson@his.se

Amos H.C. Ng  
School of Engineering Science, University of Skövde, Skövde, 54128, Sweden  
amos.ng@his.se

Jacob Bernedixen  
School of Engineering Science, University of Skövde, Skövde, 54128, Sweden  
jacob.bernedixen@his.se

Abstract  
Most evolutionary algorithms come with parameters that can be used to change the search strategy of the algorithm. One such parameter is the offspring population size. It controls the number of new individuals that are created in each generation. A higher offspring population size has the potential to reduce the runtime of an optimization, because more function evaluations can be performed in parallel. However, it will also affect the search strategy, possibly in a negative way. In this paper this trade-off is empirically investigated for NSGA-II and Differential Evolution (DE) on a subset of the DT LZ problems and a simulation-based problem. Parameter tuning is used to find the optimal parameter settings along this trade-off front. The results show that there is a trade-off between runtime and evaluation efficiency and that the impact on the efficiency of DE when the runtime is decreased is less compared to NSGA-II. This allows DE to achieve better results, both in terms of shorter runtime and better efficiency, for all the problems included in this paper. The interactions of the population size and offspring population size is also explored. The results show that there are indeed interactions between these parameters that depend both on the algorithm and problem.

Keywords  
evolutionary algorithms, NSGA-II, differential evolution, parameter tuning, offspring population.

1 Introduction  
The performance of an optimization algorithm is usually measured in terms of how many Function Evaluations (FEs) that are needed to reach a pre-determined performance target. Or alternatively, the quality of the solution(s) after a maximum number of FEs. Here this performance aspect is referred to as the function evaluation efficiency, or just the efficiency, of the optimization algorithm. Another performance aspect is the runtime of the optimization. The runtime and the number of FEs are proportional to each other when no parallelization is utilized, and when the computational overhead of the optimization is low compared to a function evaluation. However, the relationship between the runtime and efficiency of an optimization algorithm is more complex when parallel computing resources are utilized. The aim of this paper is to study this relationship for Evolutionary Algorithms (EAs).
The runtime of a population-based EA can be estimated as a product of how long each FE takes \((t_{FE})\), the size of the offspring population \((P)\), i.e., how many FEs in each generation as well as the number of generation \((N)\), if the FEs are run in serial in a single computing node:

\[
RT_s = t_{FE} \times P \times N
\]

In a parallel computing environment, if the number of available parallel computing nodes is set to match the size of one generation, then the runtime of the optimization is estimated to be only the product of how long a FE takes and the number of generation:

\[
RT_p = t_{FE} \times N
\]

In other words, given a fixed number of generation, the runtime of a population-based EA in a parallel computing environment can be reduced by a factor of \(P\). Furthermore, increasing the offspring population size (from \(P\) to \(P'\) where \(P' > P\)) will allow more FEs to be performed in parallel in each generation, it appears that the higher parallelization potential, i.e., more computing nodes running in parallel in each generation, the runtime \((RT_p)\) can be further reduced because the total number of generation (i.e. from \(N\) to \(N'\) where \(N' < N\)) can be cut. However, increasing the offspring population size could also have an negative impact on the efficiency. In other words, an algorithm that is configured to be optimal in terms of efficiency is not guaranteed to be optimal in terms of runtime. This gives rise to an unknown, complex relationship between total runtime and FE efficiency in a parallel computing environment because minimizing runtime by using more parallel runs in each generation does not guarantee the efficiency to be maximized. At some point, the efficiency of the optimization is actually compromised when the parallelization potential to reduce runtime \((RT_p)\) is increased.

A distinction should be made between the parallelization potential and the actual utilization of parallel resources in an algorithm. The parallelization potential represents the maximum amount of parallel computing resources an algorithm can exploit. When the available computing resources are greater than the parallelization potential some of the available resources will not be utilized. Conversely, when the available computing resources are less than the parallelization potential, the runtime of the optimization will be longer than its potential.

Another way of stating the conflict between these two performance metrics is that at some point the efficiency of the optimization is decreased when the parallelization potential is increased. The trade-off is therefore between a shorter runtime, at the cost of more FEs. Assuming a constant cost and time for each function evaluation and that computing resources are matched against the parallelization potential, the trade-off would be between cost and time.

The motivation for exploring this trade-off between cost and time is that for some computationally expensive applications, such as Discrete Event Simulation (DES) or Finite Element Method (FEM), it is not always enough to only use an efficient algorithm. Parallel resources must also be exploited in order to find good-enough solution(s) under, either soft or hard, time constraints. It is also now easier than ever to create hundreds or even thousands of computing resources on demand. It is however not well studied how EAs work at that scale. The aim of this paper is to close some of that knowledge gap for two EAs, NSGA-II and Differential Evolution (DE) applied on a subset of the DTLZ problems and a DES problem.

Both the efficiency and the parallelization potential can be controlled by the parameters of the optimization algorithm. The parallelization potential is directly controlled
by the offspring population size and the efficiency is determined by all parameters of
the EA. So by tuning the offspring population size together with the rest of the control
parameters the trade-off front between the runtime and efficiency can be obtained. The
approach taken in this paper is to formulate this as a multi-objective bilevel optimiza-
tion problem where the upper level objectives are to minimize both the number of FEs
and the number of generations to a pre-determined performance target. The decision
variables of the upper level optimization are the control parameters of the lower level
optimization.

From the non-dominated front of each upper level optimization experiment a sol-
solution is then picked so that the interactions between the population size and the off-
spring population size can be examined in greater detail. A sensitivity analysis is done
around the picked solution where the changes in number of FEs and number of genera-
tions is recorded for different combinations of population size and offspring population
size values.

This rest of the paper is structured as follows. First a background is given in Sec-
tion 2. Then related work is discussed in Section 3. The choice of tuning algorithm
and the extensions to it are given in Section 4. The experiment setup and experiment
results are presented in Section 5 and Section 6 respectively. Finally, the conclusions are
summarized in Section 7.

2 Background

This section gives an overview of EAs, multi-objective optimization problems and the
parameter setting problem.

2.1 Parallelization Potential of EAs

EA is a category of metaheuristics that are inspired by concepts from nature in general
and evolution by natural selection in particular. They mimic evolution by maintaining
a population of individuals that cross-breed and mutate to form new individuals. The
best performing individuals are given more opportunities to breed, creating a selec-
tion pressure that drives the population towards better solutions. EAs can be broadly
divided into two categories based on how they manage the population of individu-
als, generational and steady-state. A generational EA separates the generation of new
individuals and the updating of the population. That is, the population will only be up-
dated after all new individuals have been created and evaluated. In contrast, a steady
state EA will immediately update the population after an individual is evaluated.

The parallelization potential of an generational EA is determined by the offspring
population size. It controls how many new individuals that are created each in gen-
eration, meaning the maximum parallelism is capped by it. Increasing this number
will increase the parallelism but it will also affect the search behavior of the algorithm,
possibly in a negative way. The trade-off is therefore between increased parallelism
potential and the negative effects on the search behavior.

Another method for increasing the parallelization potential is to have multiple co-
operating algorithms that exchange candidate solutions among each other. The advan-
tage of this method is that it is possible to achieve synergy effects, but it is also more
complicated to implement and it is not trivial to know what algorithms to combine and
in what way.

The parallelization potential of a steady-state algorithm is controlled by how many
simultaneous individuals that are created. The advantaged is a high utilization of the
available resources since the algorithm never have to wait for any specific evaluation.
However, more simultaneous individuals will also increase the age of the information that is available to the optimization algorithm. The result of an optimization is also dependent on the order in which evaluations are completed. Which means that reruns of an optimization is not guaranteed to find the same solution(s) even when the same seed is used.

2.2 Parameter Setting Problem

The no free lunch theorem states that no algorithm can be better than all other algorithms on all problems (Wolpert and Macready, 1997). This, of course, also applies to the parameter setting problem. For example, a simple problem might benefit from greedy Control Parameter Values (CPVs), while the same CPVs applied to another problem can cause the algorithm to get stuck in a local optima. Great care must therefore be taken when selecting CPVs for any new problem. It is however difficult to predict which CPVs that will work well for any given problem. This issue can be mitigated by designing the CPVs to work well across a range of problems, yielding a generalist. In contrast, a specialist is a CPV tuple that is only tuned to work well for a small set of problems. The performance difference between a generalist and specialist on the same problem can be significant (Andersson et al., 2015; Smit and Eiben, 2010).

Solving the parameter setting problem can be classified into two approaches, parameter tuning and parameter control. Parameter tuning is when the CPVs are set at the beginning and kept constant throughout the optimization. With parameter control the CPVs are initialized at the beginning and are then adapted according to some rule or heuristic during the optimization. This distinction is however fuzzy since parameter tuning can also be used to set the initial values and also to tune the heuristics.

Parameter tuning can be further divided into three layers: The application layer, the algorithm (lower) layer and the design (upper) layer (Eiben and Smit, 2011). The problem that is solved is on the application layer and the optimization algorithm used to solve it is on the algorithm layer. On the design layer is the tuner that tests different CPV tuples for the optimization on the algorithm layer. The quality of solutions on the application layer is called fitness and the quality of the parameters is called utility (Eiben and Smit, 2011).

2.3 Multi-objective Optimization

This paper considers multi-objective optimization problems as described in Equation (1), where $M$ is the number of objectives, $\mathbf{x} \in \mathbb{R}^N$ is a vector of $N$ decision variables and $f$ represent objective functions which map a solution in the decision space to the objective space as $f_i : \mathbb{R}^N \rightarrow \mathbb{R}^M$.

Minimize $\{f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_M(\mathbf{x})\}$

Subject to $\mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u$ (1)

Comparing solutions in a single objective optimization is trivial since the value of the objective can be used directly in the comparison. However, when there are multiple conflicting objectives this is not possible. Instead, the concept of domination is used to compare solutions in a multi-objective problem. For a minimization problem a solution $u$ is said to dominate another solution $v$ if $u_i \leq v_i$ for all $i = 1, \ldots, M$ and $u_i < v_i$ for at least one $i = 1, \ldots, M$. When neither of two solutions dominate each other, they are said to be non-dominated. The result of a multi-objective optimization is a set of solutions called the non-dominating set and the set of all feasible non-dominated solutions is called the Pareto-optimal set. There are two main properties that are desired of
On the Trade-off Between Runtime and Evaluation Efficiency In Evolutionary Algorithms

a set of non-dominated solutions, convergence and spread. The convergence of a non-dominated set is a measure of how close it is to the Pareto-optimal set and the spread measures the diversity of the solutions within the set. In order to simplify the comparison of CPV tuples at the upper level optimization a unary performance indicator of the optimization at the algorithm layer is desired. Two such indicators are the inverted generational distance ($I_{IGD}$) (Zhang et al., 2009) and hypervolume ($I_H$) (Zitzler, 1999).

1. The inverted generational distance of a non-dominated solution set $S$ obtained from a Multi-Objective Evolutionary Algorithm (MOEA) is defined as,

\[
I_{IGD}(S, P) = \frac{1}{|P|} \sum_{x^* \in P} \min_{x \in S} \|F(x) - F(x^*)\|, \tag{2}
\]

where, $P$ is a reference set of Pareto-optimal solutions that represent the desired density and distribution of points on the Pareto front, $\|\cdot\|$ denotes the Euclidean distance, and $|\cdot|$ denotes the size of a set. Solution sets with lower $I_{IGD}$ values are preferred since they closely resemble the reference set $P$. Thus, the best possible value for $I_{IGD}$ is 0.

2. The hypervolume of a non-dominated solution set $S$ obtained from an MOEA is defined as,

\[
I_H(S, r) = \lambda \left( \bigcup_{x \in S} H(F(x), r) \right). \tag{3}
\]

where, $r$ is a reference point that is dominated by all solutions in $S$, $\lambda(\cdot)$ is the Lebesgue measure, and $H(a, b)$ represents the hypercube with body-diagonal $ab$. Solution sets with higher $I_H$ values are better because they dominate a larger region of the objective space. Hypervolume is the only known unary quality indicator that is Pareto-compliant (Zitzler et al., 2007), i.e., for any two solution sets $S_1$ and $S_2$ ($\neq S_1$), $S_1 \preceq S_2 \Rightarrow I_H(S_1) > I_H(S_2)$.

The main advantage of the hypervolume metric is that it does not require the knowledge of the Pareto-optimal set. The drawback is that it is sensitive to the inclusion or exclusion of extremal points and the relative ranges of the objectives. It is also relatively slow for large number of objectives. The main advantage of the IGD metric is that is less computationally demanding than calculating the hypervolume, but it requires knowledge about the Pareto-optimal set and it is also not Pareto-compliant.

3 Related Work

This section presents related work that concerns parameter tuning approaches and the affect of the offspring population size on the search behavior of EAs.

3.1 Parameter Tuning Approaches

There are many different parameter tuning methods, see Eiben and Smit (2011) for an overview. The approach taken in this paper is to formulate the parameter setting problem itself as an optimization problem and then solve it using an optimization algorithm. This is referred to as a Meta-EA (Eiben and Smit, 2011) or bilevel optimization (Sinha et al., 2014). One of the earlier uses of this method was by Grefenstette (1986). However, the computational complexity and the less powerful hardware limited the scope and usefulness of such early studies.
Other approaches include enumeration methods, such as Latin hypercube sampling or factorial analysis. Myers and Hancock (2001) used a graeco-latin square design with a wide range for each parameter setting. This was followed by fully crossed factorial design with narrower ranges. This allowed them to find interactions between the algorithm performance and the control parameters (specifically population size). Although the exponential nature of the factorial design limited them in how many interactions they could model.

A sensitivity analysis of the control parameters of a parallel asynchronous cellular genetic algorithm was performed in (Pinel et al., 2012). They varied the control parameters in discrete steps in order to find the most important factors. This allowed them to explore the relationship between the algorithms influential parameters and the problem instances.

There are also specialized single-objective tuning algorithms, such as irace (López-Ibáñez et al., 2016) and REVAC (Nannen and Eiben, 2007). Ugolotti and Cagnoni (2014) used NSGA-II to solve a multi-objective parameter tuning problem. Among the performance metrics they considered was the number of FEs and the number of generations to a performance target. They found that there exist a trade-off between these performance metrics and that the population size is the parameter that influences the position on the non-dominated front the most.

### 3.2 Population Size and Offspring Population Size

Theoretical analysis on the effects of both the population and offspring population sizes in EAs exist. However, they are limited to both simple EAs and problems. It is also not clear how relevant these findings are for more complex EAs and real-world problems. Even so, these analysis could still provide valuable insights.

Chen et al. (2012) did a performance analysis of a \((1 + \lambda)EA\) for different values of \(\lambda\) (offspring population size). The aim was to find the largest value for the offspring population size so that the parallel computation capabilities was maximized without significantly increasing the overall computational cost. They argue for the offspring population size to be roughly the reciprocal of the probability of producing an improved offspring. For problems with simple landscapes they found moderate values of \(\lambda\) to be optimal, with only a small penalty for modest increases in \(\lambda\). For more complex landscapes they found that larger offspring population sizes were needed in order to have a higher probability of finding the global best solution.

Jansen et al. (2005) investigated the performance of an \((\mu + \mu)EA\) and found that an increase in population size lead to a significant drop in the solvable rate. They also found that when the problem has attraction in a basin leading to a local optimum, and the individuals at that basin have relatively high fitness, a large population may be harmful since it will lead to a large probability of creating new solutions at that basin. Their conclusion was therefore that a high population size can sometimes be harmful to the performance of the optimization algorithm.

