Universitat de Lleida

Contributions to automatic configuration and selection for satisfiability

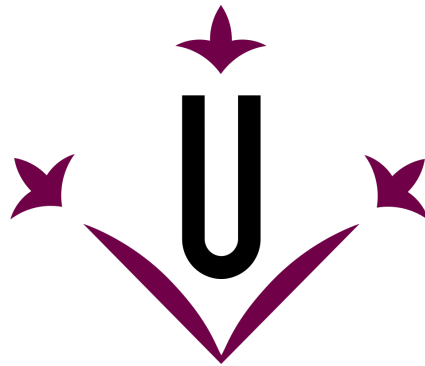
Josep Pon Farreny

http://hdl.handle.net/10803/673622

ADVERTIMENT. L'accés als continguts d'aquesta tesi doctoral i la seva utilització ha de respectar els drets de la persona autora. Pot ser utilitzada per a consulta o estudi personal, així com en activitats o materials d'investigació i docència en els termes establerts a l'art. 32 del Text Refós de la Llei de Propietat Intel·lectual (RDL 1/1996). Per altres utilitzacions es requereix l'autorització prèvia i expressa de la persona autora. En qualsevol cas, en la utilització dels seus continguts caldrà indicar de forma clara el nom i cognoms de la persona autora i el títol de la tesi doctoral. No s'autoritza la seva reproducció o altres formes d'explotació efectuades amb finalitats de lucre ni la seva comunicació pública des d'un lloc aliè al servei TDX. Tampoc s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant als continguts de la tesi com als seus resums i índexs.

ADVERTENCIA. El acceso a los contenidos de esta tesis doctoral y su utilización debe respetar los derechos de la persona autora. Puede ser utilizada para consulta o estudio personal, así como en actividades o materiales de investigación y docencia en los términos establecidos en el art. 32 del Texto Refundido de la Ley de Propiedad Intelectual (RDL 1/1996). Para otros usos se requiere la autorización previa y expresa de la persona autora. En cualquier caso, en la utilización de sus contenidos se deberá indicar de forma clara el nombre y apellidos de la persona autora y el título de la tesis doctoral. No se autoriza su reproducción u otras formas de explotación efectuadas con fines lucrativos ni su comunicación pública desde un sitio ajeno al servicio TDR. Tampoco se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al contenido de la tesis como a sus resúmenes e índices.

WARNING. Access to the contents of this doctoral thesis and its use must respect the rights of the author. It can be used for reference or private study, as well as research and learning activities or materials in the terms established by the 32nd article of the Spanish Consolidated Copyright Act (RDL 1/1996). Express and previous authorization of the author is required for any other uses. In any case, when using its content, full name of the author and title of the thesis must be clearly indicated. Reproduction or other forms of for profit use or public communication from outside TDX service is not allowed. Presentation of its content in a window or frame external to TDX (framing) is not authorized either. These rights affect both the content of the thesis and its abstracts and indexes.



Universitat de Lleida

LOG
Logic and Optimization Group

PhD Thesis

Contributions to automatic configuration and selection for satisfiability

Josep Pon Farreny

Thesis submitted in partial fulfilment of the degree Doctor of Philosophy
Doctoral program in Engineering and Information Technologies

Supervisor
Dr. Carlos Ansótegui Gil

2021

Acknowledgements

This work would not have been possible without the people who have accompanied me throughout the process. I want to start by thanking my supervisor, Carlos Ansótegui, his advice, creativity, and rigor, have shaped this work and laid the foundations for my development in the challenging task that is research. I also want to thank my colleagues at UdL, Eduard and Jesus, with them I spent many joyful moments that made the journey more pleasant.

I also want to express my gratitude for Dr. Meinolf Sellmann for the opportunity of visiting his research group at IBM. It was an enriching experience that helped me grow personally and professionally.

To my two dogs, Puc and Valentina, which offered me simple, predicable moments that served as a safe place to escape and ease my mind.

Finally, I would like to express my deepest gratitude to my close ones. My family that has been there any time for anything, without your support I would not have been able to pursue all my goals, and to my life partner, Sandra, for her unconditional support, guidance, love and for giving me the most precious thing in my life, my daughter Júlia.

Abstract

Within the computer science community, it is well known that algorithms may exhibit completely different behaviours depending on how their parameters are configured. This is even more obvious when the problems being tackled are NP-Complete. For example it has been proved experimentally that configuring solvers for the SATisfiabilty (SAT) problem, the Maximum SATisfiability (MaxSAT) problem or for Mixed Integer Programming (MIP), can result in improvements of orders of magnitude.

In this thesis, we show how automatically configured metahueristic algorithms can efficiently solve families of industrial and crafted instances of the MaxSAT problem. Then, we focus on improving the automatic algorithm configurator GGA providing a new distributed configurator which outperforms GGA on families of industrial and crafted SAT and MIP instances.

Finally, in our aim of making this technology more accessible for all research communities and industry we present the tool PyDGGA that implements our new configurator. We additionally introduce the OptiLog framework for rapid prototyping of SAT-based systems that also incorporates a module for automatic configuration for the easy application of several configuration tools.

Resum

En l'àrea de les ciències de la computació, és sabut que els algoritmes poden exhibir comportaments completament diferents depenent de com estiguin configurats els seus paràmetres, fet que s'aguditza quan els problemes que s'aborden són NP-Complets. Per exemple, s'ha demostrat experimentalment que configurar *solvers* pels problemes de la SATisfactibilitat (SAT), la Màxima SATisfactibilitat (MaxSAT) o la programació lineal d'enters mixta (MIP), pot resultar en millores d'ordres de magnitud.

En aquesta tesi, mostrem com algoritmes metaheurístics configurats de manera automàtica poden resoldre de forma eficient diverses families d'instancies industrials i artificails del problema MaxSAT. A continuació, ens centrem a millorar el configurador automàtic d'algoritmes GGA proporcionant un nou configurador distribuït que el supera en diverses famílies d'instancis industrials i artificials dels problemes SAT i MIP.

Finalment, per fer aquesta tecnologia més accessible a totes les comunitats científiques i la indústria, presentem l'eina PyDGGA que implementa el nostre nou configurador. A més, presentem el paquet OptiLog, el qual permet la creació ràpida de sistemes basats en SAT que addicionalment incorpora un mòdul de configuració automàtica per facilitar l'ús de diverses eines de configuració.

Resumen

En el área de las ciencas de la computación, es sabido que los algoritmos pueden exhibir comportamientos completamente diferentes dependiendo de como estén configurados sus parámetros, agudizándose este fenómeno cuando los problemas que se abordan son NP-Completos. Por ejemplo, se ha demostrado experimentalmente que configurar *solvers* para los problemas de la SATisfactibilidad (SAT), la Maxima SATisfactibilidad (MaxSAT) o la programación lineal de enteros mixta (MIP), puede resultar en mejoras de órdenes de magnitud.

En esta tesis, mostramos cómo algoritmos metaheurísticos configurados de manera automática pueden resolver de forma eficiente diversas familias de instancias industriales y artificiales del problema MaxSAT. A continuación, nos centramos en mejorar el configurador automático de algoritmos GGA proporcionando un nuevo configurador distribuido que lo supera en varias familias de instancias industriales y artificiales de los problemas SAT y MIP.

Finalmente, para hacer que esta tecnología sea más accesible para todas las comunidades científicas y la industria, presentamos la herramienta PyDGGA que implementa nuestro nuevo configurador. Además, presentamos el paquete OptiLog el cual permite la creación rápida de sistemas basados en SAT que adicionalmente incorpora un módulo de configuración automática para facilitar el uso de varias herramientas de configuración.

Contents

1 Introduction **1**
1.1 Objectives 2
1.2 Structure of this thesis 2
1.3 Publications and Awards 3

2 State-of-the-art **4**
2.1 The Satisfiability Problem 4
2.2 The Maximum Satisfiability Problem 5
2.3 The Integer Linear Programming Problem 5
2.4 The Automatic Algorithm Configuration Problem 5
2.5 The Automatic Algorithm Selection Problem 6
2.6 GGA: Gender-Based Genetic Automatic Algorithm Configuration . . 7
2.7 SMAC: Sequential Model-Based Algorithm Configuration 8
2.8 IRACE: Iterated Racing for Automatic Algorithm Configuration . . . 9
2.9 ISAC++: Improved Instance-Specific Algorithm Configuration 10

3 Reactive Dialectic Search Portfolios for MaxSAT **12**
3.1 Hyper-Parameterization 13
3.2 Dialectic Search 13
3.3 Reactive Dialectic Search Portfolios 15
3.4 Experimental Results 18
3.5 Conclusions 22

4 Hyper-Reactive Tabu Search for MaxSAT **23**
4.1 Hyper-Parameterized Reactive Tabu Search 24
4.1.1 Tabu Search Parameters 24
4.1.2 Dynamic Search Features 26
4.1.3 Hyper-Parameterization 27
4.2 Hyper-Reactive Tabu Search for MaxSAT 28
4.3 Experimental Results 28
4.4 Conclusions 35