### 4 Tuning Algorithm

The aim of this paper is to explore the trade-off between the number of generations and the number of FEs to a performance target in EAs. The approach is to use parameter tuning to find the non-dominated CPV tuples that minimizes both these objectives. This two-objective optimization problem is then solved using an EA. This section describes how the parameter setting problem is formulated as an optimization problem and the EA, and its extensions, that is used to solve it.
4.1 Measuring Efficiency

There are two ways of measuring the efficiency of a CPV tuple when it is applied to an EA at the algorithm layer. The first method is to measure how many FEs that are needed by the EA to reach a pre-determined performance target. The second method is to calculate the performance after a maximum number of FEs have been reached. Both approaches have their advantages and disadvantages and both require a pre-determined value, either the maximum budget or the performance target. A maximum budget value should preferably be high enough so that the EA can get a good diversity and convergence of the solutions. However, if it is too high and most CPV tuples can get close the Pareto-front it can be difficult to differentiate between them. Therefore, the method used in this study is to measure the efficiency as the number of evaluations to a performance target. However, a maximum budget is still needed since some CPV tuples may never reach the performance target. When a CPV tuple fail to reach the performance target within the maximum budget, a constraint violation equal to the performance target minus the actual performance will be assigned to it. That way, the tuner can still differentiate between CPV tuples that fail to reach the target.

4.2 Measuring Time

With the assumption that an entire generation can be run in parallel the runtime of an EA is the number of generations to the target performance. This assumption is of course an simplification and may not hold under all circumstances, especially for stochastic problems with lots of variability. However, it should be accurate enough for the purpose of this study. Running the entire generation in parallel can be acheived by exactly matching the offspring population size to the available parallel computing resources. The offspring population size is also arguably the parameter with the largest influence on the runtime since it is directly proportional to the size of a generation. However, there could also be complex interactions with the other parameters that might affect how many generations that is needed to reach the target.

4.3 Problem Formulation and Tuner Algorithm

With the above descriptions for efficiency and runtime of an optimization the optimization problem is formulated as follows.

\[
\begin{align*}
\text{Minimize} & \quad (FE(p), \mathcal{G}(p)) \\
\text{where,} & \quad FE(p), \mathcal{G}(p) \text{ is the mean number of function evaluations and generations needed to reach a target hypervolume value when the following problem is solved with parameters } p \\
\text{Minimize} & \quad \{F(x) = f_1(x), f_2(x), \ldots, f_M(x)\} \\
\text{Subject to} & \quad x_l \leq x \leq x_u 
\end{align*}
\]

(4)

The algorithmic parameters of the lower level optimization problem become the variables for the upper level optimization problem.

The optimization problem as defined above have at least the following properties that make it difficult. It is noisy since the lower level optimizations are stochastic, it has interacting variables such as mutation probability and mutation strength and it has non-linear objective functions with local optima. NSGA-II (Deb et al., 2002a) is selected as the tuner, since EAs have been shown to be very competitive on problems with these characteristics (Eiben and Smit, 2011). NSGA-II is extended with a multiple stage ex-
tension that is designed to overcome two problems, noisy optimization algorithms and local optima. The stages are configured with different CPVs for an explicit exploration and exploitation search strategy (Andersson et al., 2016).

### 4.4 Noise handling

EAs are stochastic algorithms that can produce very different results from one run to another. There are numerous ways of dealing with this noise. The most straightforward way is to sample each CPV tuple multiple times. Sampling strategies can be divided into two categories, static sampling and dynamic sampling. With static sampling the same number of samples is used throughout the entire optimization. In contrast, a dynamic sampling strategy modifies the number of samples based on some heuristic or rule as the optimization progresses. There is also already some inherit noise handling in population based algorithms, because new individuals are sampled from a local area. This is referred to as implicit averring (Jin and Branke, 2005).

The sampling strategy extension used in this study is a simple dynamic sampling scheme that increases the number of samples in a three discrete steps during the optimization. The assumption is that in the beginning of the optimization it not necessary to accurately asses the utility of a CPV tuple, but instead rely on the implicit averring. That effort is instead better spent on testing more unique CPV tuples. It is first later in the optimization as the final non-dominated front is starting to take form, that the number of samples should be increased. Only the CPV tuples with the highest number of samples are considered for the results.

The experiments in this study use the estimated mean as the utility for both $F(p)$ and $G(p)$ for a CPV tuple with parameters $p$. Common Random Numbers (CRN) are used within the same design layer experiment, so that two CPV tuples with the same population size will always generate the same initial population. This reduces the impact (which can be substantial) of the starting conditions, thereby making it easier to determine if any difference in the utility is due to a CPV change or by random chance. The drawback is that the parameters will be tuned to a specific set of seeds. Hence, the final number of replications must be high enough so that the tuned parameters are not overfitted with respect to the selected seeds.

### 4.5 Parameter Settings for the Tuner

Since NSGA-II is used as the tuner, the parameter setting problem also appear on the design layer. And as previously discussed, the parameter setting problem, on the algorithm layer, is both difficult and computationally expensive. So the performance of the tuner is important both in terms of finding the global best CPV tuples and doing so in a reasonable amount of time. The extension to NSGA-II is designed to handle these two issues. It allow NSGA-II to change both the CPVs, of the tuner, and the number of replications of a CPV tuple during the optimization. Changing the CPVs is intended to minimize the risk of converging to a local optima by searching with exploratory CPVs early in the optimization. Later in the optimization the CPVs are switched to ones that are more exploitative. Considerable effort can also be saved by using fewer replications of an CPV tuple in the exploratory stage.

Three stages are used for all experiments in this paper. The CPVs for each stage is shown in Figure 1.
Table 1: Experiment settings and configuration for the different stages.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Offspring population size</td>
<td>100</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>3</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>Mutation Distribution Index</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>Crossover Distribution Index</td>
<td>10</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Tournament size</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

5 Experiment Setup

This section will provide details about the empirical experiments that are performed in order to gather data about the optimal CPV tuples with respect to the trade-off between number of FEs (cost) and generations (time). One CPV tuple from the knee region of each experiment will be selected to provide the values for the parameters that will be kept static during the sensitivity analysis. This allows for a sensitivity analysis of the population size and offspring population size around the CPV tuple that provides the best trade-off, for that particular EA and problem combination.

5.1 Algorithms Tuned

Two evolutionary algorithms are tuned in this study, NSGA-II and DE. Following is a brief description of them and the control parameters that are tuned.

5.1.1 NSGA-II

NSGA-II (Deb et al., 2002a) is an evolutionary multi-objective optimization algorithm that mimics the process of natural evolution. It starts with a random population of individuals that are evolved over time by crossing over and mutating individuals. In each generation a fixed number of new individuals are added to the population. At the end of an generation the population is trimmed by preferring non-dominated individuals and as a second criteria the least crowded individuals. The crossover operator is SBX and the mutation is polynomial.

1. Population size ($\mu$): The number of individuals in the population. An integer in the range [10, 300].
2. Offspring population size ($\lambda$): The number of children created each generation. An integer in the range [10, 5000].
3. Tournament size ($ts$): The number of tournament rounds. A value of 1 means that parents are selected randomly from the population. An integer in the range [1, 20].
4. Mutation probability ($p_m$): The probability of random changes to the decision variables as $p_m / N$, where $N$ is the number if decision variables. A real-value in the range [0, 5].
5. Mutation Distribution Index ($\eta_m$): Index governing the proximity of the mutated child to its parent. Larger values mean a smaller change to the original value. A real-value in the range $[0, 500]$.

6. Crossover probability ($p_c$): The probability of creating offspring from two parents. A real-value in the range $[0, 1]$.

7. Crossover Distribution Index ($\eta_c$): Index governing the proximity of the children to their parents. Larger values mean the children are closer to their parents. A real-value in the range $[0, 500]$.

5.1.2 Differential Evolution

DE (Storn and Price, 1997) is population based evolutionary algorithm. The original version is single-objective, but the version implemented here is extended to be able to handle multi-objective problems as well. It starts by filling the population with a random set of individuals. A new individual is then created by adding the weighted difference between two individuals in the population to a third individual. This new individual is then mixed with a fourth individual and is then added to the population. All parents are selected randomly from the population. When all offspring have been created and evaluated the population is trimmed using the same approach as in NSGA-II. That is, non-dominated individuals are preferred over dominated ones and crowding distance is used as a secondary criteria. The mutation operator is polynomial.

1. Population size ($\mu$): The number of individuals in the population. An integer in the range $[10, 300]$.

2. Offspring population size ($\lambda$): The number of children created each generation. An integer in the range $[10, 5000]$.

3. Differential weight ($F$): Controls the amplification of the differential variation. A real-value in the range $[0, 5]$.

4. Crossover probability ($p_c$): The probability of accepting the differential perturbation for each decision variable. A real-value in the range $[0, 1]$.

5. Mutation probability ($p_m$): The probability of random changes to the decision variables as $p_m / N$, where $N$ is the number if decision variables. A real-value in the range $[0, 5]$.

6. Mutation Distribution Index ($\eta_m$): Index governing the proximity of the mutated child to its parent. Larger values mean a smaller change to the original value. A real-value in the range $[0, 500]$.

5.2 A Scalable Simulation Model

The simple stochastic simulation model considered in the experiments of this paper represents an un-paced flow line, consisting of $s$ workstations with $s - 1$ inter-station buffers. The terms ‘workstation’ and ‘machine’ are interchangeable in the following discussions, since there is only one machine in each workstation. The productivity of each machine $i$ is governed by its availability ($\alpha_i$), processing time ($\beta_i$) and repair time ($\gamma_i$). In the initial state, the workloads of all the workstations are perfectly balanced, each having a processing time of $\beta^{\text{orig}} = 80$ seconds per job. All machines have an availability of $\alpha^{\text{orig}} = 90\%$ and repair time of $\gamma^{\text{orig}} = 300$ seconds. The processing times
are assumed to be constant, which is realistic for automated machining processes. The times to failure of the workstations are modeled with exponential distributions and the randomness of the repair times $\gamma_i$ is modeled using Erlang distributions.

In a complex flow line with unbalanced workloads, the detection of bottlenecks is essential for a subsequent improvement of the production rate or throughput. The location of bottlenecks depends on many factors, including the job flow logic, variability and disturbance of the machines and the buffer allocations. Even for a simple, straight flow line with balanced workloads as described above, detecting which workstation(s) to improve in order to increase the overall throughput of the line to a certain level is not a trivial task. The concept of treating this throughput improvement problem as a multi-objective optimization problem of identifying the optimal (minimal) number of changes to maximize the throughput was first proposed in (Pehrsson, 2013) and later further elaborated in (Ng et al., 2014). In such an optimization formulation, the system throughput ($TH$) is the primary objective for improvement, so that $f_1(x) = \max \{TH(x)\}$. The total number of changes, i.e. improvement actions, can be defined as the secondary objective function, $f_2(x)$. There are three integer improvement variables $\{\alpha_i, \beta_i, \gamma_i\}$. The available improvement actions for availability, processing times and repair times and their corresponding step-sizes are:

$$\begin{align*}
\alpha &= \{80, 96\} & \Delta \alpha &= 1 \\
\beta &= \{60, 80\} & \Delta \beta &= 1 \\
\gamma &= \{180, 360\} & \Delta \gamma &= 1
\end{align*}$$

The second objective, $f_2(x)$, can then be written as a summation of improvements (Ng et al., 2014):

$$f_2(x) = \min \left\{ \sum_{i=1}^{s} \hat{\alpha}_i + \sum_{i=1}^{s} \hat{\beta}_i + \sum_{i=1}^{s} \hat{\gamma}_i \right\}$$

where

$$\begin{align*}
\hat{\alpha}_i &= \frac{\alpha_i - \alpha_{orig}}{\Delta \alpha} \\
\hat{\beta}_i &= \frac{\beta_i - \beta_{orig}}{\Delta \beta} \\
\hat{\gamma}_i &= \frac{\gamma_i - \gamma_{orig}}{\Delta \gamma}
\end{align*}$$

Additionally, in order to simultaneously solve the lean buffer problem (Enginarlar et al., 2005), the capacity of inter-station buffer spaces are also optimized, $B_i = \{1, 2, \ldots, 10\} \forall i \in \{1, \ldots, s - 1\}$, by adding a third objective of minimizing the total number of buffers, i.e. $f_3(x) = \min \{\sum_{i=1}^{s-1} B_i\}$.

The model described in this section is developed and simulated using Facts Analyzer 2.0 (Ng et al., 2011). It will be referred to in the rest of the paper as the SIM problem.

5.3 Tuning Problems

NSGA-II and DE are tuned against DTLZ1 and DTLZ2 from the DTLZ test suite (Deb et al., 2002b) and the SIM problem as described above. Each problem is configured for 3 objectives with two instances that differ only in the number of decision variables. The experiment settings and the configuration for each problem is presented in Table 2. The motivation for varying the number of decision variables is to detect any differences in the optimal CPV tuples for the same problem but with different difficulty. The reason the reference point and hypervolume target is different between the two SIM problems is that only one of the objectives is simulated. The other two objectives are also decision variables that change when additional machines are added. The hypervolume targets
Table 2: Experiment settings and configuration for the included problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Hypervolume</th>
<th>Target</th>
<th>Reference point</th>
<th>Decision variables</th>
<th>Maximum budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTLZ1</td>
<td>1330.97</td>
<td>11, 11, 11</td>
<td>7</td>
<td>50000</td>
<td></td>
</tr>
<tr>
<td>DTLZ1</td>
<td>1330.97</td>
<td>11, 11, 11</td>
<td>14</td>
<td>60000</td>
<td></td>
</tr>
<tr>
<td>DTLZ2</td>
<td>1330.45</td>
<td>11, 11, 11</td>
<td>12</td>
<td>70000</td>
<td></td>
</tr>
<tr>
<td>DTLZ2</td>
<td>1330.45</td>
<td>11, 11, 11</td>
<td>24</td>
<td>110000</td>
<td></td>
</tr>
<tr>
<td>SIM</td>
<td>4488000</td>
<td>20, 648, 200</td>
<td>11</td>
<td>30000</td>
<td></td>
</tr>
<tr>
<td>SIM</td>
<td>13278000</td>
<td>20, 1000, 400</td>
<td>19</td>
<td>40000</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: The number of replications and budget for each stage.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Replications</th>
<th>Budget</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>DTLZ1-7</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>DTLZ1-14</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>DTLZ2-12</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>DTLZ2-24</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>SIM</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>ALL</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

6 Experiment Results

The results from the experiments will be presented in three parts. The first part is all the trade-off fronts between the number of FEs and generations. The second part is all non-dominated solutions from each experiment visualized in parallel coordinates.
plots. The final part is a sensitivity analysis for the parameters population size and offspring population size.

6.1 Generations and Evaluations Trade-off Front

All non-dominated solutions from all experiments are presented in Figure 1. Each sub-figure contains the front for both NSGA-II and DE on the respective problems. This allows for a performance comparison between NSGA-II and DE with the best CPV tuples found for each algorithm. For each problem the non-dominated solutions from the DE experiments all dominate the non-dominated solutions from the NSGA-II experiments. So it is clear that DE is superior both as an specialist and as a generalist, for the problems included in this study.

The trade-off front for DE on DTLZ1 is very steep, for both 7 and 14 variables. For this particular problem DE performs best when the offspring population size is very high, see Figure 5. The range for the number of generations is therefore only between 4 and 9. Since DTLZ1 contains many local fronts, one possible explanation is that a large number of offspring are needed to minimize the chances that the population converges on a local front. With many offspring it is more likely that at least some find their way past the current front. More experiments on other problems with and without local front are required to investigate this phenomenon further.

The results also show that there are clear knee regions on all problems for both NSGA-II and DE. Outside this region small gains in one objective leads to large sacrifices in the other. The NSGA-II knee regions are both larger and more round compared to DE. The efficiency of DE does not seem to be significantly affected by the offspring population size. At least until the point where only a handful of generations are needed to reach the performance target. After that point the efficiency drops off significantly.

6.2 Parallel Coordinates Plot of Non-dominated CPVs

Parallel coordinates plots of all non-dominated CPV tuples for all experiments are presented here. Each plot show how the values change for each parameter across the non-dominated front of CPV tuples, if at all. It also highlights possible interactions and correlations between the parameters, correlations are measured using the Pearson correlation coefficient. It is clear from these figures that there is no CPV tuple that is optimal across all problems.

The reason that not all algorithm and problem combinations utilize the entire range of the offspring population size parameter (5000). Is that only CPV tuples with no constraint violation is included in the results.

Figure 2 show the optimal CPV tuples for NSGA-II on DTLZ1. There are 128 and 69 CPV tuples for DTLZ1 with 7 and 14 variables. The parameters $p_m$, $n$, $\eta_c$, $\eta_m$ and $p_c$ have both small ranges and are similar for 7 and 14 variables. Indicating that their optimal values are not affected by either the position on the trade-off front nor the number of variables. As expected, it is clear that the offspring population size controls to an large extent the trade-off between number of FEs and generations. There is also a correlation ($\rho_7 = 0.89$ and $\rho_{14} = 0.73$) between the offspring population size and the population size. In that the optimal population size also increases as the offspring population size increases, but not proportionally or strict.

In Figure 3 the optimal CPV tuples for NSGA-II on DTLZ2 are shown. There are 204 and 201 CPV tuples for 12 and 24 variables. Compared to DTLZ1 these results are more complex. There are two distinct groups of CPV tuples that split the non-dominated front in two parts. Each side of this point have very different optimal CPV
tuples. The first group have a higher \( p_m \) and a \( t_s \) of one, much higher \( \eta_c \) and low \( \eta_m \). They have the same, high, \( p_c \) though. The first group also have both higher population sizes and offspring population sizes. The reason for these two distinct groups of CPV tuples is hard to determine without further experiments. The same correlation (\( p_{12} = 0.96 \) and \( p_{24} = 0.96 \)) as in DTLZ1 between the population size and offspring population size are also present in DTLZ2.

The non-dominated CPV tuples for NSGA-II on SIM are presented in Figure 4. There are 56 and 48 CPVs for SIM with 11 and 19 variables. One difference from the DTLZ1 and DTLZ2 problems is that the optimal CPVs favors relatively more mutation and less crossover. The mutation rate is higher than the normally recommended value of 1 and the crossover rate, which was close to one for DTLZ1 and DTLZ2, is concentrated around 0.6 to 0.8. The parameters are similar for both problem instances, except for \( t_s \) and \( \eta_c \). Which is higher and lower respectively for the 11 variable problem. The correlation between the population size and offspring population size is weaker though (\( p_{11} = 0.27 \) and \( p_{19} = 0.62 \)) compared to DTLZ1 and DTLZ2.

Figure 5 show the non-dominated CPV tuples for DE on DTLZ1. There are 33 and 24 CPV tuples for 7 and 14 variables. The optimal CPV tuples are similar for both 7 and 14 variables. The most significant difference is in the range for the optimal offspring population size. For 7 variables the optimal range starts at around 800 and for 14 variables it starts around 2000. This is considerably higher than for the other problems. The
On the Trade-off Between Runtime and Evaluation Efficiency In Evolutionary Algorithms

Figure 2: Parallel coordinate plot over the objectives and parameter settings of all non-dominated CPV tuples for NSGA-II on DTLZ1 with 7 and 14 decision variables.

The non-dominated CPV tuples for DE on DTLZ2 are shown in Figure 6. There are 220 and 207 CPV tuples for DTLZ2 with 12 and 24 variables. The optimal CPV tuples are similar for both the 12 and 24 variable problem, except for $p_m$ that is shifted upwards for the 24 variable problem. The range for the parameters $F$, population size and $p_c$ are small. While $p_m$ vary over almost the entire range and $\eta_m$ varies from around 200 up to 500. Both $p_m$ and $p_c$ is higher in the optimal CPV tuples in DTLZ2 compared to DTLZ1. However, only small mutations are performed since $\eta_m$ is also high. There is no linear correlation between $p_m$ or $\eta_m$ and the other parameters. It could be that they do not have a significant impact on the performance metrics, or that they have complex interactions with one or more of the other parameters.

The non-dominated CPV tuples for DE on SIM are presented in Figure 7. There are 39 and 46 CPV tuples for SIM with 11 and 19 variables. There are two main differences compared to the DTLZ results. First, the range for $F$ is larger and shifted upwards. The population size is also larger, around 50 to 120. The 19 variable problem also have higher values for $p_m$ and a group of CPV tuples with low $p_c$.

The non-dominated generalist CPV tuples for both DE and NSGA-II on all problems are shown in Figure 8. There are 51 and 32 CPVs for ALL on NSGA-II and DE.
respectively. The NSGA-II parameters $p_m$ and $\eta_m$ is close but a little bit lower than their recommended defaults. That is, fewer but larger mutations are done compared to the recommended settings. The parameter $p_c$ is around 0.9 and $\eta_c$ is around 100, so that offspring are almost always crossed over, but the offspring are created relatively close to their parents. There is also a correlation between population size and offspring population size similar the specialists ($\rho = 0.80$).

For DE the parameter $F$ is 0.5 which is an often recommended value. The population size is kept low (20 to 30, except when the offspring population size is also very low) regardless of the offspring population size. A moderate amount of mutation is used but with a low $\eta_m$. The parameter $p_c$ ranges from just above 0 to around 0.2, which is just slightly lower than the often recommended value of 0.3.

### 6.3 Population Size and Offspring Population Size Sensitivity Analysis

The results from the population size and offspring population size sensitivity analysis will be presented here. These results will help to understand the correlation and interactions between these two parameters, and their effect on the objectives. The generation analysis plots are shown in an log scale because of the large differences between the results. It worth noting that the CPVs that are the starting point for this sensitivity analysis are taken from the knee region from the respective trade-off experiment. So the parameters that are not part of the analysis might have different values for each of the figures in this section. The large circle show where the evaluation minimum and the large square show where the generation minimum is for each value of offspring
Evolutionary Computation Volume x, Number x

Figure 4: Parallel coordinate plot over the objectives and parameter settings of all non-dominated CPV tuples for NSGA-II on SIM with 11 and 19 decision variables.

Figure 9 show the results for NSGA-II on the DTLZ1 problem. The correlation between population size and the offspring population size as mentioned earlier is also visible here. Increasing the offspring population size also shifts the optimal population size towards higher values. The optimal population size is also higher for the more difficult 14 variable problem and the difference between the optimal population size for the minimum and maximum offspring population size is also larger. From the results it is also clear that there is a larger penalty of selecting a too small population size compared to selecting a too large value. The optimal population sizes are similar but not exactly equal for both objectives.

In Figure 10 the results for NSGA-II on DTLZ2 are presented. Here the correlation between optimal population size and offspring population size is even more evident than on DTLZ1. The sensitivity of the population size also differs between different offspring population sizes. For example, on DTLZ2 with 12 variables an offspring population size of 10 has an optimal range of around 20 to 50. Whereas the performance for an offspring population size of 1000 does not change much between 80 and 300. Another difference between the 12 and 24 variable problems, is that the performance drop off is less pronounced on the harder problem.

The result for NSGA-II on SIM are shown in Figure 11. The offspring population size and population size correlation is less pronounced on SIM compared to DTLZ1 and DTLZ2. The SIM problem is also sensitive to setting a too low population size, but insensitive too setting it too high.
In Figure 12 the result for DE on DTLZ1 are shown. This is the problem where the optimal range for offspring population size was high. So the CPVs are optimized for a high offspring population size, which is probably why the population size affects an offspring population size of 10 differently than for other values of offspring population size. There is also a correlation between the optimal population size and the offspring population size. However, the correlation is inverted compared to NSGA-II, since a larger offspring population size has a smaller optimal population size. It is also sensitive, in terms of number of evaluations, to the appropriate population size, especially for the harder 14 variable problem.

Figure 13 show the results for DE on DTLZ2. The landscape is completely different from DTLZ1. Here the optimal population size is small, around 20 to 30, and there is no correlation between the optimal population size and the offspring population size, it stays small regardless of the value of the offspring population size. The performance penalty is quite large for choosing a too high population size.

In Figure 14 the results for DE on SIM are shown. The result are more similar to DTLZ1 than DTLZ2. It shows a high sensitivity for a too small population size and low sensitivity for a too high value. There is also no discernible correlation between the optimal population sizes for different values of offspring population size.

Figure 15 show the results for NSGA-II and DE on all problems. For NSGA-II there is again a correlation between the optimal population size and the offspring population size. In that a higher offspring population size also has an higher optimal population size. There is no such correlation for DE. The optimal population size is instead small
On the Trade-off Between Runtime and Evaluation Efficiency In Evolutionary Algorithms

Figure 6: Parallel coordinate plot over the objectives and parameter settings of all non-dominated CPV tuples for DE on DTLZ2 with 12 and 24 decision variables.

for all values of offspring population size, except when the offspring population size is also small. The difference, both in terms of evaluations and generations, is however small for that configuration. The landscape for NSGA-II and DE are quite different. In that NSGA-II is sensitive to a too small population size and DE is instead sensitive to a too large population size.

7 Conclusion

In this paper the trade-off between the runtime and the efficiency of DE and NSGA-II has been explored. The runtime is measured in how many generations that are needed to reach a performance target. The efficiency, or cost, of the optimization is measured in how many FEs that are needed to reach the same target. Both the runtime and the efficiency can be controlled by the optimization algorithm parameters. The approach taken in this paper is to formulate this as a two-objective parameter tuning problem, where the parameters of NSGA-II and DE is tuned when applied to DTLZ1, DTLZ2 and a DES problem. Each problem has two variants with different number of decision variables.

The performance of DE is clearly superior to NSGA-II for the included problems. Both when specifically tuned for each problem instance and when tuned as a generalist for all problem instances. The efficiency for DE does not decrease as rapidly when the offspring population size is increased as it does for NSGA-II. It maintains good efficiency even for high offspring population sizes. That makes DE a better choice when low runtimes are desired. But DE is also superior to NSGA-II in terms of efficiency.
The number of decision variables had only a minor affect on the non-dominated CPV tuples. Indicating that the number of variables is not as important as knowing the problem type when configuring an optimization algorithm. There were a few exceptions, such as the optimal population size for NSGA-II was, for some problems, increased for the harder problem.

A sensitivity analysis was also done for the population size and offspring population size parameters. For NSGA-II the optimal population size changes with the offspring population size. In general, the optimal population size is increased when the offspring population size is increased. However, a heuristic is difficult to construct since the relationship changes between problems and even within the same problem with different number of decision variables. For DE there is no such relationship. Instead, the optimal population size is small regardless of the offspring population size. DE is also more sensitive to setting the population size too high, rather than too low. For NSGA-II the opposite is true.
Figure 8: Parallel coordinate plot over the objectives and parameter settings of all non-dominated CPV tuples for NSGA-II and DE on the ALL problem.
Figure 9: Sensitivity analysis for parameters population size and offspring population size for NSGA-II. The large circle shows where the evaluation minimum is. The large square shows where the generation minimum is.
Figure 10: Sensitivity analysis for parameters population size and offspring population size for NSGA-II. The large circle shows where the evaluation minimum is. The large square shows where the generation minimum is.
Figure 11: Sensitivity analysis for parameters population size and offspring population size for NSGA-II. The large circle shows where the evaluation minimum is. The large square shows where the generation minimum is.
Figure 12: Sensitivity analysis for parameters population size and offspring population size for DE. The large circle shows where the evaluation minimum is. The large square shows where the generation minimum is.
(a) Evaluation analysis on DTLZ2 - 12

(b) Generation analysis on DTLZ2 - 12

(c) Evaluation analysis on DTLZ2 - 24

(d) Generation analysis on DTLZ2 - 24

Figure 13: Sensitivity analysis for parameters population size and offspring population size for DE. The large circle shows where the evaluation minimum is. The large square shows where the generation minimum is.
On the Trade-off Between Runtime and Evaluation Efficiency in Evolutionary Algorithms

(a) Evaluation analysis on SIM - 11

(b) Generation analysis on SIM - 19

(c) Evaluation analysis on SIM - 19

(d) Generation analysis on SIM - 19

Figure 14: Sensitivity analysis for parameters population size and offspring population size for DE. The large circle shows where the evaluation minimum is. The large square shows where the generation minimum is.
Figure 15: Sensitivity analysis for parameters population size and offspring population size for NSGA-II and DE. The large circle shows where the evaluation minimum is. The large square shows where the generation minimum is.
On the Trade-off Between Runtime and Evaluation Efficiency In Evolutionary Algorithms

References


Parameter Tuning Evolutionary Algorithms for Runtime versus Cost Trade-off in a Cloud Computing Environment

M. Andersson, A.H.C. Ng
School of Engineering Science, University of Skövde, 54128, Sweden

Abstract

The runtime of an evolutionary algorithm can be reduced by increasing the number of parallel evaluations. However, increasing the number of parallel evaluations can also result in wasted computational effort since there is a greater probability of creating solutions that do not contribute to convergence towards the global optimum. A trade-off, therefore, arises between the runtime and computational effort for different levels of parallelization of an evolutionary algorithm. When the computational effort is translated into cost, the trade-off can be restated as runtime versus cost. This trade-off is particularly relevant for cloud computing environments where the computing resources can be exactly matched to the level of parallelization of the algorithm, and the cost is proportional to the runtime and how many instances that are used. This paper empirically investigates this trade-off for two different evolutionary algorithms, NSGA-II and differential evolution (DE) when applied to multi-objective discrete-event simulation-based (DES) problem. Both generational and steady-state asynchronous versions of both algorithms are included. The approach is to perform parameter tuning on a simplified version of the DES model. A subset of the best configurations from each tuning experiment is then evaluated on a cloud computing platform. The results indicate that, for the included DES problem, the steady-state asynchronous version of each algorithm provides a better runtime versus cost trade-off than the generational versions and that DE outperforms NSGA-II.

Keywords: evolutionary algorithms, NSGA-II, differential evolution, discrete-event simulation

1. Introduction

When the stopping criteria for a black-box optimization algorithm is a particular performance target, the runtime of the algorithm is simply the time it takes to achieve that target. Under those circumstances, the runtime of the algorithm is strongly related to the number of solutions that it can evaluate in parallel. For evolutionary algorithms, the parallelism is directly controlled by
the offspring population size. Increasing the offspring population size will therefore, in general, reduce the runtime of the algorithm. However, increasing the offspring population size will also alter the search strategy and in turn can cause the algorithm to become less efficient in terms of the number of function evaluations it requires. The reason the algorithm can become less efficient is that the optimal value of the offspring population size is associated with the probability of creating solutions that contribute to the convergence towards the global optimum [1]. When the offspring population size is larger than the optimal value, the probability of creating such solutions increases and this can result in wasted computational effort. A trade-off, therefore, arises between the runtime and the function evaluation efficiency for different offspring population sizes of an optimization algorithm. Increasing the offspring population size will reduce the runtime, but at the same time increase the number of function evaluations that are needed. However, this relationship is probably not linear as there will be diminishing returns of increasing the offspring population size beyond a certain point.