5 Boosting Evolutionary Algorithm Configuration **36**
5.1 Improving Parallel Efficiency Using an Evolution Simulator 37
5.1.1 Experimental Results 40
5.2 Improving GGA 41

5.2.1 Instance Selection Strategy 41
5.2.2 Elite Mini-tournament 42
5.2.3 Experimental Results 42
5.3 Instance-Specific Parameter Selection 45
5.3.1 A Portfolio of All Parameterizations 46
5.3.2 A Portfolio of Selected Parameterizations 47
5.3.3 Selecting the Portfolio 47
5.3.4 Experimental Results 47
5.4 Conclusions 52

6 PyDGGA: Distributed GGA for Automatic Configuration 53
6.1 Distributed Architecture 53
6.2 Scheduling & Canceling 54
6.3 Tool Enhancements 55
6.4 Using PyDGGA 55
6.5 Experimental Results 57
6.6 Conclusions 57

7 OptiLog: A Framework for SAT-based Systems 58
7.1 OptiLog Framework Architecture 59
7.2 Formula Module 59
7.3 SAT Solver Module 60
7.4 PB Encoder Module 60
7.5 Automatic Configuration (AC) Module 61
7.6 Adding SAT Solvers to OptiLog through iSAT Interface 62
7.7 Using OptiLog 63
7.8 Experimental Results 65
7.9 Conclusions 65

8 Conclusion and Future Work 66

9 References 68

List of Figures

2.1 Visualization of automatic algorithm configuration. 6

3.1 Statistics recorded during search for two different HRDS parameterizations. 20

4.1 Normalized characteristics over the course of running three SCPNRE instances, with seconds on the x axis. 32
4.2 Normalized characteristics over the course of running the SCPNRG2 instance with two parameterizations, with seconds on the x axis. . . . 33

5.1 Evolution simulation graph 39
5.2 Timeline of evaluations in GGA-E and GGA 40
5.3 # of Evaluations and Executions on IBM. 44

6.1 Master-Worker architecture 54

7.1 OptiLog's architecture. 59

List of Tables

3.1 Average time to best bound and number of best upper bounds found for HRDS and CDS. 17
3.2 Average time and number of best upper bounds found on MS, PMS and WPMS crafted (top) and random (bottom) for WPM3, the various solvers developed in this chapter, and the top 3 solvers of each category from the MaxSAT Evaluation 2016. 18
3.3 MaxSAT evaluation 2016 winners, with the average solution time (sec.) and number of best upper bounds found. 22

4.1 Average score and number of instances solved for HRTS and SRTS on MSE'16 instance families after 300 seconds. We also give number of solved instances by local search solver ramp and the dsat/wpm3 portfolio from MSE'16. 30
4.2 Head-to-head comparison of maxroster and a maxroster+HRTS portfolio. 35

5.1 Number of generations after 2 days elapsed time. 41
5.2 Number of generations after 2 days elapsed time. 43
5.3 PAR10 performance (# solved instances). 43
5.4 PAR10 Performance (# solved instances). Def (Default). Solved (ratio of all solved instances). 48
5.5 Comparison with state-of-the-art, test performances. 51

6.1 PAR10 performance (# solved instances) on the test instances 57

Chapter 1

Introduction

Over the years, researchers and engineers have developed countless algorithms to address the vast array of problems that exist in computer science, and with the raise of *artificial intelligence* technologies we are more aware of the impact they have in our day-to-day. Nowadays, for many computational problems there are several viable algorithms with its strengths and weaknesses, and it is the task of computer scientist to find the ones that best match the domain-specific user requirements. To help the final users adapt the algorithm to their tasks, these expose several lower-level choices as *parameters*. By setting the parameters appropriately the algorithm may exhibit a completely different behaviour, changing its strengths and weaknesses to improve its efficiency for a particular task. A good example are constraint programming solvers, these solvers have to make several decisions while exploring the problem search space, many of which involve several heuristics. To put it in numbers, the SAT solver SparrowToRiss [1], that we use later in this thesis, exposes 222 parameters to the user. In this regard, algorithm configuration has emerged as an essential technology for the improvement of high-performance solvers.

Other examples of parameterized algorithms can be found in areas as diverse as sorting [2], linear algebra [3], numerical optimization [4], compiler optimization [5], parallel computing [6], computer vision [7], machine learning [8, 9], database query optimization [10], database server optimization [11], protein folding [12], formal verification [13], and even in areas far outside of computer science, such as water resource management [14].

Another area were algorithm configuration poses itself as an interesting research venue is *Metaheuristics*, which are procedures developed to provide general purpose approaches for solving hard combinatorial problems. These frameworks often serve as the starting point for the development of problem-specific search procedures. However, in practice, they very rarely work straight out of the box. An expert has to experiment with an approach an tweak parameters, remodel the problem, and adjust search concepts to achieve a reasonably effective approach.

In this thesis, we explore if automatic algorithm configuration is suited for metaheuristics by configuring *Dialectic Search* [15] and *Reactive Tabu Search* [16] for the incomplete MaxSAT problem. As well as, more problem oriented algorithms such as SparrowToRiss [1] for SAT problems and CEPLEX [17] for MIP problems.

Encouraged by the potential of automatic configuration algorithms we pursue a

more challenging goal which consist on further improving these sophisticated and already advanced algorithms. To this end, we introduce several modifications to GGA [18], an existing state-of-the-art algorithm for automatic configuration, what constitutes the main result of this thesis. In particular, we improve GGA's parallel performance, as well as the configuration process itself.

Last but not least, we take a step forward on making as accessible as possible to our research community, other fields and industry the algorithms we have designed and implemented. In this sense, within the Logic & Optimization Group (LOG) at UdL, we have developed the tool PyDGGA, that resulted of our attempt to improve GGA, and OptiLog, a Python framework for rapid prototyping of SAT-based systems that leverages the power of automatic configurators by allowing to easily create configuration scenarios including multiple solvers and encoders. The tool PyDGGA and Optilog framework have been already applied to several industrial projects where the LOG group is involved as well as several research papers not part of this thesis.

1.1 Objectives

The main objective of this thesis is contribute to the advancement of automatic algorithm configuration technology. In particular, we want to improve their performance when configuring constraint programming solvers, as we deem the synergy between these two technologies will play an important role in the development of future industrial solutions. In order to reach the main goal, we focus on the following objectives:

- Study and evaluate if automatic algorithm configuration is capable of configuring metaheuristic algorithms for the MaxSAT problem.
- Identify strength and weaknesses of a current state-of-the-art algorithm configurator in order to improve it. In particular we focus on the evolutionary algorithm GGA [18].
- Propose and evaluate different solutions to boost GGA's performance.
- Implement and make publicly available tools that let non-expert users integrate automatic algorithm configuration in their systems.

1.2 Structure of this thesis

The first chapter is devoted to the introduction of this thesis. Chapter 2 details the algorithm configuration problem, the problems tackled in this thesis using automatic configuration, and the state-of-the-art algorithms for automatic algorithm configuration. Chapters 3 and 4, evaluate automatic algorithm configuration applied to the metaheuristic algorithms: *Dialectic Search* [15] and *Reactive Tabu Search* [16]. In chapter 5 we present different improvements for GGA and prove its effectiveness experimentally. The next two chapters 6 and 7 are devoted to the two tools created

during the development of this thesis. Finally, in chapter 8 we conclude and present ways of extending the work done in this thesis.

1.3 Publications and Awards

The research conducted during the development of this PhD thesis has been presented in articles published or submitted to journals and conferences. In addition, the solutions developed were submitted to international competitions and won several awards. Next, we list our contributions and the achieved awards.

- C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, "Reactive dialectic search portfolios for maxsat," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA* (S. P. Singh and S. Markovitch, eds.), pp. 765–772, AAAI Press, 2017
- C. Ansótegui, B. Heymann, J. Pon, M. Sellmann, and K. Tierney, "Hyper-reactive tabu search for maxsat," in *Learning and Intelligent Optimization - 12th International Conference, LION 12, Kalamata, Greece, June 10-15, 2018, Revised Selected Papers* (R. Battiti, M. Brunato, I. S. Kotsireas, and P. M. Pardalos, eds.), vol. 11353 of *Lecture Notes in Computer Science*, pp. 309–325, Springer, 2018
- C. Ansotegui, J. Pon, and M. Sellmann, "Boosting evolutionary algorithm configuration," *Annals of Mathematics and Artificial Intelligence*, 2021
- C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, "PyDGGA: Distributed GGA for Automatic Configuration," in *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021*, Submitted for publication
- C. Ansótegui, J. Ojeda, A. Pacheco, J. Pon, J. M. Salvia, and E. Torres, "OptiLog: A Framework for SAT-based Systems," in *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021*, Submitted for publication

In addition to the articles, for our first contribution [19] we entered the MaxSAT evaluation 2016 [24], incomplete track, and won 10 medals: 4 gold, 2 silver and 4 bronze.