Recent advances in cloud computing services have made cheap computing resources readily available for anyone to utilize. Resources can be created on demand and destroyed when no longer required. This reduces the need to own large clusters of physical computers to solve computational problems quickly. The trade-off between runtime and cost is particularly relevant in this context because the computing resources can be exactly matched to the offspring population size and assuming a per second billing by the provider of cloud computing services, the cost is simply:

\[
\text{cost} = r \times i_n \times i_c
\]

where \( r \) is runtime of the optimization, \( i_n \) is the number of computing instances and \( i_c \) is the cost of running one instance for one second. It is also assumed here that all instances are created before the optimization is started, since it is impractical to create and destroy computing instances instantaneously while the optimization is running. This introduces the utilization of the computing instances as another factor that needs to be considered, since the cost of running an instance is not related to its utilization, but to the amount of time it is kept alive (running). Given the same amount of time, an algorithm with a higher utilization will, therefore, be able to do more function evaluations at the same cost as an algorithm with a lower utilization.

Steady-state algorithms are a class of algorithms that differs from generational algorithms in the sense that they do not handle multiple solutions at the same time. Instead, solutions are created, evaluated and merged into the population one-by-one. A regular steady-state algorithm processes each solution serially, which makes them unsuitable for parallel workloads. A steady-state asynchronous algorithm removes the serial requirement of a regular steady-state algorithm. It processes solutions asynchronously so that multiple solutions can be handled simultaneously. Unlike a generational algorithm, a steady-state asynchronous algorithm does not have to wait for the entire generation to be
evaluated before creating more offspring. On the downside, a steady-state asynchronous algorithm may suffer from lower function evaluation efficiency because of something called selection lag that is discussed later.

Having information about the relationship between runtime and cost allows the practitioner to make an informed decision on whether to spend on more computing resources for faster results or wait longer to cut costs. However, this trade-off is largely unknown for evolutionary algorithms. The aim of this study is to experimentally investigate this trade-off for two evolutionary algorithms, NSGA-II [2] and Differential Evolution [3] using a master-slave architecture when applied on a discrete-event simulation problem. The primary aims of this study are:

- Compare steady-state asynchronous and generational versions of NSGA-II and multi-objective DE.
- Analyze the runtime and cost trade-offs for all included algorithms.

Another complicating factor is that the optimal values for the other control parameters might be different for different offspring population sizes. This aspect will be addressed by performing parameter tuning experiments on the control parameters, included the offspring population size. A bilevel optimization is used where the upper level optimization involves decision variables that are the control parameters of the lower level optimization. The upper level objectives are minimization of the runtime and the cost of optimization. The lower level objective is to minimize the number function evaluations required to attain a performance target.

The approach is to perform parameter tuning experiments using a simplified version of the DES-based optimization problem. A subset of the trade-off solutions from those experiments will be replicated on a commercial cloud computing platform [4].

2. Background

This section gives a short introduction to parallel evolutionary algorithms and related works.

2.1. Parallel Evolutionary Algorithms

Evolutionary algorithms (EAs) are a category of metaheuristics that are inspired by concepts from nature, in general, and evolution by natural selection, in particular. They maintain a population of individuals which have to compete for survival. New offspring are created by recombining and mutating individuals selected from the population. A selection pressure that drives the population towards better solutions is created by favoring better individuals when selecting parents and also when deciding which individuals survive into the next generation. There are two main strategies for handling new offspring in an EA, generational and steady-state evolution. A generational EA has separate
stages, performed in series, for generating new individuals and merging those individuals back into the population. In contrast, a steady-state EA immediately updates the population after a new individual is created and evaluated. An asynchronous steady-state algorithm performs multiple offspring creation, evaluation and update steps in parallel. The advantage of a steady-state asynchronous algorithm in a parallel computing context is high utilization of the available resources, since the algorithm never has to wait for any specific evaluation. In contrast, a generational algorithm must wait for all evaluations to be completed before it can proceed to the next generation. However, this also means that a steady-state asynchronous EA will have less information than a generational EA at any given generation. The authors of [5] calls this selection lag and define it as the number of solutions that are created while an observed solution is being evaluated. The result of an optimization is also dependent on the order in which evaluations are completed.

The above differences are more apparent when solving simulation-based optimization problems where the simulation times can differ drastically from one solution to another. There is a concern that a steady-state asynchronous algorithm will have a bias towards fast-evaluating solutions, since they are bound to be created, evaluated and merged into the population more frequently than slow-evaluating solutions. This question was investigated in [6] where the authors found that this effect is generally negligible.

The parallelization potential of both generational and steady-state asynchronous EAs is determined by the offspring population size. For a generational EA, it controls the number of new individuals that are created in each generation and for a steady-state asynchronous EA, it is the maximum simultaneous evaluations that can exist. This means that the maximum parallelism is capped by the offspring population size. Increasing this number will increase the parallelism, but it will also affect the search behavior of the algorithm, possibly in a negative way. The trade-off is therefore between increased parallelization potential and the negative effects on the search behavior.

The parallel architecture used in this paper is the master-slave model and it is arguably the simplest to implement. There are, however, other parallel architectures through which the parallelism of an EA can be increased. For example, by combining multiple cooperating EAs that exchange candidate solutions among each other using distributed islands or cellular models [7]. The advantage of these methods are that it is possible to achieve synergistic effects. However, they are also more complicated to implement and it is not trivial to know what algorithms to combine and in what way.

2.2. Related Work

A performance comparison between generational and steady-state asynchronous evolutionary algorithms (NSGA-II and SPEA2) was presented in [8]. They showed that a substantial speedup can be achieved with steady-state asynchronous algorithms. They reported a speedup between 12 and 25 percent compared to the generational algorithms. They also concluded that the variance in time it took to evaluate a solution and parallelization ratio are the key factors.
that influence the relative performance between steady-state asynchronous and generational algorithms.

The authors of [6] performed both theoretical and practical analysis of a simple asynchronous master-slave evolutionary algorithm. They found that the amount of increased throughput that can be expected from an asynchronous EA depends on the number of slave processors, the size of the population and the variance of the evaluation time distribution. Another aspect they investigated was if there is a bias towards fast-evaluating solutions in asynchronous EAs. They found no evidence in support of this hypothesis.

An asynchronous master-slave implementation of DE was presented in [5]. Experimental results showed that the asynchronous version outperformed the synchronous version even when a heterogeneous computing architecture is used. They also investigated the selection lag that arises in asynchronous algorithms and found that it adversely affects the algorithm, but also that it was negligible for cases when the number of processors is about half the population size.

The authors of [1] investigated the role of the offspring population size in evolutionary algorithms and how it relates to parallel evolutionary algorithms. Based on an analysis of simple problems they conclude that the offspring population size should be roughly the reciprocal of the success probability, i.e. the probability of creating an improved offspring. They found that the performance was moderately affected by increasing the offspring population size up to a point, after which the penalties are more severe.

2.3. Algorithm Configuration

The no free lunch theorem states that no algorithm can be better than all other algorithms on all problems [9]. To allow some flexibility, algorithms usually reserve important design decisions as control parameters to be specified by the user. The no free lunch theorem also applies to the control parameters. For example, a relatively simple problem might benefit from greedy control parameter values (CPVs), while the same CPVs applied to another problem may cause the algorithm to get stuck in local optimum. It is, however, difficult to predict which CPVs will work well for any given problem. This means that the practitioner must rely on past studies and heuristics for selecting appropriate CPVs. This issue can be mitigated by designing the CPVs to work well across a range of problems, thus yielding a generalist. In contrast, a specialist is a CPV tuple that is only tuned to work well for a small set of similar problems. The performance difference between a generalist and specialist on the same problem can be significant [10, 11].

2.4. Multi-objective Optimization

This paper considers multi-objective optimization problems as described in Equation (2), where \( M \) is the number of objectives, \( x \in \mathbb{R}^N \) is a vector of \( N \) decision variables and \( f \) represents a vector of objective functions that map a
solution in the decision space to the objective space as $f_i : \mathbb{R}^N \to \mathbb{R}$.

Minimize $x \{ f_1(x), f_2(x), \ldots, f_M(x) \}$
Subject to $x_l \leq x \leq x_u$ \hspace{1cm} (2)

Comparing solutions in a single-objective optimization is trivial, since the value of the objective can be used directly in the comparison. Another approach is needed for optimization problems with multiple conflicting objectives, since a solution can be better for one objective and worse for another. One approach is to convert the multi-objective problem into a single-objective problem by assigning a weight to each objective and using the sum of these modified objectives when comparing solutions. The biggest drawback of this approach is that the weights must be assigned before the optimization starts, and it can be difficult to know what the appropriate weights should be. Instead, the concept of domination can be used to compare solutions in a multi-objective problem. For a minimization problem, a solution, $u$, is said to dominate another solution, $v$, if $f_i(u) \leq f_i(v)$ for all $i = 1, \ldots, M$ and $f_i(u) < f_i(v)$ for at least one $i = 1, \ldots, M$. When neither solution dominates the other, the two solutions are said to be non-dominated. The result of a multi-objective optimization is a set of non-dominated solutions. When no other feasible solutions in the search space dominate these solutions, they are said to be Pareto-optimal. Two main properties are desired in a set of non-dominated solutions; convergence and diversity. The convergence of a non-dominated set is a measure of how close it is to the Pareto-optimal set, whereas the diversity measures the spread of the solutions within the set.

In order to simplify the comparison of CPV tuples at the upper level optimization, a unary performance indicator of the optimization at the lower level is desired. The hypervolume indicator is chosen for this purpose ($I_H$) [12]. The hypervolume of a non-dominated solution set $S$ obtained from an MOEA is defined as:

$$I_H(S, r) = \lambda \left( \bigcup_{x \in S} H(F(x), r) \right),$$

where, $r$ is a reference point that is dominated by all solutions in $S$, $\lambda(\cdot)$ is the Lebesgue measure, and $H(a, b)$ represents the hypercube with body-diagonal $ab$. Solution sets with higher $I_H$ values are better because they dominate a larger region of the objective space. Hypervolume is the only known unary quality indicator that is Pareto-compliant [13], i.e. for any two solution sets $S_1$ and $S_2$ ($\neq S_1$), $S_1 \preceq S_2 \Rightarrow I_H(S_1) > I_H(S_2)$. The hypervolume indicator does not require the knowledge of the Pareto-optimal set. However, it is sensitive to the inclusion or exclusion of extremal points and the relative ranges of the objectives. It is also computationally expensive to calculate for large number of objectives.
3. Simulation-based Optimization

Simulation models of real-world processes are often built in order to test new ideas and improvements before they are implemented in the real world. This enables simulation studies where the aim is to find the best performing set of design variables according to predetermined criteria. The design variables could, for example, be improvements to machine processing times or the size of buffers. Any change to the design variables are evaluated against some criteria, such as throughput or work in process. Such simulation studies enable rapid experimentation with minimal disturbance and provide the decision maker with additional confidence about the effectiveness of the changes.

The number of possible solutions grows exponentially as the number of design variables is increased. It is therefore only feasible to perform manual or exhaustive experimentation when the design variables are relatively few. An alternative is to formulate the question to be answered as an optimization problem and solve it with an optimization algorithm. Since the optimization algorithm will only test a small fraction of the available solutions, more design variables can be included in the problem formulation than would have been possible with manual experimentation. This is referred to as simulation-based optimization (SBO) or simulation-based multi-objective optimization (SMO) for problems with more than one objective.

SMOs are difficult to solve because of:

- Interacting variables. Changing a buffer size might require a corresponding change to the processing time of an earlier machine.
- Noise. Simulation models can have stochastic elements such as breakdowns.
- Multiple non-linear objectives. With multiple conflicting objectives, such as throughput and lead time, there is more than one optimal solution and the optimization need to find this trade-off set.
- Multiple local optima. When there are multiple local optima it is possible that the optimization can get stuck and fail to find the global optima.

The computational cost of solving an optimization problem with these characteristics can be significant because of the vast search spaces that need to be navigated. This is especially true for SMO since the time of running a simulation can be measured in minutes or even hours. Utilizing parallel computing resources efficiently enable larger optimization to be performed, or alternatively the results to be obtained faster.

3.1. FACTS Analyzer As a Cloud-Based SMO Software

With the recent trend of computing resources that can be distributed on-demand as services provided in a “cloud”, cloud computing has become the state-of-the-art platform for the research and implementation of parallel and distributed simulations. As defined by NIST, “cloud computing is a model for
enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [14]. Cloud-based simulation can, therefore, be thought of as simulation functions on cloud services. It has now become an attractive platform for simulation practitioners [15]. Cloud-based simulation, built on the concept of Software-as-a-Service (SaaS), can provide a lower-cost, highly scalable platform for small or large-scale simulations, particularly attractive to SMEs [16], because it provides a cheaper alternative to investing large capital outlays in hardware with low utilization, high complexity, and high labor cost of software maintenance [17]. With the cloud-based simulation environment, SMEs can easily access state-of-the-art simulation technology and powerful computing infrastructure which they cannot afford otherwise.

With the advancements of simulation-based optimization technologies and considering that optimization runs are much more computationally expensive than ordinary simulation runs, it can be envisaged that the forthcoming challenge is the realization of cloud-based simulation-optimization platforms. FACTS Analyzer (FACTS for short) and its underlying optimization component is designed to meet this challenge. FACTS has been developed with the objective to promote manufacturing executives to play the role of simulation users, particularly in the conceptual phase [18]. SMO is an integral module of FACTS that facilitate managers/engineers to run advanced optimizations in order to seek the optimal combinations of design variables to support more confident decision making in the design and improvement of production systems. Unlike other SMO software, FACTS promotes the use of post-optimality analysis of the Pareto-optimal solutions so that deeper knowledge can be gained, for instance on the relationships of the key influencing variables on the output objectives for the purpose of decision support. Real-world applications of this SBO-based decision making process has been demonstrated in ([18], [19], [20]).

3.2. A Scalable Simulation Model

The simple stochastic simulation model considered in the experiments of this paper represents an un-paced flow line, consisting of $s$ workstations with $s - 1$ inter-station buffers. The terms ‘workstation’ and ‘machine’ are interchangeable in the following discussions, since there is only one machine in each workstation. The productivity of each machine $i$ is governed by its availability ($\alpha_i$), processing time ($\beta_i$) and repair time ($\gamma_i$). In the initial state, the workloads of all the workstations are perfectly balanced, each having a processing time of $\beta_{\text{orig}} = 80$ seconds per job. All machines have an availability of $\alpha_{\text{orig}} = 90\%$ and repair time of $\gamma_{\text{orig}} = 300$ seconds. The processing times are assumed to be constant, which is realistic for automated machining processes. The times to failure of the workstations are modeled with exponential distributions and the randomness of the repair times $\gamma_i$ is modeled using Erlang distributions.

In a complex flow line with unbalanced workloads, the detection of bottlenecks is essential for a subsequent improvement of the production rate or throughput. The location of bottlenecks depends on many factors, including
the job flow logic, variability and disturbance of the machines and the buffer allocations. Even for a simple, straight flow line with balanced workloads as described above, detecting which workstation(s) to improve in order to increase the overall throughput of the line to a certain level is not a trivial task. The concept of treating this throughput improvement problem as a multi-objective optimization problem of identifying the optimal (minimal) number of changes to maximize the throughput was first proposed in [21] and later further elaborated in [22]. In such an optimization formulation, the system throughput (TH) is the primary objective for improvement, so that \( f_1(x) = \max\{TH(x)\} \). The total number of changes, i.e. improvement actions, can be defined as the secondary objective function, \( f_2(x) \). There are three integer improvement variables \( \{\alpha_i, \beta_i, \gamma_i\} \). The available improvement actions for availability, processing times and repair times and their corresponding step-sizes are:

\[
\begin{align*}
\alpha &= \{80, 96\} & \Delta \alpha &= 1 \\
\beta &= \{60, 80\} & \Delta \beta &= 1 \\
\gamma &= \{180, 360\} & \Delta \gamma &= 1
\end{align*}
\]

The second objective, \( f_2(x) \), can then be written as a summation of improvements [22]:

\[
f_2(x) = \min \left\{ \sum_{i=1}^{s} \tilde{\alpha}_i + \sum_{i=1}^{s} \tilde{\beta}_i + \sum_{i=1}^{s} \tilde{\gamma}_i \right\}
\]

where

\[
\begin{align*}
\tilde{\alpha}_i &= \frac{\alpha_i - \alpha_{i}^{\text{orig}}}{\Delta \alpha}, \\
\tilde{\beta}_i &= \frac{\beta_{i}^{\text{orig}} - \beta_{i}}{\Delta \beta}, \\
\tilde{\gamma}_i &= \frac{\gamma_{i}^{\text{orig}} - \gamma_{i}}{\Delta \gamma}
\end{align*}
\]

Additionally, in order to simultaneously solve the lean buffer problem [23], the capacity of inter-station buffer spaces are also optimized, \( B_i = \{1, 2, \ldots, 10\} \ \forall \ i \in \{1, \ldots, s - 1\} \), by adding a third objective of minimizing the total number of buffers, i.e. \( f_3(x) = \min \{\sum_{i=1}^{s-1} B_i\} \).