Chapter 2

State-of-the-art

To better understand the subjects explored in this PhD thesis, let us introduce the automatic algorithm configuration problem and the problems tackled by the algorithms being configured, Boolean Satisfiability (SAT), Boolean Maximum Satisfiability (MaxSAT) and Integer Linear Programming (ILP).

2.1 The Satisfiability Problem

The Boolean satisfiability (SAT) problem was the first problem proven to be NP-complete by Stephen A. Cook [25] and has since become the quintessentially NP-complete problem. It consists in determining if there is an interpretation that satisfies a Boolean formula. The following definition assumes the formula is in conjunctive normal form (CNF), which is the format used in the SAT competitions.

A Boolean formula is composed by *truth variables*, which are variables that either take *true* or *false* as their value. Such an assignment is called a *truth assignment*. A *literal* is a truth variable (positive literal) or its negation (negative literal). A positive literal is said to evaluate to true iff its variable is set to true, likewise a negative literal is said to evaluate to true iff its variable is set to false.

In a formula, literals are grouped in *clauses*. A *clause*, in CNF, is a disjunction of zero or more literals, and is said to be *satisfied* under a *truth assignment* or interpretation of its variables, if at least one of the literals in the clause evaluate to true. Otherwise the clause is said to be *falsified*.

Finally, A SAT *formula* in CNF, is a conjunction of zero or more clauses and is said to be *satisfiable* if there is at least one interpretation that satisfies all the clauses. Otherwise the formula is said to be *unsatisfiable*.

In CNF, the empty clause, denoted $\Box$, is equivalent to the identity of the disjunction, *false*. Thereby, the empty clause is always *falsified*. Following the same reasoning, the empty formula is equivalent to the identity of the conjunction, *true*. Thereby the empty formula is always *satisfied*.

2.2 The Maximum Satisfiability Problem

The Boolean Maximum Satisfiability (MaxSAT) problem is the optimization version of the SAT problem introduced before, and is further divided in (plain) MaxSAT, *Partial MaxSAT* (PMS) and *Weighted Partial MaxSAT* (WPMS).

MaxSAT is an important problem because several significant practical problems can be formulated naturally as MaxSAT problems. To name just a few, examples range from scheduling [26] to FPGA routing [27] to circuit design and debugging [28].

Following the definition of the *SAT* problem, in MaxSAT a clause is augmented with a *weight* to form a *weighted clause* (C, w), where C is a clause and w is a natural number or infinity, indicating the cost for fasifying clause C.

A WPMS formula is a set of weighted clauses $\varphi = \{(C_1, w_1), ..., (C_m, w_m), (C_{m+1}, \infty), ..., (C_{m+m'}, \infty)\}$ where the first m clauses are called *soft* clauses and the last m' clauses are called *hard* clauses. The objective of the *MaxSAT problem* is to find a truth assignment that satisfies all the hard clauses while minimizing the total cost of all the soft clauses that are falsified.

A PMS is a WPMS formula where the weights of all the soft clauses are equal. A *(plain) MaxSAT formula* is a PMS formula with no hard clauses. In both, PMS and (plain) MaxSAT, it is usual to set the weight of the soft clauses to 1, in fact, for CNF (plain) MaxSAT formulas the weight is omitted and assumed to have a value of 1 by the solvers.

2.3 The Integer Linear Programming Problem

Integer Linear Programming (ILP) consists of two parts: a cost function and a set J of constraints. Both parts involve a set $X = \{x_i\}$ of integer-valued variables, to form linear functions of the general form shown in 2.1 and 2.2 respectively.

$$C = \sum_i a_i x_i \quad \text{with } a_i \in \mathbb{R}, x_i \in \mathbb{Z} \tag{2.1}$$

$$\forall j \in J : \sum_i b_{i,j} x_i \geq c_j \quad \text{with } b_{i,j}, c_j \in \mathbb{R}, x_i \in \mathbb{Z} \tag{2.2}$$

The ILP problem is the problem of minimizing/maximizing the cost function, subject to a set of constraint functions. Notice that ILP is a variant of *Linear Programming* (LP), but the variables are restricted take *integer* values, this subtle restriction makes ILP an NP-complete problem. There is an extension of ILP were only some of the variables $x_i \in X$ are restricted to take on integer values, in such case the problem is called *Mixed Integer Programming* (MIP).

2.4 The Automatic Algorithm Configuration Problem

Given a target algorithm A with parameters $\{p_1, \ldots, p_n\}$ of domain $d(p_i)$. We define the parameter space Θ of A as the subset $d(p_1) \times \ldots \times d(p_n)$ of *valid* parameter com-

binations. Depending on the parameter, $d(p_i)$ can be categorical, a discrete domain of fixed values with no predefined order, or numerical, which represent integer or real values. Then, we define the *Automatic Algorithm Configuration* (AAC) problem as the optimization problem that consists on exploring Θ to find a configuration $\theta \in \Theta$ of A, which given a set of problem instances Π, minimizes a cost metric $\hat{c} : \Theta \times \Pi \rightarrow \mathbb{R}$, without exceeding a configuration budget B.

It is common for A to be a black-box, meaning it accepts some inputs (the parameters) and provides some output (e.g., $\hat{c}$), but we cannot see its internal functionality. This lets AAC generalize to any type of algorithm but makes it more difficult for algorithm tuners, as they cannot use A to infer additional information about Θ. Practically speaking, A is implemented as a binary file that outputs its results in a format adequate for its domain, but likely not for the AAC tool. Moreover, it may also be necessary to limit the resources that A can use to solve an instance, such as memory or CPU time. The standard way of addressing these issues in AAC tools is for the user to replace A with a wrapper script that handles these and any other aspects that may be necessary.

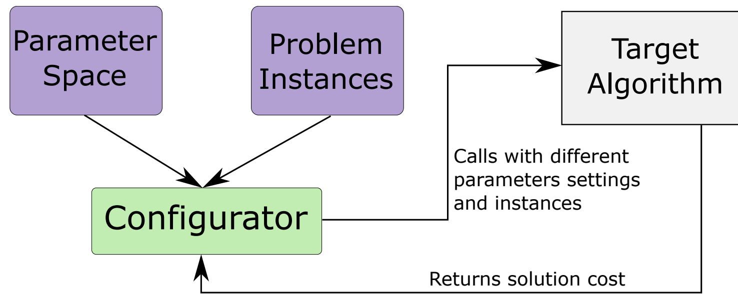


Figure 2.1: Visualization of automatic algorithm configuration.

2.5 The Automatic Algorithm Selection Problem

The algorithm selection problem was introduced in [29]. An can be defined as, given

- a set Π of instances
- a set of algorithms $\mathcal{A}$
- performance measure for each (instance, algorithm) pair $m : \Pi \times \mathcal{A} \rightarrow \mathbb{R}$

Find a mapping $s : \Pi \rightarrow \mathcal{A}$ that optimizes the expected performance measure $\sum_{i \in \Pi} m(i, s(i))$

In practice, the mapping i often implemented by using so-called instance features, i.e., numerical characterizations of the instances. These instance features are then mapped to an algorithm using machine learning techniques. However, the computation of instance features incurs additional costs, which have to be considered in the performance measure m.

2.6 GGA: Gender-Based Genetic Automatic Algorithm Configuration

Gender-Based Genetic Automatic Algorithm Configuration (GGA) is a genetic algorithm to search for a high-quality parameterization introduced in [18]. It was one of the first algorithms to support continuous parameters and introduced the concept of *gender* to apply different selection pressures to the individuals of the population.

Algorithm 1 GGA

Input: Target Algorithm A, Parameter Space Θ, Instances Π, Performance Metric $\hat{c}$, Configuration Budget B, # MiniTournaments N

1: **function** GGA($A, \Theta, \Pi, \hat{c}, N, B$)
2: pop $\leftarrow$ initPopulation(Θ)
3: $j = 0$
4: **while** B not exhausted **and** threshold not achieved **do**
5: $j = j + 1$
6: $\Pi_j \leftarrow$ selectInstances(Π, j)
7: $<w_1, ..., w_N> \leftarrow$ runMiniTournaments(A, pop.comp, Π_j, $\hat{c}$, pop.comp$/N$)
8: offspring $\leftarrow$ applyCrossoverAndMutate(pop.noncomp, $<w_1, ..., w_N>$, Θ)
9: pop $\leftarrow$ agingAndDeath(w_1, pop) $\cup$ offspring
10: **return** w_1

In particular, GGA (Alg.1) accepts a parameterized target algorithm A to be configured, a parameter space Θ, a training set of instances Π, a performance metric to be optimized $\hat{c}$ (time, accuracy, quality after a fixed timeout, etc), the number N of GGA mini-tournaments (we will explain shortly what those are), and a configuration time budget B.

Based on this information, GGA then runs its specialized genetic algorithm that executes the target algorithm with different parameters in order to find a good setting. The genome is defined by parameter settings and the fitness of each parameterization is defined as the algorithm performance as measured by the given metric on the training inputs. GGA partitions the population into a *competitive* and *non-competitive* group. The competitive group (*pop.comp*) is directly evaluated on the target algorithm, whereas the non-competitive group simply acts as a source of diversity. This allows GGA to use strong intensification procedures on one part of the population while remaining diverse enough to escape local optima. All crossover takes place between one competitive and one non-competitive individual.

To evaluate the genomes, a subset of the instances is randomly selected (Line 6) at each iteration of the master loop (Line 4), which we call a "generation". Then, for the first generation we select the first k (5 by default) instances and then grow linearly until roughly 75% of all generations have been carried out and the subset becomes Π. When tuning for runtime, this procedure is efficient, because the, usually rather bad, parameterizations at the beginning of the configuration process would otherwise require a disproportionate fraction of the total configuration time, without adding much value.

GGA uses a parallel racing scheme, a so-called "mini-tournament", in which a

set of $\frac{|\text{pop.comp}|}{N}$ parameterizations from the competitive sub-population are evaluated simultaneously (either by parallel execution on multiple cores or via threaded execution) on the selected inputs (Line 7). As soon as we have determined the winner of a mini-tournament, all pending evaluations of other participants in the mini-tournament are interrupted and terminated or removed from the queue. This *racing* saves significant amounts of CPU time that would otherwise be spent evaluating bad genomes.

The winners from each mini-tournament ($\{w_1, \ldots, w_N\}$) are the only competitive genomes that will parent new offspring in this generation (Line 8). An aging policy (Line 9) is used to prevent population growth, whereby the overall best competitive individual (referred as w_1) survives as long as it performs better than the other mini-tournament winners. For further details on GGA, we refer the reader to [18].

2.7 SMAC: Sequential Model-Based Algorithm Configuration

Sequential Model-Based Algorithm Configuration (SMAC) is based on Bayesian optimization [30]. The core motivation for using Bayesian optimization is that evaluations of a solver parameterization take a very significant amount of time: We need to run the parameterized solver on the training instances to get a new data point that quantifies the performance of the respective parameterization. This observation motivates spending a little more time for probing the parameterization space intelligently. In Bayesian optimization, we use few evaluations of the target solvers to train a so-called "surrogate model" that predicts the performance of the solver for a given parameterization. Then, we can use this fast-to-evaluate surrogate model to search for promising new configurations before we then tediously run a new candidate parameterization on the training instances.

Algorithm 2 SMAC

Input: Target Algorithm A, Parameter Space Θ, Instances Π, Performance Metric $\hat{c}$, Configuration Budget B

1: **function** SMAC($A, \Theta, \Pi, \hat{c}, B$)
2:     [R, θ_{inc}] $\leftarrow$ initialize(Θ, Π)
3:     **while** B not exhausted **do**
4:         [M, t_{fit}] $\leftarrow$ fitModel(R)
5:         [$\vec{\Theta}_{new}$, t_{select}] $\leftarrow$ selectConfigurations(M, θ_{inc}, Θ)
6:         [R, θ_{inc}] $\leftarrow$ intensify(A, $\vec{\Theta}_{new}$, θ_{inc}, R, Π, $\hat{c}$)
7:     **return** θ_{inc}

SMAC (Alg.2), first initializes a *best candidate* configuration θ_{inc} and the, possibly empty, history of conducted evaluations of different *(configuration, instance)* pairs R (Line 2). Then, until the configuration budget is exhausted, it fits a surrogate model M (Line 4), using the information available in R. Once M has been created, it uses it to select a new set of promising candidate configurations $\vec{\Theta}_{new}$

(Line 5). Finally, it evaluates $\vec{\Theta}_{new}$ and θ_{inc} on instances from Π to determine the next best candidate θ_{inc}, according to $\hat{c}$ (Line 6).

A key point in SMAC is the exploration/exploitation trade-off, handled by measuring how promising a candidate configuration, θ, is using the surrogate model's predictive distribution. To quantify it, they proposed a way to compute its *expected* positive *improvement* $EI(\theta)$ [31], which is large for θ with low predicted cost (*exploitation*) and for those with high predicted uncertainty (*exploration*). Equipped with this metric, they conduct a multi-start local search, from which they extract the configurations with locally maximal EI, and combine them with randomly-sampled configurations. Finally, they sort the resulting list of configurations in descending order of EI.

2.8 IRACE: Iterated Racing for Automatic Algorithm Configuration

Iterated Racing for Automatic Algorithm Configuration (IRACE) is an iterated racing procedure, which is an extension of the Iterated F-race configurator [32]. IRACE iteratively runs the following 3 steps until a termination criterion is met, as shown in Algorithm 3

1. Sample new configurations according to a particular distribution.
2. Evaluate the sampled configurations using a racing scheme.
3. Update the sampling distribution in order to bias the sampling towards the best configurations.

Algorithm 3 IRACE

Input: Target Algorithm A, Parameter Space Θ, Instances Π, Performance Metric $\hat{c}$, Configuration Budget B

1: **function** IRACE($A, \Theta, \Pi, \hat{c}, B$)
2: $\Theta_1 \leftarrow$ sampleUniform(Θ)
3: $\Theta^{elite} \leftarrow$ race($A, \Theta_1, \hat{c}, B$)
4: $j = 1$
5: **while** B not exhausted **do**
6: $j = j + 1$
7: $\Theta^{new} \leftarrow$ Sample(Θ, Θ^{elite})
8: $\Theta_j \leftarrow \Theta^{new} \cup \Theta^{elite}$
9: $\Theta^{elite} \leftarrow$ race(A, Θ_j, Π, $\hat{c}$, B)
10: **return** Θ^{elite}

In IRACE, each parameter has an associated sampling distribution, which is either a truncated normal distribution for numerical parameters, or a discrete distribution for categorical parameters. Normal distributions are updated by updating the mean and standard deviation, while discrete distributions are updated by changing the probability value of each element. Initially these distributions are uniformly

initialized and used to select the first configurations Θ_1 (Line 2), and then updated during the search and used to sample new configurations (Line 7).

Similar to the idea of increasingly selecting more instances in GGA, IRACE generates many configurations on the first iterations, where the sampling distributions are expected to generate configurations with very different performance results. Then, as more iterations are conducted, fewer and fewer configurations are generated. The idea is that at some point the distributions will start producing configurations with similar performance results, and more evaluations will be needed to discern which are better. Hence, it makes sense to generate less configurations and spend more time evaluating them.

During the racing procedure (Lines 3 & 4), after a minimum number of instances have been evaluated, IRACE runs a statistical test every time a new instance is evaluated on all the current configurations, Θ_j, as a selection heuristic to determine which configurations should be discarded. Once the number of remaining configurations is below a certain threshold the race stops and those configurations are the Θ^{elite} of that iteration. For the statistical tests, IRACE relies on the non-parametric Friedman's two-way analysis of variance by ranks or the paired *t-test*, both with a significance level of 0.05 by default.

2.9 ISAC++: Improved Instance-Specific Algorithm Configuration

ISAC++ is an improvement over the original ISAC [33]. It is based on the assumption that instance features are enough to determine which instances are similar in nature, and, consequently, by clustering them we should expected groups that behave similarly under the same algorithm and configuration.

Algorithm 4 ISAC++ Training

Require: Target Algorithm A, Parameter Space Θ, Instances Π, Instances Features $\Pi_{features}$, Performance Metric $\hat{c}$, Algorithm Tuner T

1: **function** TrainISAC++(A, Θ, Π, $\Pi_{features}$, $\hat{c}$, T)
2: $clusters \leftarrow$ computeClusters(Π, $\Pi_{features}$)
3: $configs \leftarrow$ []
4: $performances \leftarrow$ []
5: **for** $\Pi_c \in clusters$ **do**
6: $conf \leftarrow T(A, \Theta, \Pi_c, \hat{c})$
7: $configs$.append($conf$)
8: $perf \leftarrow evaluate(A, conf, \Pi, \hat{c})$
9: $performances$.append($perf$)
10: $portfolio \leftarrow$ CSHC(A, Π, $\Pi_{features}$, $performances$)
11: **return** $portfolio$

Therefore, when training (Alg.4), ISAC++ first clusters the training instances according to their features (Line 2). The number of clusters is determined automatically by the g-means [34] algorithm, which determines that a cluster can be split into two if the projection of the points onto the line, that traverses the two new centroids, does not follow a normal distribution.

Then, given a target algorithm A, it simply runs an instance-oblivious AAC (Line 6), such as GGA, to produce a configuration that improves the performance of A on each cluster. This means that ISAC++ will have to chose among as many configurations as clusters.

Each of the resulting configurations is then evaluated on the whole training set (Line 8), and, finally, results are used to train a cost-sensitive hierarchical clustering (CSHC)[35] algorithm selector (Line 10). At runtime (Alg.5), ISAC++ simply computes the features (Line 2) and queries CSHC (Line 3) to select the best configuration for the given instance.