The model described in this section is developed and simulated using FACTS. A three machine configuration is used for a total of 11 decision variables. The short model is run for 2 days with 6 hour warmup and 3 replications. The long model is run for 60 days with a 6 day warmup and 15 replications.

4. Experiment Design

The aim of this study is to investigate the trade-off between runtime and cost, for generational NSGA-II (NSGA-II-GEN), steady-state asynchronous NSGA-II (NSGA-II-SSA), generational DE (DE-GEN) and steady-state asynchronous DE (DE-SSA). This section describes the experiments that will form the basis of the analysis.

The offspring population size has direct control over the level of parallelization of the algorithms. So the simplest method for comparing the algorithms is to run each algorithm with different offspring population sizes. There are at least two problems with this approach. First, heuristics or recommended values
have to be used for the other control parameters since it is difficult to know what their optimal values are. Second, it is not guaranteed that optimal values for the other control parameters remain unchanged for different values of the offspring population size. The experiments in this paper are separated into two steps in order to handle these two issues.

The first step is to perform parameter tuning experiments on all four algorithm instances. These experiments will be performed on a faster version of the DES problem which has fewer replications and an easier performance target. This is necessary because the tuning experiments are computationally expensive since multiple optimizations have to be performed for each upper level evaluation. The second step is to take a subset of the solutions (the CPVs) from the tuning experiments and run them with the full DES model on Digital Oceans cloud computing platform. The rest of this section will describe the experiments in more detail, after a brief description of NSGA-II, DE and their control parameters.

4.1. NSGA-II

NSGA-II [2] is an evolutionary multi-objective optimization algorithm that mimics the process of natural evolution. It starts with a random population of individuals that are evolved over time by crossing over and mutating individuals. The crossover operator is SBX and the mutation is polynomial. A fixed number of new individuals (the offspring population size) are created at the start of each generation and sent for evaluation. The algorithm then waits for all evaluations to be completed after which they are merged back into the population. The population is then trimmed down to the population size by preferring non-dominated individuals and as a second criterion the least crowded individuals.

The following parameters, common for both NSGA-II-GEN and NSGA-II-SSA, are tuned in this study:

1. Population size ($\mu$): The number of individuals in the population. An integer in the range [10, 300].
2. Offspring population size ($\lambda$): The number of simultaneous offspring. An integer in the range [32, 3200] in steps of 32.
3. Tournament size ($ts$): The number of tournament rounds. A value of 1 means that parents are selected randomly from the population. An integer in the range [1, 20].
4. Mutation probability ($p_m$): The probability of random changes to the decision variables as $p_m / N$, where $N$ is the number of decision variables. A real-value in the range [0, 5].
5. Mutation distribution index ($\eta_m$): Index governing the proximity of the mutated child to its parent. Larger values mean a smaller change to the original value. A real-value in the range [0, 500].
6. Crossover probability ($p_c$): The probability of creating offspring from two parents. A real-value in the range [0, 1].
7. Crossover distribution index ($\eta_c$): Index governing the proximity of the children to their parents. Larger values mean the children are closer to their parents. A real-value in the range [0, 500].
4.2. Differential Evolution

DE [3] is a population-based evolutionary algorithm. The original version is single-objective, but the version implemented here is extended to be able to handle multi-objective problems as well. It starts by filling the population with a random set of individuals. A new individual is then created by adding the weighted difference between two individuals in the population to a third individual. This new individual is then mixed with a fourth individual and is then added to the population. All parents are selected randomly from the population. The mutation operator is polynomial.

In each generation, a fixed number of individuals (the offspring population size) is created and sent for evaluation. When the entire generation is evaluated, they are merged back into the population. The population is then trimmed down to the population size using the same approach as in NSGA-II. That is, non-dominated individuals are preferred over dominated ones and crowding distance is used as a secondary criterion.

The following parameters, they are the same for both DE-GEN and DE-SSA, are tuned in this study.

1. Population size ($\mu$): The number of individuals in the population. An integer in the range $[10, 300]$.
2. Offspring population size ($\lambda$): The number of simultaneous offspring. An integer in the range $[32, 3200]$ in steps of 32.
3. Differential weight ($F$): Controls the amplification of the differential variation. A real-value in the range $[0, 5]$.
4. Crossover probability ($p_c$): The probability of accepting the differential perturbation for each decision variable. A real-value in the range $[0, 1]$.
5. Mutation probability ($p_m$): The probability of random changes to the decision variables as $p_m / N$, where $N$ is the number if decision variables. A real-value in the range $[0, 5]$.
6. Mutation distribution index ($\eta_m$): Index governing the proximity of the mutated child to its parent. Larger values mean a smaller change to the original value. A real-value in the range $[0, 500]$.

4.2.1. Steady-state Algorithms

The steady-state asynchronous adaptations are the same for both NSGA-II and DE. The difference between the generational and steady-state asynchronous versions is that the steady-state asynchronous versions immediately merge evaluated solutions back into the population. They do not wait for the entire generation to be ready. In the generational version, the offspring population size determines how many offspring that is created each generation. In the steady-state asynchronous version the offspring population size instead determines how many simultaneous individuals that are currently under evaluation. A dominance check is performed against the entire population when an individual is returned from evaluation. The new individual replaces the first individual that it dominates from the population. In the case that it is dominated it is instead discarded. If it neither dominates nor is dominated it is inserted into the
population and a crowding distance calculation is performed on the entire population. The individual with the least crowding distance is then removed from the population. A new individual is then created from the updated population and sent for evaluation.

4.3. Parameter Tuning Experiments

It would not be possible to run the parameter tuning experiments on the longer DES model, because of the time it would take to run multiple lower-level optimizations with a computational expensive model. Instead, the parameter tuning experiments are run on the shorter version of the DES model. Multiple lower level optimizations are also run in parallel in order to further speed up these experiments. It is therefore not possible to use the wall-clock time of each simulation, the simulation times are instead generated from an empirical distribution that was obtained by running a latin hypercube design of 10000 simulations with the longer DES model on the same cloud platform that will run the cloud experiments later. The empirical distribution is shown in Figure 1.

![Figure 1: Distribution of 10000 simulation runtimes](image)

NSGA-II is used as the tuner in upper level optimization. It is configured for an explicit exploration and exploitation stages by utilizing different CPVs for the different stages. Three stages are used for all experiments in this paper. The CPVs for each stage is shown in Table 1. Only CPVs with 25 replications are considered for the cloud experiments. The objective at the lower level is to minimize the number of evaluations to a target hypervolume. For certain CPVs this may never happen or take a very long time. To handle this issue the maximum number of evaluations is set to 35000. CPVs that can not get to the target hypervolume within that limit are assigned a constraint violation given by the difference between the target hypervolume and the achieved hypervolume. This will allow the upper level optimization to be still able to differentiate between CPVs that are unable to reach the target hypervolume.

4.4. Cloud Experiments

The purpose of the cloud experiments is to run a subset of the best solutions from the parameter tuning experiments on a cloud computing service and record
Table 1: Experiment settings and configuration for the different stages.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Offspring population size</td>
<td>100</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>3</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>Mutation Distribution Index</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.9</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>Crossover Distribution Index</td>
<td>10</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Tournament size</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Replications</td>
<td>1</td>
<td>10</td>
<td>25</td>
</tr>
</tbody>
</table>

the actual runtime of each experiment. Digital Ocean is chosen as the cloud provider because of its competitive and simple pricing model. Digital Ocean refers to a computing instance as a droplet. The longer DES model is used for all cloud experiments. The average simulation time is around 3 seconds which is relatively short, see Figure 1. There are two reasons for that. First, a shorter simulation time means that any computational overhead in the optimization algorithm will have a higher impact. The parameter tuning experiments do not take this aspect into account, so if it has a significant effect, it should show up as a discrepancy between the parameter tuning experiments and the cloud experiments. Second, a shorter simulation time also allows a higher overall number of optimizations, since each optimization takes less time.

The offspring population size is limited to multiples of 32 in order to limit the number of droplets used while also simplifying the process of exactly matching the offspring population size to the amount of cores available.

The measured outcome of each cloud experiment is the runtime and cost. Runtime is measured as the wall clock time from when the optimization is started to when the target hypervolume is reached. The cost is calculated as:

\[
droplets = o_o / o_r
\]

\[
cost = r \ast droplets
\]

where droplets are the number of droplets that is needed, \(o_o\) is the size of the offspring population and \(o_r\) is the resolution of the offspring population size (32) and \(r\) is runtime of the optimization.

To keep the overall cost of experiments down, only a subset of all solutions from the parameter tuning experiments is selected for the cloud experiments. Although the parameter tuning experiments were run up to an offspring population size of 3200, only solutions with an offspring population size of less than 2000 are considered for the cloud experiments. K-means clustering is used to
identify 13 clusters from which the solution nearest to the centroid of each cluster is selected. Additionally, the solutions with the highest and lowest offspring population sizes are also included in the experiments if they are not already present. Each experiment is run 5 times from which the mean runtime and cost are calculated.

5. Results

The results from the parameter tuning experiments, for all four algorithms, are shown in Figure 2. Figure 2a show the cost and time trade-off and Figure 2b show the evaluations versus time. The most important result is the cost versus time figure and it shows that the steady-state asynchronous algorithms provide a better trade-off since those completely dominate the generational algorithms. DE is also clearly better than NSGA-II for this DES problem; even DE-GEN dominates NSGA-II-SSA. The difference between DE-SSA and NSGA-II-SSA is significant. In terms of time, the fastest DE solution is twice as fast as the fastest NSGA-II solution (24 vs. 50 seconds), and in terms of cost DE is also almost twice as good (452 vs. 829).

The hypothesis that the steady-state asynchronous algorithms would have higher utilization but lower function evaluation efficiency than the generational algorithms is supported by these results. From Figure 2b it can also be concluded that the better utilization of resources of the steady-state asynchronous algorithms more than outweigh the worse function evaluation efficiency. The steady state algorithms have a lower cost than the generational algorithms, even though they require more function evaluations.

A subset of solutions from Figure 2 is selected to be replicated on the Digital Ocean’s cloud platform. The results from those experiments are shown in Figure 3. Only non-dominated solutions are shown. By comparing Figure 2a and Figure 3a it can be seen that the relative ordering of the algorithms is preserved in the cloud experiments. The steady-state asynchronous algorithms are better than the generational algorithms and DE is better than NSGA-II.
Figure 3a also shows that the generational algorithms use, slightly, fewer evaluations. However, that advantage is more than lost by their worser utilization of the computing resources.

![Figure 3](image1.png)

(a) Cost versus time trade-off 
(b) Evaluations versus time trade-off

Figure 3: Non-dominated solutions from cloud experiments

Figure 4 shows the offspring population size versus both the runtime and cost for the cloud experiments. The cost increases linearly with the offspring population size while the runtime decreases almost exponentially. There is, therefore, a region (sometimes referred to as the knee) in the trade-off front that provides a good balance between the runtime and cost. Improving one objective outside this region causes large sacrifices in the other objective. For these particular algorithms and problem combinations, the knee region includes solutions with the offspring population size between 200 to 400.

![Figure 4](image2.png)

(a) Time versus offspring population size 
(b) Cost versus offspring population size

Figure 4: Non-dominated solutions from cloud experiments

In Figure 5 all non-dominated solutions from the parameter tuning experiments for NSGA-II are shown as a parallel coordinates plot. As expected, the offspring population size is the main factor that gives rise to the trade-off between runtime and cost. The optimal parameters are similar for both NSGA-II-GEN and NSGA-II-SSA, with only some minor differences. NSGA-II-SSA is
not able to utilize the entire range of the offspring population size. The reason for this is that the function evaluation efficiency is so low over an offspring population size of 2000 that it cannot reach the target hypervolume within the maximum budget of 35000 evaluations. There are some notable differences in the tuned parameter values when compared with other commonly used values, such as $\lambda = 100$, $p_m = 1/N$, $\eta_m = 20$, $p_c = 0.9$, $ts = 2$ and $\eta_c = 20$. The tuned population size and crossover probability are similar to the recommended values. The mutation probability is higher, around $2/N$, which means that on average two decision variables are mutated for each offspring. It is also configured to be more elitist since the tournament size is between 4 and 8. $\eta_m$ is low so that mutation produces large changes and $\eta_c$ is high so that offspring are produced near their parents.

![Figure 5: Parallel coordinate plot over the objectives and parameter settings of all non-dominated CPV tuples for NSGA-II](image)

The parallel coordinates plot of all non-dominated solutions from the parameter tuning experiments for DE is shown in Figure 6. The most significant difference between DE-SSA and DE-GEN is that the optimal population size for DE-GEN is around 100 while the optimal population size for DE-SSA is just slightly below 300. The other parameter values are similar. There are some notable differences of the tuned parameter values when compared with other commonly used values, such as $\lambda = 100$, $p_c = 0.3$ and $F = 0.5$. The polynomial mutation probability is low and $\eta_m$ is high, indicating that the polynomial
mutation is not particularly beneficial. The differential weight is around 1 and the crossover probability is around 0.9, which is higher than what is usually recommended.

There is also no linear relationship between the offspring population size and the other parameters, for any of the algorithms, indicating that there is no simple heuristic that can be used to adjust the other control parameters based on the value of the offspring population size.

![Parallel coordinate plot over the objectives and parameter settings of all non-dominated CPV tuples for DE](image)

(a) Generational DE

(b) Steady State DE

Figure 6: Parallel coordinate plot over the objectives and parameter settings of all non-dominated CPV tuples for DE

6. Conclusions

This paper has empirically investigated the trade-off front between runtime and cost for two evolutionary algorithms, NSGA-II and DE. This trade-off appears as the offspring population size is varied. Increasing the offspring population size will lower the runtime of the optimization, but it will also cause the algorithm to become less efficient in terms of the number of function evaluations it requires to reach a certain performance target. Both generational and steady-state asynchronous versions of DE and NSGA-II are included.

The results indicate that the steady-state asynchronous version of each algorithm is better than the generational version, for the included DES-based problem, indicating that the loss of efficiency of the steady-state algorithms is
more than compensated by higher utilization of the available resources. Further, DE is much better than NSGA-II, up to twice as fast or less than half the cost. The tuned parameter values are most likely specific to this particular problem. Still, it is possible to observe some interesting patterns. For NSGA-II the tuned parameters were the same for both versions and mutations were both more probable and larger than what is usually recommended. The steady-state DE version had significant higher population size than the generational version, further experiments are needed to establish just how important that difference is.