Algorithm 5 ISAC++ Run

1: **function** RunISAC++(A, $portfolio$, i)
2: $i_{features} \leftarrow$ computeFeatures(i)
3: $conf \leftarrow$ selectAlgorithm($i_{features}$, $portfolio$)
4: **return** runAlgorithm(A, $conf$, i)

Chapter 3

Reactive Dialectic Search Portfolios for MaxSAT

The meta-algorithmics community has seen major advances in recent years. On the one hand, techniques for algorithm selection [36, 37] have led to major advances in our ability to solve great ranges of different types of instances in various domains [38, 39, 40, 41, 35, 42]. On the other hand, algorithm configurators have advanced from limited tuning approaches [43] to scalable, high-powered general methods [44, 18, 30, 45, 32]. Combining portfolios and automatic configuration has led to input-specific tuners [46, 33, 47] that not only choose superior parameterizations for a target algorithm, but also create new ones based on the input to be processed.

The methods above all focus on choices made before the actual target algorithm is run. Approaches that aim at modifying algorithm behavior a posteriori during the actual run have also been developed. The reactive tabu search (RTS) algorithm from [16] is the prototypical example. RTS modifies the length of the tabu list dynamically during search depending on how the search progresses. Another example is the Stage approach from [48], a heuristic local search method that analyzes search trajectories to construct predictive evaluation functions that are used during search to escape local minima. New theoretical results (e.g. by [49]) prove that dynamic updates during search can guarantee strictly better asymptotic performance. In addition, [50] argues that offline configuration can be used to create online control strategies.

In this chapter we combine automatic, input-specific algorithm configuration with reactive search to create a hyper-reactive search algorithm. We do so with the objective of complementing an existing MaxSAT solver with a portfolio of automatically generated search algorithms and thus achieve a more robust, state-of-the-art heuristic MaxSAT solver.

We first review the concept of hyper-parametrized algorithms, followed by an introduction to dialectic search and our new reactive metaheuristic. Finally, we apply it to MaxSAT and compare it empirically with state-of-the-art MaxSAT solvers as well as its non-reactive counterpart, both in combination with the existing solver and in isolation. In the end, we obtain a solver that outperforms the state of the art in various categories of heuristic MaxSAT solving, as assessed independently in the 2016 MaxSAT Evaluation [24].

3.1 Hyper-Parameterization

The goal of reducing the reliance on human experts when solving search problems has been the focus of a number of communities and works. We provide a brief overview of the literature in this area.

Hyper-heuristics [51, 52] are “heuristics to choose heuristics”. Given a combinatorial optimization problem to be solved, hyper-heuristics attempt to dynamically choose the correct heuristic to apply at any given time, potentially switching between multiple search paradigms (genetic algorithms, local search, etc.) during a single solution procedure. Reinforcement learning is used in [53] to adjust the search strategies and parameters for a hyper-heuristic based great deluge algorithm applied to a timetable examination problem. A further work in this area, [54], adjusts heuristic selections and adapts parameters dynamically with static, reactive strategies during search.

Doerr and Doerr show that the population size of an evolutionary algorithm can be optimally controlled for a generalized OneMAX problem [49, 55]. While this result it not for a “real-world” problem, it provides strong evidence that online adjustment strategies are much more effective than their static counterparts.

The idea of hyper-parameterizing local search was pioneered in [56] who invented a solver named SATenstein. SATenstein is an approach for tackling the SAT problem and can be instantiated to process like a number of successful local search approaches that have been developed for SAT earlier. That is, SATenstein’s hyper-parameters determine what search strategy is used. Therefore, SATenstein can be trained for specific families of SAT instances using a parameter tuner. In fact, using instance-specific tuning, a solver like SATenstein can even be configured to set its own hyper-parameters based on characteristics of the concrete instance to be solved [46, 33]. Essentially, different instantiations of the same solver then form an algorithm portfolio [37].

3.2 Dialectic Search

Dialectic search was introduced in [57], and its parameterized form is shown in Algorithm 6. The algorithm accepts the following parameters:

- f: The objective function to be optimized.
- g: The size of the greedy candidate set as percentage of all variables in the problem.
- a_l, a_u: A lower and upper bound on the percentage of variables to be changed to construct an antithesis. The exact size of the change is then chosen uniformly at random in the interval given whenever a new antithesis is generated.
- p_a: The probability of greedily improving the antithesis.
- p_r: The probability of restarting the search.
- r_l, r_u: A lower and upper bound on the percentage of variables to be changed to construct a new starting point. The exact size of the change is then chosen uniformly at random in the interval for each restart.

Algorithm 6 Parameterized Dialectic Search

```
1: function DIALECTIC-SEARCH (f, g, a_l, a_u, p_a, p_r, r_l, r_u)
2:     INIT(thes)
3:     best ← thes ← GREEDY(thes, g)
4:     while not timeout do
5:         while true do
6:             anti ← MODIFY(thes, a_l, a_u)
7:             if ANTIGREEDY(p_a) then
8:                 anti ← GREEDY(anti, g)
9:             syn ← GREEDY(MERGE(thes, anti), g)
10:            if f(syn) < f(best) then best ← syn
11:            if f(syn) < f(thes) then
12:                thes ← syn
13:            else if RESTART(p_r) then break
14:        thes ← GREEDY(MODIFY(thes, r_l, r_u), g)
15:    return best
```

The algorithm starts with an initial assignment to all variables, and aims to improve this assignment in a greedy search, making the best individual variable change in each step until any such change would lead to a worsening of the objective function. Depending on the problem addressed, the greedy search itself may take significant time. In dialectic search, we therefore randomly limit the search for a best variable to only alter a set of candidate variables that are randomly selected in each greedy step. If none of these can improve the objective, the greedy algorithm halts, otherwise we make the best change of a candidate variable.

The resulting assignment is called the "thesis." Dialectic search now alters parts of the thesis, thus generating a so-called "antithesis" (function *Modify*). The antithesis may be improved by a greedy search itself. Thus equipped with a thesis and an antithesis, dialectic search then aims to generate a "synthesis" by searching the space between thesis and antithesis (function *Merge*).

The synthesis is constructed by searching the area between the thesis and antithesis, for example by a nested local search [58] or, our choice for this work, by path relinking [59]. Starting from the thesis, we greedily select the best (in the sense that it most favorably affects the objective) variable among those where the current assignment still differs from the antithesis. We then set this variable to the value it takes in the antithesis. We repeat this until we arrive at the antithesis. The synthesis is then the best assignment we encountered. Then, we greedily improve the synthesis. If the resulting synthesis improves the current thesis, it becomes the new thesis. If the synthesis is worse than the thesis, we restart with a given probability at a new starting point that obtained by applying some random modification to the current thesis. If we do not restart, we choose a new antithesis. One such iteration is called a "move." Each change in a variable, either within a greedy search or when traversing from thesis to antithesis, is called a "step."

There are a number of decisions that dialectic search must make: What is the best size for the candidate set in the greedy search (parameter g)? How many

variables should be changed to generate the antithesis (parameters a_l, a_u)? With what probability should we greedily improve the antithesis (parameter p_a)? With what probability should we restart the search (parameter p_r)? When a restart is triggered, how many variables should be changed to new random values to generate a new starting point (parameters r_l, r_u)?

Note that the parameters could be set to realize a wide variety of search algorithms, from an iterated local search with an outer random walk over nested greedy searches ($p_r = 1$, $a_l = a_u = 0$, $g = 1$) to iterated path relinking between the currently best known solution and randomly generated local optima ($p_r = 0$, $g = 1$, $a_l = a_u = 0.5$, $p_a = 1$).

We could simply employ meta-algorithmics technology to automatically and, possibly input-specifically, tune our metaheuristic solver for any application. As such, our starting point is similar to the SATenstein solver that was developed for SAT [56].

In the following, we go one step further by making all dialectic search parameters *reactive*. That is, we do not simply build a portfolio of different dialectic searches for MaxSAT (and thus build some sort of MaxSATenstein). Instead, we allow the metaheuristic to *change its characteristics dynamically during search* based on the search progression.

3.3 Reactive Dialectic Search Portfolios

To set the parameters that guide the search, we will track the progress of the time-limited search as it is unfolding. In particular, we propose to track the following eleven values. We emphasize that other properties could be tracked in addition to these ones:

1. Time elapsed as percentage of total time before timeout.
2. Number of restarts conducted as a percentage of total restarts expected to be completed within the time limit.
3. Number of moves as a percentage of the total moves expected to be completed within the time limit.
4. Number of steps as a percentage of the total steps expected to be completed within the time limit.
5. Total number of improving syntheses found over the total number of dialectic moves expected to be completed within the time limit.
6. Number of moves in the current restart over the total number of dialectic moves expected to be completed within the time limit.
7. Number of moves since the current best known solution was found over the total number of dialectic moves expected to be completed within the time limit.
8. Number of moves since the last thesis update in the current restart over the total number of dialectic moves expected to be completed within the time limit.
9. Number of steps in the current restart over the total number of steps expected to be completed within the time limit.