The results indicate that it is the offspring population size that has the greatest influence on the trade-off between the runtime and the cost of an optimization. No linear relationship between the offspring population size and the other parameters were observed. It is therefore difficult to construct a heuristic for adapting the other control parameters given a specific offspring population size. The optimal values for the other control parameters were, in general, within a small range. This indicates that the best and simplest strategy is, therefore, to keep the other parameters static while only using the offspring population size to control the trade-off between runtime and cost. However, more experiments are needed in order to establish the importance of adapting the other control parameter values together with offspring population size.

The trade-off fronts for all algorithms clearly show a knee region in which there is a balance between runtime and cost. For the particular problem in consideration that is when the offspring population size is between 200 and 400. If cost is the only concern, the offspring population size should be set to a very low value for this particular problem. Large gains in the runtime can be seen up to an offspring population size of about 400, after which there are diminishing returns with increased costs.


A Parallel Computing Software Architecture for the Bilevel Parameter Tuning of Optimization Algorithms

Martin Andersson, Amos H.C. Ng, Sunith Bandaru

Abstract—Most optimization algorithms extract important algorithmic design decisions as control parameters. This is necessary because different problems can require different search strategies to be solved effectively. The control parameters allow for the optimization algorithm to be adapted to the problem at hand. It is however difficult to predict what the optimal control parameters are for any given problem. Finding these optimal control parameter values is referred to as the parameter tuning problem. One approach of solving the parameter tuning problem is to use a bilevel optimization where the parameter tuning problem itself is formulated as an optimization problem involving algorithmic performance as the objective(s). In this paper, we present a framework and architecture that can be used to solve large-scale parameter tuning problems using a bilevel optimization approach. The proposed framework is used to show that evolutionary algorithms are competitive as tuners against irace which is a state-of-the-art tuning method. Two evolutionary algorithms, differential evolution (DE) and a genetic algorithm (GA) are evaluated as tuner algorithms using the proposed framework and software architecture. The importance of replicating optimizations and avoiding local optima is also investigated. The architecture is deployed and tested by running millions of optimizations using a computing cluster. The results indicate that the evolutionary algorithms can consistently find better control parameter values than irace. The GA, however, needs to be configured for an explicit exploration and exploitation strategy in order avoid local optima.

I. INTRODUCTION

Selecting the appropriate control parameter values (CPVs) for an optimization algorithm is crucial since it can have a large impact on the performance. However, this can be a difficult task because it is not possible to perfectly predict how an algorithm, together with a particular CPV tuple, will perform on an arbitrary problem. One of the reasons for this is the stochastic and complex nature of optimization algorithms which make them difficult to analyze theoretically. Instead, the performance and generality of a CPV tuple must be experimentally assessed by applying it to diverse set of problems. The no free lunch theorem states that it is not possible to find a CPV tuple that will be optimal for all problems [26].

The problem of finding the optimal CPV tuple for a particular algorithm and problem(s) combination is referred to as the parameter tuning problem.

As evident from the literature survey above, there are many different approaches to tackle the parameter tuning problem. Manually constructing and evaluating CPV tuples in an ad hoc manner is only feasible when a small number of them is tested. For larger tuning budgets a more systematic approach is required. One such approach is to formulate the parameter tuning problem as an optimization problem involving algorithmic performances as the objective(s). This is a challenging optimization problem because of nonlinear nature of performance metrics, interacting control parameters, multiple local optima and noise in performance metrics due to the stochastic nature of optimization algorithm. However, it is on precisely this kind of difficult optimization problems that evolutionary algorithms have been shown to be effective [10].

Solving a parameter tuning problem requires computationally expensive experiments, since evaluating a CPV tuple requires that a complete optimization run be performed. The performance of the tuner is therefore important for the problem to be solved to a satisfactory level with a relatively small tuning budget. This paper investigates two aspects of evolutionary algorithms and how they relate to the performance of the algorithm, namely the search strategy and noise handling capability. The search strategy is controlled by changing the CPVs and since they are mostly tested on test problems they might not be optimal for parameter tuning problems. Optimization algorithms are usually also stochastic, which means that multiple optimization replications are needed in order to find a good and robust CPV tuple. Performing replications will, however, reduce the number of unique CPV tuples that can be evaluated within a given tuning budget. Thus, there is a trade-off between the quantity and quality of CPV tuples. Various resampling strategies will be tested in order to assess the importance of optimization replications.

Solving the parameter tuning problem using a bilevel optimization approach is not a new idea, and goes back to the 1980s. However, these early attempts had limited success because bilevel optimization is computationally expensive. With recent advances in computational power and reduction in the cost of hardware, it is now feasible to utilize many computers for large-scale experiments. However, this requires a unique framework and a compatible software architecture that can efficiently utilize parallel computing cores. This paper presents such a framework and an architecture that can scale up to hundreds of computers running millions of optimizations.
II. EXISTING STUDIES ON PARAMETER TUNING

One of the earliest examples of using evolutionary algorithms to solve the parameter tuning problem can be found in [17]. The authors only used a single run of the evolutionary algorithm, but the approach was later extended to multiple runs in [11]. Although the study succeeded in finding good control parameter values, the computational complexity limited the scope of the experiments and also their usefulness.

An early attempt of utilizing parallel computing resources for parameter tuning was presented in [8]. A master-slave approach was used to run the lower-level optimization on separate processors.

In [18], the authors use a Graeco-Latin square design with a wide range for each parameter setting. This was followed by fully crossed factorial design with narrower ranges. This allowed finding interactions between the algorithm performance and the control parameters (specifically population size). The exponential nature of the factorial design limited the number of interactions that could be modeled.

In [5], the authors use sequential parameter optimization (SPO) in three different parameter tuning scenarios: a new algorithm on well-known problems, well-known algorithm on well-known problems and well-known algorithms on complex real-world problems. SPO consists of three iterative stages. The first stage selects new design points. The second stage estimates the algorithm’s performance by running an optimization with the selected parameter configurations. The final stage builds a model that estimates the algorithm’s performance for untested parameter configurations. The authors conclude that SPO found parameter configuration that led to increased performance in all three scenarios.

The authors of [3] propose CALIBRA, a parameter tuning method that uses Taguchi’s factorial experimental designs with a local search to find the best parameter configuration for algorithms with up to five control parameters. The authors tested their method on six different problems and found that it could find better parameter configurations for some, but not all, of them.

ParamILS is an iterated local search method [13] that uses a combination of default and random settings for initialization, which are then perturbed by iterative local search method with a fixed number of random moves. It also has random restarts and neighboring exchange mechanisms to avoid converging to sub-optimal parameters.

Relevance Estimation and Value Calibration (REVAC) [19] is a method based on information theory to measure parameter relevance. It is an estimation of distribution algorithm that estimates the expected performance of the parameter values chosen from a probability density distribution. A distribution with a narrow peak indicates a highly relevant control parameter and a broad plateau suggests a moderately performing control parameter. In [22] the authors use REVAC to improve an award-winning algorithm.

A comparison of different parameter tuning methods and the effects of two add-ons: racing and sharpening, are presented in [21]. The included parameter tuning methods are: Meta-EA with CMA-ES, REVAC, and SPO. The two add-ons are opposing forces since racing reduces the number of tests and sharpening increases them. The add-ons are also combined to get the benefit of both. The authors conclude that Meta-EA with CMA-ES is preferred if the purpose is to find the best parameter configuration.

Sequential model-based optimization (SMBO) is a model-based regression method. Three key limitations with SMBO for parameter tuning are identified in [12]: (i) it only supports numerical parameters; (ii) it can only tune specialists; (iii) it cannot eliminate poorly performing parameter configurations early. The main contribution in [12] was to remove the first two constraints. The authors tested their improvements by tuning CPLEX on 17 configurations and found that it could find equal or better parameter configurations than all other included parameter tuning approaches.

A parameter tuning method that can find the best parameter configurations for multiple computational budgets in a single run was proposed in [7]. It uses a rank-based approach inspired by the non-dominated sorting mechanism of NSGA-II. The authors use this method to tune the parent and offspring population size of CMA-ES on eight test problems. The authors conclude that their flexible budget method can find CPVs that are as good as those from a fixed budget method, but with substantial time savings.

A bilevel optimization approach to solve the parameter tuning problem was presented in [20]. Quadratic approximations are used to reduce the number of lower-level optimizations, thereby reducing the overall computational complexity. The optimization algorithms they tuned were DE (differential weight and crossover probability) and Nelder-Mead simplex search (reflection coefficient, expansion coefficient and contraction coefficient). The results show that their algorithm was able to converge close to best parameter values for most cases.

In [25] the authors used NSGA-II to solve a bi-objective parameter tuning problem which takes into consideration both the precision and speed when tuning DE and particle swarm optimization (PSO). Precision was defined as the best result obtained after a certain number of fitness evaluations. Speed was defined as the number of function evaluations needed to reach a minimum fitness threshold. From the non-dominated front of these two objectives they could extract knowledge about the effects of control parameters. For example, they found that the population size mainly influences the location of a configuration on the Pareto-optimal front.

A multi-objective evolutionary algorithm (NSGA-III) to find robust control parameters for multi-objective optimization algorithms was presented in [15]. The task of finding robust or generalist CPVs requires that the they are tested on multiple problems. Here, the approach was to use the hypervolume from each problem as a separate upper level objective. Although only two problems were used, the study found that the optimal parameter configurations were different for the problems.

The Many-Objective Tuning Algorithm (MOTA) was proposed in [9]. It has in its problem formulation speed as both an objective and a decision variable. Each CPV tuple is assessed against multiple function evaluation budgets. It also has a preemptive terminating strategy for noise handling. The results show that MOTA was able to outperform other parameter
In the second step, the best configurations are selected from the newly sampled ones using racing. The final step is to update the sampling distribution in order to bias it towards the best configurations. These steps are repeated until a stopping criterion is met. The input to \textit{irace} is one or more problem instances that are used during each race. The order in which the instances are used is randomized. Racing refers to the process of systematically eliminating statistically worse configurations in a series of races or iterations. Each race starts by running a minimum number of instances (the default is five) after which the parameter configurations that perform statistically worse are discarded. Then additional instances are run for each parameter configuration so that more parameter configuration can be discarded. This process is repeated until only a few parameter configurations are left and the race is completed.

Each parameter has its own independent sampling distribution. Numerical and ordinal parameters use a truncated normal distribution, while categorical parameters use a discrete distribution. The statistical test can be any appropriate test, the default choice in \textit{irace} is Friedman’s non-parametric two-way analysis of variance by ranks.

**III. APPROACHING THE PARAMETER TUNING PROBLEM**

There are many different ways to approach the parameter tuning problem. This section will discuss some of the aspects that need to be considered and the design decisions that were made in this paper.

### A. Parameter tuning

Parameter tuning can be divided into three layers, design layer, algorithm layer, and application layer [10]. On the design layer, also called upper-level optimization, is the tuner that decides which control parameter configurations to test. The optimization algorithm that is tuned is on the algorithm layer, which is also called the lower-level optimization. On the application layer is the problem that is being solved by the optimization on the algorithm layer. In order to avoid confusion when discussing the quality of solutions on the different layers the authors of [10] suggest to use the standard term ‘fitness’ to refer to the quality of solutions on the application layer and the term ‘utility’ for the quality of a CPV tuple.

### B. Upper level performance metrics

The choice on how to assess the utility of a CPV tuple determines the utility landscape and therefore also what the best CPV tuple is. There are various ways to assess the performance of an optimization algorithm. For example, the number of function evaluations to a performance target, the fitness after a certain number of function evaluations or a success rate. When these methods are aggregated over a number of runs they are denoted as follows:

- Mean best fitness (MBF)
- Average number of solutions until a target is reached (AES)
- Success rate (SR)

All these methods require at least one pre-specified value. For MBF it is the budget given to the optimization algorithm and for AES it is the performance target, and possibly also a maximum budget to avoid spending too much effort on very bad CPV tuples. Selecting appropriate values for both MBF and AES can be difficult and it will also affect the optimal CPV tuple. This paper will only consider AES as the performance metric. However, the framework and architecture can as easily use MBF as well. The reason AES is chosen over MBF is that it is easier to interpret the relative difference between the number of function evaluations than it is to understand the relative difference between fitness values.

### C. Noise Handling in Evolutionary Algorithms

Since evolutionary algorithms are randomized, they generate different results with different seeds. This is something that needs to be considered when they are tuned. There are multiple ways of handling this noise. The simplest method is to do explicit averaging of a CPV tuple by running the optimization multiple times with different seeds. Population-based methods might also benefit from implicit averaging since they sample solutions from a local area [14]. However, resampling a CPV tuple with a fixed budget means that fewer unique tuples can be explored. Handling this trade-off between replications and testing new CPV tuples is an important aspect of a parameter tuner.

The simplest resampling strategy is to always resample each CPV tuple the same number of times. This is referred to as the static resampling strategy. Too few static replications and there is a possibility of selecting greedy parameters. While too many replications will limit the number of unique CPV tuples that can be explored, which could prevent the tuner from converging to the optimal CPV tuple. The advantage of this strategy is that it is simple to understand and implement. However, there are circumstances where a more flexible strategy could be beneficial. For example, early in the optimization, it might not be as important to do replications as it is in the later stages. Also, a dynamic resampling strategy could detect and discard poor performing CPV tuples before spending too much replication effort on them.

The bilevel approach used in this paper runs each problem instance at least once. The resampling strategy determines only the number of replications, not what problem instances to run. For parameter tuning scenarios with many problem instances,
this will result in a large number of minimum optimizations per parameter configuration. In such scenarios, it might be more efficient to remove poorly performing parameter configurations based on a subset of problem instances. That is the approach used by irace. The order of the problem instances is randomized at the start of irace. Problem instances are selected, in order, from this list when additional samples are needed. When the end of the list is reached, it starts over from the beginning of the list, but with a different seed. The drawback of this approach is that the order of the problem instances can cause otherwise good parameter configurations to be discarded. For example, in a parameter tuning scenario with 100 problem instances. A parameter configuration that performs well on 95 problems and poor on the remaining 5 could be discarded if by chance the problem instances on which it performs poorly are selected first.

When comparing tuning experiments that use different resampling strategies, it is important that they are compared in a fair way. One way to do this is to perform post-replications of the best CPV tuple found by the tuner and use the performance value obtained from the post-replications as its utility. If the same seeds are used for all post-replications it does not matter how many replications the tuner used during the search, since that value is not used as the final utility.

D. Specialist versus Generalist CPV tuples

A CPV tuple that is tuned against a single problem or a small set of similar problems becomes a specialist, while a CPV tuple tuned against large set of diverse becomes a generalist. The performance difference between a specialist and a generalist can be substantial [4], [23], which is not surprising given the no free lunch theorem.

A specialist tuned against a single performance metric will result in an upper level optimization that is single-objective. However, tuning a generalist means that the same CPV tuple is applied to optimizations over many different problems. That will produce a performance value for each problem instance. In order to keep the upper level optimization single-objective, these performance values need to be combined in some manner. The most straightforward way is to do a weighted summation of all performance values. The drawback of this method is that there could be a bias towards certain problems if their relative performance values differ. This can, of course, be mitigated by assigning appropriate weights to each performance value. One way to avoid this bias altogether is to use a multi-objective optimization algorithm at the upper level, and turn the performance of each problem instance into its own objective. However, for generalists with many problem instances and given that the budget of parameter tuning experiments is practically limited (because of the computational complexity), there is a risk that only non-dominated specialists are found and not generalists that are equally good at all problems. Therefore, the best method depends on the purpose of the generalist and there is no single right answer.

IV. PARALLEL ARCHITECTURE FOR PARAMETER TUNING

This paper uses a bilevel optimization approach to solve the parameter tuning problem. Since this approach is computationally expensive, it is desirable to have a software architecture that can efficiently use multiple cores on multiple computers. It should also be able to tune both specialist and generalist CPVs and handle both static and dynamic resampling strategies. Figure 1 shows a high-level view of the framework that is used in this paper.