10. Number of steps since the current best known solution was found over the total number of steps expected to be completed within the time limit.
11. Number of steps since the last thesis update in the current restart over the total number of steps expected to be completed within the time limit.

The objective is now to find a way to make the search use these values to set the seven parameters that guide the search dynamically. Note that all parameters are values between 0 and 1, either because they represent probabilities or percentages of the total number of variables in the given problem instance. Naming the values above $v_1, \ldots, v_{11}$, we set

$$p_k = \frac{1}{1 + e^{(w_0^k + \sum_i v_i w_i^k)}}$$

for each dialectic search parameter p_k, $k = 1 \ldots 7$.

We have thus transformed the configurable dialectic search with seven static parameters into a *hyper-configurable reactive dialectic search (HRDS)* with 84 (7 times 12) meta-parameters.

An adequate interpretation of this approach is that the dependence of the dialectic search parameters from the statistics of the unfolding search is determined by a *logistic regression* (in the original sense, not its common application to classification). The obvious challenge now is to *learn* the meta-parameters w_i^k in a way that will lead to good search performance.

Lacking any other supervision than the total search performance, we employ ISAC++ [47] for this task. We first cluster training inputs, then run GGA++ [45] on each cluster, and finally build a portfolio of the parameterizations found using cost-sensitive hierarchical clustering (CSHC) [35].

It is noteworthy that, until now, nothing in our approach has been MaxSAT specific. That is, in principle we can employ the reactive dialectic search approach outlined above on any combinatorial search problem. Additionally, the idea can be adapted to any reactive search metaheuristic. To use it for time-limited local search for MaxSAT, however, we need to make some decisions.

Evaluating Parameterization Performance:

We first need to set a metric to ascertain when a parameterization is better than another. For each MaxSAT training instance, we record the best known-solution from prior experience. When HRDS finds a truth assignment with that quality, we will consider the instance *solved* for training purposes and stop the dialectic search run.

In the beginning of our tuning, all parameterizations may be so bad that, within our training time limit, none can find a solution with that best known cost. Therefore, all parameterizations time out and all we can compare is how well they did relative to the best known solution within the time limit. Later, when some parameterizations get some instances to solve to the best known solution quality before the time limit, we can count the number of instances the parameterization is able to *solve* in this way, and the one with the highest count wins. Finally, towards the

end of the tuning, we will hopefully have high quality parameterizations that can *solve* all instances. In this case, we can consider the average time it took to solve the instances to determine the winning parameterization.

Algorithm 7 Parameterization Comparison Function

1: **function** SELECTWINNER (p_1, p_2)
2: **if** $\textit{num-solved-bk}(p_1) \neq \textit{num-solved-bk}(p_2)$ **then**
3: **return** $\text{argmax}_{p \in \{p1,p2\}}(\textit{num-solved-bk}(p))$
4: **if** $\textit{num-finished}(p_1) \neq \textit{num-finished}(p_2)$ **then**
5: **return** $\text{argmax}_{p \in \{p1,p2\}}(\textit{num-finished}(p))$
6: $G_1 \leftarrow \textit{sort-by-gap}(\textit{results}(p_1))$
7: $G_2 \leftarrow \textit{sort-by-gap}(\textit{results}(p_2))$
8: **for** $i = 1 \ldots |G_1|$ **do**
9: **if** $G_1[i] < G_2[i]$ **then return** p_1
10: **else if** $G_1[i] > G_2[i]$ **then return** p_2
11: **return** $\text{argmin}_{p \in \{p_1,p_2\}}(\textit{avg-cpu-time}(p))$

Category	Solver	MS		PMS		WPMS	
		Time	#	Time	#	Time	#
Crafted	HRDS	**0.80**	79	18.73	**48**	41.68	**24**
	CDS	6.44	79	24.79	45	17.40	21
Random	HRDS	9.67	**82**	73.44	**37**	**2.29**	99
	CDS	3.73	76	25.41	24	2.64	99

Table 3.1: Average time to best bound and number of best upper bounds found for HRDS and CDS.

Algorithm 7 shows the comparison function that guides how we determine top-performing parameterizations within GGA++ accordingly. Rather than giving each parameterization a numeric score (such as a penalized runtime, e.g., as done in many prior applications of algorithm tuners such as [60]) at the end of each tournament, we have defined a comparison algorithm that allows us to sort the parameterizations and to determine the winners.

The first criterion is which parameterization solves more instances to the best known quality within the time limit. If these are the same (for example because neither p_1 nor p_2 can *solve* any instances), then the second criterion is to compare the number of runs that finished correctly, i.e., where there were no problems with memory etc. We next compare which parameterization is closer to getting one more instance solved to best-known quality by considering the quality gap to best-known solutions. Finally, if all these criteria do not lead to a winner (for example because both p_1 and p_2 *solve* all instances) we return the parameterization that needs lower average runtime, with ties broken randomly.

We devised this method so as to give the GGA++ tuner a responsive objective function that guides the tuning search effectively no matter how well the current pool of parameterizations currently performs. However, this leaves us with a problem for the surrogate model used to genetically engineer the offspring within GGA++.

Solver	Time	#
WPM3	17.73	5
CDS	6.44	79
HRDS	0.80	79
CDS/WPM3	6.56	79
HRDS/WPM3	0.87	79
CCLS	**5.69**	**81**
CnC-LS	2.49	80
CCEHC	3.42	80

(a) MS Crafted (81 instances)

Solver	Time	#
WPM3	17.30	106
CDS	24.79	45
HRDS	18.73	48
CDS/WPM3	18.69	114
HRDS/WPM3	**18.52**	**116**
WPM3-2015-in	15.93	107
Optiriss6	37.85	99
Dist	6.97	81

(b) PMS Crafted (136 instances)

Solver	Time	#
WPM3	26.61	34
CDS	17.40	21
HRDS	41.68	24
CDS/WPM3	19.22	44
HRDS/WPM3	**23.20**	**46**
CCEHC	18.54	39
Ramp	12.18	29
SC2016	4.13	27

(c) WPMS Crafted (65 instances)

Solver	Time	#
WPM3	0.00	0
CDS	3.73	76
HRDS	9.67	82
CDS/WPM3	3.74	76
HRDS/WPM3	11.87	83
CnC-LS	**2.05**	**89**
borealis	2.28	89
SC2016	2.37	89

(d) MS Random (89 instances)

Solver	Time	#
WPM3	34.75	8
CDS	25.41	24
HRDS	73.44	37
CDS/WPM3	12.09	25
HRDS/WPM3	53.49	34
Dist-r	**2.07**	**42**
SC2016	2.55	42
CCLS	3.00	42

(e) PMS Random (42 instances)

Solver	Time	#
WPM3	91.72	1
CDS	2.64	99
HRDS	**2.29**	**99**
CDS/WPM3	2.67	99
HRDS/WPM3	2.56	99
SC2016	2.53	99
Ramp	4.22	99
CCLS	4.46	99

(f) WPMS Random (99 instances)

Table 3.2: Average time and number of best upper bounds found on MS, PMS and WPMS crafted (top) and random (bottom) for WPM3, the various solvers developed in this chapter, and the top 3 solvers of each category from the MaxSAT Evaluation 2016.

Namely, the surrogate model needs to predict in what regions of the parameter space we may expect superior parameterizations to be found. We solved this problem by using relative ranks rather than absolute performance when training the surrogate.

Evaluating Truth Assignments:

HRDS solves the MaxSAT problem using an (incremental) evaluation of a truth assignment. We simply maintain make-profits and break-costs (the weighted variants of make-counts and break-counts [61]) for each variable to quickly compute the effect on the objective when flipping a variable's truth assignment.

Characterizing MaxSAT Instances:

For instance-specific configuration we need features to characterize the inputs. We use the features proposed in [47] for this purpose.

3.4 Experimental Results

Having developed our approach in the previous section, we now evaluate it empirically. We run all our experiments on a cluster featured with Intel Xeon CPU E5-26020 @ 2.6GHz processors, a memory limit of 3.5 GB, and each machine runs an instance of Rocks Cluster 6.5 (Linux 2.6.32), which is the exact same environment used in the MSE16.

3.4.1 Benchmark Set

Recall that we eventually intend to complement a solver that was already designed for industrial instances, WPM3 [62]. As this solver already achieves state-of-the-art performance on industrial MaxSAT instances, we focus the training of our hyper-configurable dialectic search approach on randomly generated instances (Random category) as well as instances that are derived as encodings of other problems (Crafted category). Our base set of MaxSAT instances are all instances in the Random and Crafted categories in the MaxSAT Evaluation 2016 (MSE16) [24].

The three variants of the MaxSAT problem divide the base set further: (plain) MaxSAT (MS), Partial MaxSAT (PMS), and Weighted Partial MaxSAT (WPMS). In each category, instances are grouped into families: 3 families for MS crafted, 2 for MS random, 11 for PMS crafted, 4 for PMS random, 11 for WPMS crafted and 3 for WPMS random. We cleanly split each group randomly 80 to 20, whereby the 80% are assigned to our training set while the remaining 20% are set aside for testing.