A. Framework

At the design layer is the tuner (upper-level optimization). The objective of the tuner is to find the CPV tuple that minimizes the number of function evaluations needed to reach each problems target performance ($P_{target}$ for problem $q$). For each problem, $r$ number of replications is performed. Every problem that a CPV tuple is evaluated on receives the same number of replications. However, different CPV tuples might be replicated a different number of times. For example, in the exploring phase of the upper-level optimization, a low number of replications could be used, while the number of replications is increased later in the exploiting phase. Alternatively, heuristics together with statistical testing can be used to decided how to allocate replications on the entire set of CPV tuples. Common Random Numbers (CRNs) are used for all replications. This reduces the variance by using the same set of seeds for every CPV tuple. The drawback is that the CPVs can be overfitted to that set of seeds.

Both DE and GA are population based evolutionary algorithms. A population based algorithm can be either generational or steady-state. A generational algorithm consists of three main steps: first, new solutions are created; those solutions are then evaluated; finally, the new solutions are merged into the population. All these steps are performed serially. The number of solutions that are created in each generation is determined by the offspring population size. In contrast, a steady-state algorithm only creates one new solution at a time, evaluates it and merges it back into the population. A steady-state asynchronous algorithm does this step in parallel.

The versions of DE and GA used as tuners in this paper are generational. Thus, parallelism is capped by the offspring population size. However, the replications remain independent, meaning that all replications belonging to the same upper-level generation can be performed in parallel. So the parallelization potential in each generation is

$$\max_{parallelism} = \lambda \times r \times q$$  \hspace{1cm} (1)

where $\lambda$ is the offspring population size, $r$ is the number of replications and $q$ is the number of problems. Even for moderate values the maximum parallelization is quite substantial.

The upper-level tuner evaluates the utility of a CPV tuple $p$ by running a lower-level optimization with those parameters. This lower-level optimization can be single- or multi-objective, but it has to return a single performance measure. IGD and hypervolume are unary performance measures that can be used for this purpose. The lower-level optimization is run until the performance measure reaches the target or when the maximum number of FEs have been reached. Both the performance target and maximum FEs can be set individually for each problem. When the lower-level optimization has stopped it returns how
Utility(p∗) = ∑q,t=1∑rposts=1 FE
s
t (p∗)

Post tuning replications

Minimize
p
FE(p)

Subject to pl ≤ p ≤ ... and that the work load is distributed evenly over all computers in order to assure that computational resources

Control parameters (P) or FE

xq ≥ q,0 ∀ q

xq ≤ q, target

Stop when 

xq

FE

q

=q

FE

q

max

Mq

Fig. 1. Parameter tuning framework.

The software architecture that implements the framework described above is shown in Figure 2. The main architecture and the accompanying design decisions that were made are explained in this section. The main design goals of the architecture are: flexibility, scalability, and efficiency. The architecture is flexible in the sense that the different components can be combined in various ways to suit different requirements. The components can be combined into a single modular component that can easily be embedded into an application. Alternatively, the components can be distributed over many different computers for increased computational power. The architecture can therefore also scale from running on a single computer to several hundred computers without any modifications. Inefficient design decisions might not be noticeable if the architecture is only used for small deployments. However, for large deployments it is important that the amount of information being sent between different components is minimized and that the work load is distributed evenly over all computers in order to assure that computational resources

The average number of FEs (AES) is calculated when all replications for a problem instance with a particular CPV tuple have completed. The utility of that CPV tuple is then taken as the summation of all AES performance measures and then returned to the tuner. That cycle then continues until the stopping criterion for the upper-level optimization is met. After the upper-level optimization is completed the best CPV tuple (p∗) is replicated on a set of post replication seeds that are different from the seeds used during the tuning.

B. Software Architecture

Problem 1

Replication 1

Using p

Minimize

Subject to

Stop when

Fig. 1. Parameter tuning framework.

\( \text{Utility}(p^*) = \sum_{q,t=1}^{\sum_{r \text{post}} \sum_{s=1}^{\text{FE}} s^t (p^*)} \)

many FEs it performed and in the case that it never reached the target a constraint violation is assigned that is equal to the performance target minus the obtained performance. That allows the upper-level tuner to distinguish between two CPV tuples that both fail to reach the target performance. The cap on the number of FEs is needed in order to avoid spending excessive amount computation on very poor CPV tuples. The lower-level optimization could also get to a point where no progression is made, without a maximum cap it would go on forever.

However, for large deployments it is important that the amount of information being sent between different components is minimized and that the work load is distributed evenly over all computers in order to assure that computational resources
are maximally utilized.

Each solid box in Figure 2 represents one indivisible component. They can be run independently from each other on different computers when needed. The dashed boxes represent a logic grouping of components for the instantiation of the architecture used in this paper. Each opt-manager therefore only manages the optimizations running on the same computer and similarly, each worker-manager only manages workers running on the same computer. Another possibility would be to have a global opt-manager that handles every optimization instance in the network. However, such a configuration was not used as it is too centralized in that the opt-manager can become a bottleneck in the system. The outermost dashed box represents an instance that is run on one computer. Each computer can have a different number of cores, \( N_i \), where \( i \) is the computer number. \( \sum N_i \) determines the number of optimization and worker jobs that can be running simultaneously.

There are two types of optimizations in this architecture: direct evaluation and indirect evaluation. A direct evaluation optimization never sends any evaluations to the proxy-manager. It instead evaluates them directly in the same thread. An example would be when an algorithm that is being tuned on a test problem whose objective(s) are extremely cheap to evaluate. Such test problems are much more time consuming to send to a worker for evaluating than they are to simply evaluate them directly. Even if it is evaluated on the same computer (but on a different thread), the computational cost of serializing and sending the message through the different components will outweigh any asynchronous benefits. However, for more time consuming objective function evaluations it is better to send the job for evaluation asynchronously so that multiple jobs can be evaluated at the same time. This distinction has no impact on the framework or the experimental results. It is strictly a computational optimization that reduces the amount of work needed when the lower-level objective function(s) is(are) cheap to evaluate.

C. Implementation and Software Libraries

The entire system is implemented in C++ and uses various software libraries. Communication between components is performed by message passing. Having well defined interfaces between components also helps to reduce coupling between them. The two core external libraries that facilitate the message passing design are ZeroMQ [1] for the actual message passing and Protocol Buffers [2] for the serialization of messages. ZeroMQ is a lightweight and portable messaging library. It has native implementations in many different programming languages and wrappers for many more. It also runs on many different operating systems. It provides a thin abstraction layer over TCP/IP that helps it achieve a small overhead while simultaneously providing useful features such as different messaging patterns. ZeroMQ does not define a serialization protocol; its only job is to send bytes over a socket. The definition and serialization of messages are instead done with Protocol Buffers. Protocol Buffers started as an internal Google project but is now run as an open source project. Its goals are to provide a portable, language neutral and extensible way of serializing data. It is also available in many different programming languages and operating systems. Messages are defined in a special format that is fed to a compiler that creates code in a specified programming language. This code is then used to create, serialize, and deserialize messages. Since components communicate with each other with message passing, it does not matter if the components are located in the same process space or halfway around the world in two different data centers. This is what makes it possible for the system to be both flexible and scalable and this would be much harder without libraries such as ZeroMQ and Protocol Buffers.

The system supports multiple operating systems and has been tested on Microsoft Windows and Linux. It works on most operating systems that have a C++ compiler and implementations of ZeroMQ and Protocol Buffers. The system also supports deploying components to different operating systems, so that computers running various operating systems can be combined into a single system.

D. Information Flow

The arrows in Figure 2 represent all possible information flow and the labeled arrows represent an example information flow for a bilevel optimization tuning experiment. It starts with a user starting a bilevel optimization experiment by sending a start optimization request to the controller (A). The controller selects a suitable compute instance where the upper-level optimization will run and relays the request to its opt-manager (C). The opt-manager will spawn a thread and start an optimization of the requested type on it (D). The upper-level optimization will create a CPV tuple and send it to the proxy-manager for evaluation (E). The proxy-manager will gather a list of all opt-managers that can start at least one more optimization and randomly select one where the lower-level optimization will run (F). The opt-manager will spawn an optimization of the correct type and start the optimization on it (G). The lower-level optimization will create a candidate solution for the problem on the application layer. For a direct evaluation optimization the candidate solution is immediately evaluated and the lower-level optimization will continue to run until a stopping criterion is met and the utility of the CPV tuple is returned to the upper-level optimization. For an indirect optimization, the candidate solution is sent to the proxy-manager (H). The proxy-manager will gather a list of all worker-managers that can run at least one more evaluation job and randomly select one where the evaluation will be sent (I). The worker-manager selects a free worker and sends the candidate solution for evaluation (K). The worker evaluates it and returns the result to the worker-manager.

E. Controller

The controller and storage are the only two externally visible components. The controller’s main purpose is to serve as the entry point for starting new optimizations.
Fig. 2. Software architecture.

However, optimizations that are created as part of bilevel optimization experiment are not sent through the controller to avoid a single bottleneck point. It is also the only component that is cognizant of all available compute instances in the system. When a compute instance starts for the first time it registers itself with the controller and regularly checks back in so that the controller can detect and remove unresponsive compute instances.

F. Storage

The storage is the other externally visible component and it is responsible for storing and retrieving information about an optimization experiment. It uses SQLite which is file based SQL library that is both fast and portable. The advantage of using SQLite is that it is both small and easy to embed into an application, while also providing the benefits of structured storage of SQL. The drawbacks are that it is more difficult to handle many concurrent users compared to standalone SQL databases such as MySQL or Postgres. However, relying on an external SQL database would make the system less flexible and portable.

G. Optimization manager

The role of the opt-manager is to keep track of all running and idle optimizations. Each optimization is run on its own thread which is owned and managed by the opt-manager. The opt-manager only communicates with the optimization through message passing. The optimization is put into an idle state when it is first started. It stays in the idle state until it receives a start optimization request that includes the CPV tuple that should be used for that particular optimization experiment. When the optimization is finished, it is not destroyed but instead goes back into the idle state. This removes the overhead of spawning and destroying threads. It also avoids creating and destroying optimization algorithm objects such as sockets.

When an opt-manager receives a start optimization request, it first checks if the optimization type is a direct or indirect evaluation. The reason for distinguishing between optimization types is that a direct evaluation optimization takes up more resources than an indirect evaluation optimization. A direct evaluation optimization is therefore strictly limited to the number of processor cores on that machine, while indirect evaluation optimizations are allowed to exceed the number of processor cores. The opt-manager then checks how many optimizations of that type that are already running, and if the maximum is already reached the optimization is queued. If the optimization is allowed to start, it looks for idle optimizations of the same algorithm already running. When it finds an idle optimization of the correct type, it sends the start optimization request and the CPV tuple to it. The optimization is then removed from the idle list. If it cannot find an idle optimization of the correct type, it instead spawns a new thread running an optimization algorithm of that type and sends the request to it. When the optimization is completed, it is put back on the idle list.

H. Worker manager

The worker-manager is responsible for all workers located on the same machine. The worker-manager only communicates with its workers using message passing. Thus,
a worker can be run in a thread on the same machine or in a separate process on a different machine. The configuration used in this paper runs the worker in threads on the same machine as the worker-manager. The purpose of a worker is to perform expensive computations such as the evaluation of a simulation model. When a worker is first started, it registers itself with the worker-manager. A worker regularly checks in with its worker-manager so that it can be removed if it becomes unresponsive. For example, if the simulation software crashes or gets stuck due to a problem in the model. Before an optimization is started a new session is created and distributed to all workers. The session contains information that does not change during the optimization, such as the simulation model. This helps to keep the information that is sent during an optimization to a minimum. When a worker-manager receives a worker job, it sends it to the first available worker. In the case that there are no workers available the job is put into a queue and is sent as soon as another worker job finishes.

1. Proxy-manager

The purpose of the proxy-manager is to relay worker jobs and optimization jobs to a suitable opt-manager and worker-manager. The proxy-manager keeps track of all opt-managers and worker-managers in the system, including the ones that are located on the same machine. This means that when a proxy-manager receives a job, it can determine if the job is best run on the same machine or if it should be sent to another machine in the system based on the free workers available on each machine. The proxy-manager decides on where a job should be run, the opt-manager and worker-manager cannot resend jobs elsewhere. That is also why the proxy-manager communicates directly with opt-managers and worker-managers and not with other proxy-managers.

A proxy-manager only knows about its own state and it shares no information with any other proxy-manager in the system. This decentralized design is intended to reduce the amount of data sent throughout the system. A consequence of this design is that a proxy-manager only knows about the jobs that it has distributed itself to the other opt-managers and worker-managers in the system. A opt-manager or worker-manager can therefore be assigned more jobs than it has processor cores, so each manager has a queue of jobs where jobs are put that cannot be started right away. This helps to reduce networking overhead since this queue is local on each instance. The drawback of this decentralized design is that there is possibility that a particular instance is overburden with jobs while another instance has no jobs. However, the risk of this happening is small since jobs are distributed randomly. This has also not proved to be a practical problem.

J. Compute

The compute is a logical grouping of components, see Figure 2, that provide a single contact point to the controller. It manages the life-cycle of its components and relays start optimization requests to its opt-manager.

V. EXPERIMENT SETUP

This section will present the experiments that are performed in order to evaluate DE and GA as upper-level tuners. These two algorithms are configured with two different sets of CPV tuples. One set consists of recommended values for each algorithm, which tend to be more exploitative. Therefore, the other set is chosen to promote more exploration. These sets will be combined in two ways: (i) purely exploitation, and (ii) exploration followed by exploitation. This allows for an analysis of the usefulness of having an explicit exploration strategy. The latter strategy switches from exploration to exploitation when 75% of the tuning budget has been used.

The trade-off between replications and testing new CPV tuples is investigated by running experiments with 1, 5, 10, 15 and 20 static replications. Two dynamic resampling strategies are also included. The first is a step-resampling strategy that starts with 5 replications and then switches to 20 replications after 75% of the tuning budget has been spent. The second dynamic resampling strategy is time based. It starts with 1 replication and then, as the optimization progresses, gradually increases the number so that the last generation uses 20 replications. The number of samples, \( s \), is calculated in each generation as shown in Equations 2 and 3, where \( p \) is the progress, \( B \) is the total budget, \( B_{\text{max}} \) is the used budget, \( b_{\text{basis}} = 1 \) and \( b_{\text{max}} = 20 \) are the minimum and maximum number of replications and \( \lambda \) is the population size.

\[
p = \frac{B}{B_{\text{max}} - (b_{\text{basis}} \cdot \lambda)}
\]

\[
x_0 = p \cdot (b_{\text{max}} - b_{\text{basis}}) + b_{\text{basis}}
\]

The tuners are evaluated on their ability to tune both specialists and generalists over standard test functions. For the specialists, these are DE on the Ackley test function and GA on the Ellipse test function and for the generalists, these are DE and GA on Ackley, Ellipse, Sphere, Griewank, Rosenbrock and Rastrigin test functions. A mixture of single- and multi-modal optimization problems are chosen to provide a variety of challenging tuning problems. The configuration for each problem is shown in Table I. The combination of DE on the Ackley test function is chosen specifically to test the ability of the tuner to find robust CPV tuples. This combination is challenging because a greedy configured DE can sometimes solve the multi-modal Ackley test function problem very fast. However, for other starting conditions it can get stuck in a local optima without the ability to escape. The combination of GA on the Ellipse test function is meant to provide a tuning problem that has multiple local optima. That is, there are CPV tuples that are relatively easy to find which result in good performance. However, there are other significantly different CPV tuples that perform even better.

The two algorithms (DE and GA) are combined with the seven resampling strategies (step, time, static 1, static 5, static 10, static 15 and static 20) and two different sets of upper-level CPV tuples (i.e., exploitation only, and ii. exploration (75%) + exploitation (25%)), plus irace, for a total of \( 2 \cdot 7 \cdot 2 \cdot 1 = 29 \) configurations per experiment. Each configuration is replicated
TABLE I

<table>
<thead>
<tr>
<th>Problem</th>
<th>Dimension</th>
<th>Range</th>
<th>Performance Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ackley</td>
<td>20</td>
<td>[−10, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Ellipse</td>
<td>20</td>
<td>[−10, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Rosenbrock</td>
<td>20</td>
<td>[−5, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Rastring</td>
<td>10</td>
<td>[−10, 10]</td>
<td>0.1</td>
</tr>
<tr>
<td>Sphere</td>
<td>20</td>
<td>[−10, 10]</td>
<td>1</td>
</tr>
<tr>
<td>Griewank</td>
<td>20</td>
<td>[−400, 600]</td>
<td>1</td>
</tr>
</tbody>
</table>

20 times for each experiment. The two specialist parameter tuning problems are run with tuning budgets of 5000, 10,000, and 20,000. The two generalist parameter tuning problems are run with tuning budgets of 30,000 and 60,000. For a grand total of \(20 \times 3 + 2 \times 2 \times 10^2 = 5800\) parameter tuning experiments. To be able to compare experiments with different resampling strategies, the best CPV tuple from each experiment is replicated 100 times with a set of random seeds that was not used during the experiments.

The two different CPV tuples for the upper-level GA are shown in Table II. The corresponding CPV tuples for upper-level DE are shown in Table III. The version of irace used in the experiments is 2.3.1806 and its parameters are kept unchanged at their default values, since there was no attempt to tune irace; any comparisons with it are only valid for this particular configuration.

A. Algorithms at the lower-level

Two evolutionary algorithms are tuned in this study, a genetic algorithm and differential evolution. Following is a brief description of them and the control parameters that are tuned.

1) Genetic Algorithm: A GA is an evolutionary optimization algorithm that mimics the process of natural evolution. It starts with a random population of individuals that are evolved over time by crossing over and mutating individuals. In each generation, a fixed number of new individuals are added to the population. At the end of an generation, the population is trimmed by sorting the individuals by their fitness and removing the worst ones. The crossover operator is SBX and the mutation is polynomial.

2) Differential Evolution: DE [24] is a population based evolutionary algorithm. It starts by filling the population with a random set of individuals. A new individual is then created by adding the weighted difference between two individuals in the population to a third individual. This new individual is then recomposed with a fourth individual, and it replaces the parent if it has better fitness, otherwise it is discarded. All parents are selected randomly from the population.

3) Tournament size \((t_s)\): The number of tournament rounds. A value of 1 means that parents are selected randomly from the population. An integer in the range [1, 20].

4) Mutation probability \((p_m)\): The probability of random changes to the decision variables as \(p_m / N\), where \(N\) is the number of decision variables. A real-value in the range [0, 5].

5) Mutation Distribution Index \((\eta_m)\): Index governing the proximity of the mutated child to its parent. Larger values mean a smaller change to the original value. A real-value in the range [0, 500].

6) Crossover probability \((p_c)\): The probability of creating offspring from two parents. A real-value in the range [0, 1].

7) Crossover Distribution Index \((\eta_c)\): Index governing the proximity of the children to their parents. Larger values mean the children are closer to their parents. A real-value in the range [0, 500].

8) Differential weight \((F)\): Controls the amplification of the differential variation. A real-value in the range [0, 5].

9) Crossover probability \((p_c)\): The probability of accepting the differential perturbation for each decision variable. A real-value in the range [0, 1].
TABLE IV

<table>
<thead>
<tr>
<th>Rank</th>
<th>Algorithm</th>
<th>Replication</th>
<th>Configuration</th>
<th>Total rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DE</td>
<td>step</td>
<td>std</td>
<td>27315.5</td>
</tr>
<tr>
<td>2</td>
<td>DE</td>
<td>time</td>
<td>std</td>
<td>32921.5</td>
</tr>
<tr>
<td>3</td>
<td>DE</td>
<td>10</td>
<td>std</td>
<td>37866.5</td>
</tr>
<tr>
<td>4</td>
<td>DE</td>
<td>step</td>
<td>exp</td>
<td>39040.0</td>
</tr>
<tr>
<td>5</td>
<td>DE</td>
<td>5</td>
<td>exp</td>
<td>40359.5</td>
</tr>
<tr>
<td>6</td>
<td>DE</td>
<td>5</td>
<td>std</td>
<td>41902.5</td>
</tr>
<tr>
<td>7</td>
<td>GA</td>
<td>step</td>
<td>exp</td>
<td>45930.5</td>
</tr>
<tr>
<td>8</td>
<td>DE</td>
<td>10</td>
<td>exp</td>
<td>46926.0</td>
</tr>
<tr>
<td>9</td>
<td>DE</td>
<td>time</td>
<td>exp</td>
<td>47499.5</td>
</tr>
<tr>
<td>10</td>
<td>DE</td>
<td>15</td>
<td>std</td>
<td>50281.0</td>
</tr>
<tr>
<td>11</td>
<td>GA</td>
<td>5</td>
<td>exp</td>
<td>51966.5</td>
</tr>
<tr>
<td>12</td>
<td>GA</td>
<td>step</td>
<td>std</td>
<td>53732.0</td>
</tr>
<tr>
<td>13</td>
<td>GA</td>
<td>10</td>
<td>exp</td>
<td>53874.5</td>
</tr>
<tr>
<td>14</td>
<td>GA</td>
<td>time</td>
<td>exp</td>
<td>55051.0</td>
</tr>
<tr>
<td>15</td>
<td>DE</td>
<td>20</td>
<td>std</td>
<td>59322.5</td>
</tr>
<tr>
<td>16</td>
<td>GA</td>
<td>time</td>
<td>std</td>
<td>60300.0</td>
</tr>
<tr>
<td>17</td>
<td>DE</td>
<td>15</td>
<td>exp</td>
<td>60580.0</td>
</tr>
<tr>
<td>18</td>
<td>GA</td>
<td>10</td>
<td>std</td>
<td>63092.0</td>
</tr>
<tr>
<td>19</td>
<td>GA</td>
<td>15</td>
<td>exp</td>
<td>63703.5</td>
</tr>
<tr>
<td>20</td>
<td>GA</td>
<td>15</td>
<td>std</td>
<td>68367.5</td>
</tr>
<tr>
<td>21</td>
<td>DE</td>
<td>20</td>
<td>exp</td>
<td>68596.0</td>
</tr>
<tr>
<td>22</td>
<td>DE</td>
<td>1</td>
<td>exp</td>
<td>69991.5</td>
</tr>
<tr>
<td>23</td>
<td>GA</td>
<td>20</td>
<td>std</td>
<td>70960.0</td>
</tr>
<tr>
<td>24</td>
<td>GA</td>
<td>20</td>
<td>exp</td>
<td>70914.0</td>
</tr>
<tr>
<td>25</td>
<td>DE</td>
<td>1</td>
<td>std</td>
<td>71178.5</td>
</tr>
<tr>
<td>26</td>
<td>GA</td>
<td>5</td>
<td>std</td>
<td>72216.0</td>
</tr>
<tr>
<td>27</td>
<td>GA</td>
<td>1</td>
<td>exp</td>
<td>72490.5</td>
</tr>
<tr>
<td>28</td>
<td>GA</td>
<td>1</td>
<td>std</td>
<td>91443.0</td>
</tr>
<tr>
<td>29</td>
<td>irace</td>
<td></td>
<td></td>
<td>93314.0</td>
</tr>
</tbody>
</table>

VI. EXPERIMENT RESULTS

This section will present the results from from the evaluation of DE and GA as upper-level tuners. Table IV presents the combined ranking of all replications and experiments. DE stands out as the best overall tuner for the set of parameter tuning problems used in this study. The best performing configuration is the one with the step-resampling strategy and standard parameters (exploitative set). The configurations with 5 and 10 static replications also perform well. DE does not seem to benefit from exploration parameters, since the configurations with exploration parameters either perform similar or worse than the configurations using purely exploitative parameters.

The best GA configuration is the one with the step-resampling strategy and exploration parameters. Like DE, the best number of static replications are 5 and 10. The explicit exploration and exploitation strategy is beneficial for the GA, since configurations with only exploitation either have similar or worse performance than the configurations with the exploration strategy.

For both DE and GA, the best resampling strategies are the dynamic ones, followed by static 5 and static 10. Using a single static replication is among the worst strategies. The probable cause is that the parameters that are found are not robust enough when they are post replicated. On the other hand, when using 15 and 20 static replications the performance also declines, mainly because there are not enough unique evaluations for the evolutionary algorithm to properly explore the parameter space. It is also clear that both DE and GA, in most configurations, significantly outperform irace.

Figures 3-11 show the performance of the upper-level configurations for each of the parameter tuning problems. The utility of the best CPV tuple found from each of the 20 experiment replications are shown as box plots, where each experiment replication utility is the average of the 100 post replication experiments of the best CPV tuple found for each configuration. The configurations are sorted ascending from left to right by their mean performance. The configurations shown in bold are statistically indistinguishable from one another according to Kruskal-Wallis testing followed by Tukey pairwise comparison, both performed with significance of 0.05.

The first part of the configuration name is the upper-level algorithm used. For GA and DE, the following two values denote the resampling strategy and parameter configuration. The value following $F$ is the number of optimizations out of the 2000 optimizations (20 experiment replications × 100 post replications) that failed to reach the target performance within the maximum budget. Finally, $E$ is the total number of unique CPV tuples that was evaluated.

The results from the experiments of tuning DE on the Ackley test function are shown in Figure 3. From the results it can be seen that on this parameter tuning problem the configurations that favor replications over unique CPV tuples perform better than the configurations that do the opposite. The dynamic resampling strategies perform well with both tuning budgets, but especially so for the more limited budget of 5000. Only using 1 or 5 static replications is clearly not enough for this parameter tuning problem since all of those configurations have a high failure rate. However, even the best performing configurations have some failed runs. It can also be noted that good results can be obtained on this parameter tuning problem using only a few hundred unique evaluations.

The results from the experiments of tuning GA on the Ellipse test function are shown in Figure 5. The results from this parameter tuning problem are quite different than from the previous problem. The best configurations from these experiments are the ones that use fewer replications. For the smaller tuning budget of 5000 the best configurations use static 1 and 5 replications. More configurations perform well with the higher tuning budget, especially the dynamic resampling strategies and static 5 and 10. The results indicate that the evolutionary algorithms need at least 1000 unique evaluations to have a good chance of finding the optimal CPV tuple. On this particular parameter tuning problem there is little to no benefit of exceeding 5 static replications, and with the smaller
tuning budget, 1 optimization replication is even sufficient. Another observation is that the control parameters of the GA are important for this problem. Since, the configurations that use the exploration strategy significantly outperform the configurations that do not. The cause is that the standard parameters more often lead to a local optima. DE, however, is neither helped nor hindered by the exploration strategy.

The results from the experiments of tuning GA as a generalist are shown in Figure 8. The results from tuning GA as a generalist are similar, but not identical, to the results of tuning GA as a specialist. Configurations that favor evaluating more unique CPV tuples over more replications perform better. One difference from the specialist experiment is that one replication is not enough, since static 1 configurations have a significantly higher failure rate. This is true for both the smaller and larger tuning budgets. Another similarity is that the GA configurations with the exploration strategy perform better than those with only standard control parameters.

The results from the experiments of tuning DE as a generalist are shown in Figure 11. As with tuning DE as a specialist, the configurations that favor replications over testing new CPV tuples perform better. However, the performance of static 20 is worse compared to the specialist experiments, possibly because it only uses 500 unique CPV tuples.

VII. CONCLUSIONS

This paper presents a bilevel optimization approach to the parameter tuning problem. This approach is computationally expensive, since a complete optimization run has to be performed multiple times to evaluate a single set of control parameters. A framework and software architecture is therefore proposed that can utilize parallel computing resources and efficiently scale to hundreds of computing nodes with thousands of CPU cores. The components of the software architecture can easily be distributed as a standalone executable or embedded into another application, as well as deployed to hundreds of computers without other hardware or software requirements. The framework supports a variety of different aspects of parameter tuning, such as ability to tune both specialist and generalist CPV tuples, handling of both single- and multi-objective parameter tuning problems, and supporting both static and dynamic upper-level resampling strategies. The framework and software architecture is tested by performing millions of individual optimizations spanning many thousands of parameter tuning experiments.

The parameter tuning problem is difficult because of non-linear objective functions, interacting variables, multiple local optima and stochasticity of the optimizers. However, it is on precisely this kind of difficult optimization problems that evolutionary algorithms have been shown to be effective. Previous works have shown that evolutionary algorithms can
Fig. 5. The performance of each upper-level configuration when tuning GA on the Ellipse test function.

Fig. 6. Tuning budget of 5000

Fig. 7. Tuning budget of 20000

be used successfully to solve the parameter tuning problem. However, the computational complexity of the parameter tuning problem has often limited existing studies to a few control parameters at a time or a limited set of problems. In this work, two evolutionary algorithms, differential evolution and a genetic algorithm, are evaluated as parameter tuners using the above mentioned framework and software architecture. The importance of replications was investigated and it was found that different parameter tuning problems can require very different resampling strategies. However, a simple dynamic resampling strategy of using few replications at the beginning of the search and later increasing their number was shown to provide a good trade-off between the quantity of CPV tuples explored and the quality of CPV tuples obtained. When the GA was configured with commonly recommended control parameter values, it tended to converge to local optima. This problem could be overcome by having explicit exploration and exploitation configurations for the upper-level control parameters.

The performance of both DE and GA as upper-level tuners was compared against that of irace, which is a state-of-the-art parameter tuning method. Their performance was evaluated by tuning both specialist and generalist CPV tuples. The parameter tuning problems were chosen to provide different types of difficulties that can be encountered when solving parameter tuning problems. Specifically, the need to balance the trade-off between replications and testing new and unique CPV tuples. Both DE and GA were significantly better than irace on the parameter tuning problems tested. Overall, DE was found to be a better tuner than the GA considering all configurations that were tested. This shows that evolutionary algorithms can be competitive against other parameter tuning methods.

REFERENCES

Fig. 8. The performance of each upper-level configuration when tuning GA as a generalist.

Fig. 9. Tuning budget of 30,000 function evaluations to target.

Fig. 10. Tuning budget of 60,000 function evaluations to target.


Fig. 11. The performance of each upper-level configuration when tuning DE as a generalist.

Fig. 12. Tuning budget of 30,000

Fig. 13. Tuning budget of 60,000
REFERENCES


Wessing, Simon et al. (2010). “Parameter Tuning Boosts Performance of Variation Operators in Multiobjective Optimization.” In: Parallel Problem Solving from Nature,


PUBLICATIONS IN THE
DISSERTATION SERIES


Martin Andersson received his B.Sc in Computer Science (2006) and his M.Sc in Automation Engineering (2011) from the University of Skövde. He has been working as a research assistant since 2006 in the School of Engineering Science at the University of Skövde.

In his thesis, Martin explores how optimization can be used to improve and gain new knowledge of optimization algorithms. The proposed approach is to use evolutionary computing to find the optimal control parameters of optimization algorithms. These optimal control parameter values can reveal information about the inner workings of optimization algorithms. This approach for solving the parameter tuning problem is computationally demanding. Therefore, a framework and software architecture capable of utilizing parallel computing resources are presented. This system is used to show that evolutionary algorithms are competitive with other state-of-the-art parameter tuning methods and how they should be configured for improved performance.