3.4.2 ISAC++ Setup

We perform algorithm configuration exclusively on the instances marked for training for each group of instances within the MSE16 dataset that have more than 15 training instances left after the 80/20 split. We use a distributed version of GGA++ with 8 machines with 8 cores each, a population size of 100 individuals and 100 generations, using a 30 second target algorithm timeout. The time limit for the test instances is as in the MSE16, 300 seconds.

3.4.3 Competitors

We compare the following algorithms. HRDS is the new hyper-reactive dialectic search. CDS is an ISAC++ generated portfolio of the statically parameterized dialectic search with seven parameters. HRDS/WPM3 is a portfolio built from HRDS parameterizations plus ISAC++-tuned parameterizations of WPM3. CDS/WPM3 is a portfolio built from CDS parameterizations plus ISAC++-tuned parameterizations of WPM3.

3.4.4 Reactive vs. Non-Reactive Dialectic Search

Our first inquiry is to find out whether making dialectic search hyper-configurable is at all beneficial. The hyper-configuration space includes all static parameterizations, which are obtained by setting all 11 weights corresponding to search statistics to zero and setting the constant weight for each of the seven parameters to the right value. This means that the best parameterization for HRDS will always be at least as good as that of CDS. However, there is no guarantee that the ISAC++ tuner is able to find that parameterization, nor that the best parameterization thus found for the training set also generalizes well to the test instances.

Table 3.1 shows that our doubts are unfounded. In all categories we tested, HRDS outperforms CDS. The difference in performance is particularly noticeable in

two categories. First, on random PMS instances where the reactive dialectic search finds 37 best upper bounds compared to only 24 for the non-reactive counterpart of the otherwise identical approach. Second, on random MS instances HRDS solves 82 instances compared to 76 for CDS. This shows that the algorithm configurator can tune the hyper-heuristic effectively.

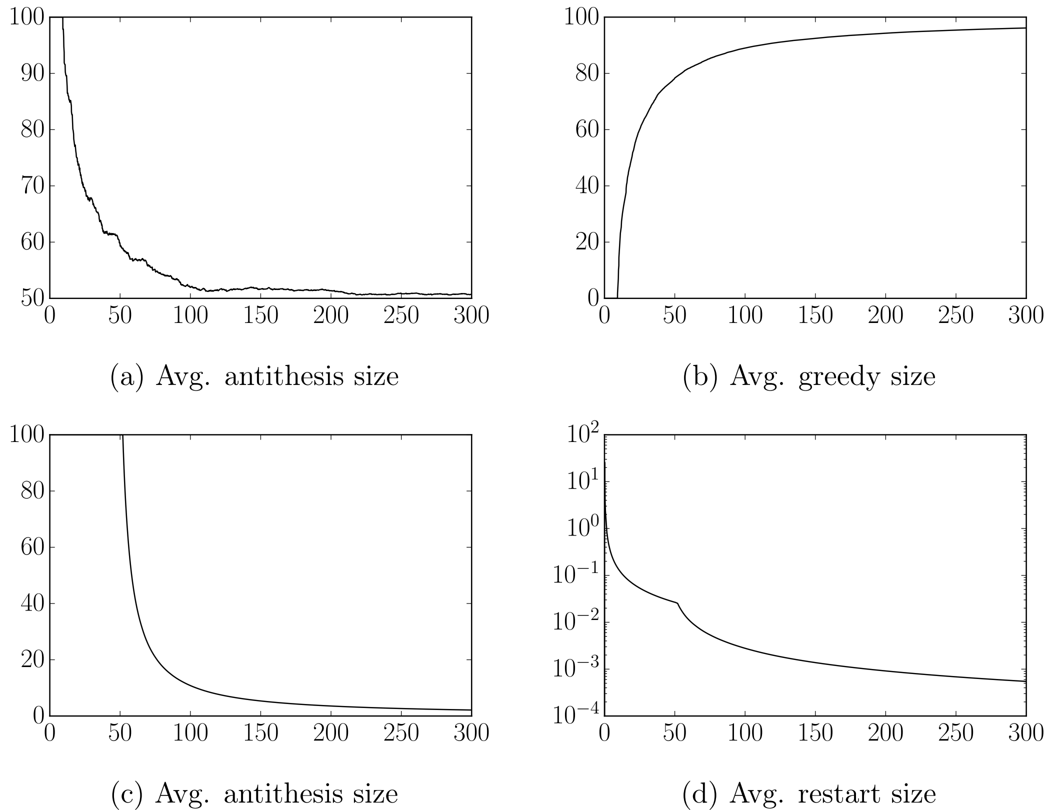


Figure 3.1: Statistics recorded during search for two different HRDS parameterizations.

Figure 3.1 sheds light on the inner workings of HRDS. On top, we track the effective running averages of the antithesis size on the left and the greedy candidate set size on the right for one parameterization, run on a PMS random instance with a time limit of 300 sec. Not shown here is that this parameterization holds the restart probability at zero and anti-greedy probability steady at 1. In this parameterization, the dialectic search thus never restarts, and it always conducts an antithesis greedy improvement, while during the search decreasing the level of difference between thesis and antithesis and simultaneously increasing the greedy candidate set size.

Contrast this with another parameterization, shown at the bottom. This one holds the restart probability and greedy candidate size firmly at 100% (not shown here). In the beginning, the antithesis size is 100%, which means there is a full greedy search started on the exact opposite pole from the current thesis and a path-relink conducted between the thesis and the result of that greedy run. At the same time, the restart size is somewhere between 20 and 50%, leading to an almost entirely new random starting point with only a slight bias towards keeping parts of the current

thesis intact in the antithesis. During the course of the optimization, the restart and antithesis sizes are then reduced further and further, making the search stay closer and closer to the current thesis. This behavior, moving from vast exploration in the beginning to more and more conservative moves in the end, is familiar from the simulated annealing metaheuristic. Yet this HRDS parameterization was not invented by a human, but found as an effective method for solving some MaxSAT instances by an algorithm configurator.

3.4.5 Random Weighted-Partial MaxSAT

The next question of interest is whether a solver that consists of reactive dialectic search parameterizations, a solver that is based on a local search metaheuristic that lacks any domain knowledge and only has access to an incremental evaluator of the target objective function, a solver that was programmed in only two working days and was then left to be tuned automatically, whether such a solver could outperform solvers that have been devised by teams of humans that have often developed and evolved their programs for years.

Table 3.2-(f) gives the answer: on random WPMS instances, HRDS outperforms all competitors from the MSE16. While multiple solvers manage to solve 99 test instances within the time limit, it is remarkable to see that the automatically generated solver can even outperform SC2016, which itself works significantly better than all other competitors.

Augmenting an Industrial Solver:

We finally arrive at our original objective, namely to make WPM3, a solver designed for industrial MaxSAT instances, a serious contender in other categories as well. Table 3.2 shows how WPM3 by itself compares with HRDS/WPM3 and CDS/WPM3 on all six random and crafted categories from the MSE16.

We observe that for some categories WPM3 is better than HRDS or CDS, while for others it is the other way around. However, joining WPM3 with HRDS or CDS parameterizations creates an algorithm portfolio that often exceeds the performance of the best choice for each category, and otherwise trails the best performance only slightly. In summary, augmenting a human-developed industrial MaxSAT solver with a machine-trained solver results in a much more robust and across-the-board applicable MaxSAT solver.

2016 MaxSAT Evaluation:

Using the methodology above, we entered the MSE16 with an earlier version of HRDS/WPM3 (DSAT-WPM3 at MSE16) with less HDRS parameterizations. In Table 3.3 we present the official results. Our two submissions (with one version, *-s, using a static schedule in the portfolio) won gold medals in 4 out of 9 categories: WPMS crafted and random, and PMS crafted and industrial. Overall, the two entrants won 10 medals out of 18 possible.

		1st		2nd		3rd	
Random	MaxSAT	borealis		SC2016		Swcca-ms	
		2.29	454	2.30	454	2.40	454
	Partial	Dist-r		CCEHC		SC2016	
		2.08	209	3.86	209	2.46	208
	W. Partial	**HRDS/WPM3-s**		SC2016		**HRDS/WPM3**	
		26.04	501	2.60	500	9.12	500
Crafted	MaxSAT	CCLS		CCEHC		SC2016	
		3.56	402	3.49	399	2.62	398
	Partial	**HRDS/WPM3**		WPM3-2015-in		**HRDS/WPM3-s**	
		20.26	575	13.33	539	17.99	522
	W. Partial	**HRDS/WPM3**		CCEHC		**HRDS/WPM3-s**	
		28.15	204	19.53	192	29.53	183
Industrial	MaxSAT	CnC-LS		**HRDS/WPM3**		WPM3-2015-in	
		49.00	47	40.84	40	22.57	38
	Partial	**HRDS/WPM3**		WPM3-2015-in		Optiriss6-in	
		26.70	513	22.29	505	37.57	433
	W. Partial	WPM3-2015-in		**HRDS/WPM3**		**HRDS/WPM3-s**	
		17.80	405	22.98	402	18.37	339

Table 3.3: MaxSAT evaluation 2016 winners, with the average solution time (sec.) and number of best upper bounds found.

3.5 Conclusions

We introduced hyper-configurable reactive dialectic search portfolios and applied them to MaxSAT. Moreover, we demonstrated that reactive search methods can be tuned effectively and outperform static instance-specific configuration in practice. By itself, the new method was able to define a new state-of-the-art for the random WPMS category, despite its general ignorance regarding the problem it is solving. Used to automatically complement an existing industrial MaxSAT solver, it defined a new approach that works robustly for random, crafted, and industrial instances. The approach was independently evaluated and compared with state-of-the-art MaxSAT solvers at the 2016 MaxSAT Evaluation where it won 4 out of 9 possible gold medals.

Chapter 4

Hyper-Reactive Tabu Search for MaxSAT

Historically, metaheuristics that did not require tuning parameters have been favored in the literature, as this should liberate the user from tediously searching for an instantiation of the metaheuristic that leads to good practical performance for the problem at hand.

To provide good performance even without many parameters, reactive search approaches (see [63]) have been developed that use built-in strategies to automatically and dynamically adjust key parameters during the search. For example, the canonical reactive search algorithm, reactive tabu search (RTS) [16], adapts the central parameter of tabu search, namely the length of the tabu list. RTS increases the tabu list length when cycles that occur in the search are shorter than a fixed threshold. The list length is decreased when all potential moves are tabu as well as when the number of steps taken since the last list length adjustment grows beyond the moving average of recent cycle lengths.

Hyper-reactive search replaces the static parameter adjustment strategies of reactive search with dynamic strategies that are tuned through an offline learning process. Building on this idea, we proposed a hyper-parameterization of dialectic search [15] in Chapter 3 (published in [19]), where hyper-parameters are used to determine how the standard parameters of dialectic search are adjusted *dynamically* based on runtime statistics that characterize how the search is progressing. The resulting hyper-reactive dialectic search (HRDS) algorithm is shown to outperform state-of-the-art MaxSAT solvers. In the following, we present how this idea can be realized for the Tabu Search metaheuristic.

Given the success of RTS in the literature, a key question is whether it can be transformed into a hyper-reactive approach. In this chapter, we pursue three objectives.

- First, we devise an approach that self-adjusts even more tabu search parameters than RTS.
- Second, we open the way the approach adjusts its parameters to the outside so that the overall approach can be customized and tailored for the problem at hand using a standard parameter tuner.

- Finally, we use the resulting, generally applicable approach to tackle the MaxSAT problem and demonstrate high quality performance.

In the following, we first devise the hyper-parameterized reactive tabu search approach that self-adjusts all key tabu search parameters based on dynamic search statistics. We then show how this new approach can be used to tackle the MaxSAT problem. Finally, we report on our experimental results on the MaxSAT problem.

4.1 Hyper-Parameterized Reactive Tabu Search

We now describe a hyper-parameterized tabu search that is based on RTS [16].

4.1.1 Tabu Search Parameters

We assume the reader is familiar with tabu search in general. For an introduction, we refer to [64] and [65]. The key parameters of the general tabu search metaheuristic are the following:

1. Length of the tabu list(s)
2. Escape rules
3. Neighborhood size
4. Aspiration criterion

The tabu search framework we develop here is calibrated for binary search problems, meaning unconstrained optimization problems where all variables take two values (0/1 or true/false). In this context, let us consider the above parameters in detail.

Length of Tabu Lists:

We use two lists, one that prevents recent solutions from being revisited, and another that keeps track of recently flipped variables that are kept from flipping again too quickly (unless an aspiration criterion allows it anyway). The length of the lists is governed by four dynamically self-updating continuous parameters in $[0, 1]$; two for each list.

After every tenth local search step, for each list, we consider the difference between both parameters. If the difference is positive, the list size is considered to be increased, otherwise it is considered to be decreased, whereby the absolute value of the difference determines the probability that a change in list length occurs. If a change occurs, the list of previously visited solutions grows or decreases by a factor of 1.01. The list of recently flipped variables increases or decreases by one one thousandth of the total number of variables.

As an aside: For the list of previously visited solutions we actually only record the variables that have been flipped. Then, to determine which variables are not allowed to be flipped in the current step, we simply traverse backwards through the list and add or remove the variables to/from a set. Every time the set has cardinality

one, the remaining variable in the set cannot be flipped as otherwise a previously visited solution recurs.

Consider this example: The variables most recently flipped (most recent first) had indices $[1, 2, 3, 4, 3, 2, 1, 5, 4]$. We first add variable 1 to the set: $\{1\}$. This set has cardinality 1, which means that the variable in the set may not be flipped (naturally, as that would directly undo the most recent move and thus return us directly to a recently visited solution). Next we add 2, 3, and 4. The set now contains $\{1, 2, 3, 4\}$. Next we consider variable 3, but since it is already in the set, we remove it: $\{1, 2, 4\}$. After also removing 2 and 1, we get: $\{4\}$. This set now has cardinality 1, which means that we cannot flip variable 4 either or we will return to the solution visited 7 steps earlier. We then add 5 and remove 4, leading to the final set of cardinality 1: $\{5\}$. Consequently, at the current step we may neither flip variables 1, 4 or 5 if we want to avoid revisiting a solution that already occurred within the most recent 9 steps.

Escape Rules:

Four dynamically self-updating parameters govern the escape behavior of the tabu search. After every tabu step, the first parameter, continuous in $[0, 1]$, determines the probability of an escape move. If an escape move is initiated, the next two parameters describe the minimum and maximum number of variables to be flipped. The concrete number of variables to be randomly selected and flipped is chosen uniformly at random in the given interval.

The last parameter $p \in [0, 1]$ then determines whether the new point is accepted as a new starting point for a new series of regular tabu search steps. To determine acceptance, we maintain a hash table of all previously visited solutions. The new potential starting point is hashed using a universal hash function and we then consider how many previously visited solutions r hash to the same value.

- If this hash value has seen more than the average μ solutions compared to the other values, the new point is rejected and we repeat the process of building a new starting solution *starting from the original solution where the tabu search last stopped.*
- If the new solution falls into a bucket that has seen less than the average number of solutions μ per bucket, we check if it is even less than $(1 + p)$ standard deviations (σ) below the average. If so, the new point is accepted as our new starting point.
- If the point hashes to a value that has been hashed to q times before, and q falls somewhere in the interval $[\mu - (1 + p)\sigma, \mu]$, then we consider the ratio $r \leftarrow \frac{\mu - q}{(1+p)\sigma} \in [0, 1]$. The new point is then be accepted as new starting point with probability r.

Neighborhood Size:

Especially when tackling problems with many variables, we may not want to consider all variables for flipping to determine which move would result in the best neighboring solution. We therefore introduce another continuous, dynamically self-updating parameter in $[0, 1]$ that determines the percentage of variables that are considered for flipping. If none of these variables yields a non-tabu solution, we initiate an

escape move. Otherwise, we choose the first non-tabu variable, in random order, that results in an objective function improvement, or the variable that results in the least performance decrease.

Aspiration Criterion:

In a tabu search, when a tabu criteria is used that goes beyond recording which solutions have already been visited, we may accidentally prevent moves that would lead to improving solutions. In our case, we use a second tabu list that prevents recently flipped variables from being flipped again quickly.

After each escape movement, we keep a running average μ_a of the cost (assuming a minimization problem) of all solutions as well as the cost of the best solution b_a found so far in the current "regime" (as the search period between two escape moves is often called in the literature). Then, we override the tabu lists if the solution we would arrive at has cost of at most $b_a + (\mu_a - b_a) * s$, where s is a continuous, dynamically self-updating parameter in $[0, 1]$.

4.1.2 Dynamic Search Features

Note that all parameters listed above take continuous values in $[0, 1]$, but we did not explain how they self update. In this section, we explain how.

Over the course of the tabu search, we keep track of a number of dynamic search features (runtime statistics) that are meant to characterize the status of the search. In total, we track eleven search features:

PercentTimeElapsed: At runtime, the user must specify how much CPU time is available for the search. This feature keeps track what percentage of CPU time has already elapsed.

BestUpdates: This feature counts the number of times we improve the objective function during the course of the entire optimization.

MovesAfterLastBestUpdate: In this dynamic search feature, we count the number of tabu search moves that have taken place since the last overall best solution was found.

MovesAfterLastImprovementInCurrentRegime: Analogous to the previous feature, in this value we count the number of moves that took place since the last improvement was found within the current regime. That is, after an escape move, we keep track of the best solution quality found afterwards. In this feature, we record how many moves have occurred since this improvement was found.

MovesInCurrentRegime: We count the number of tabu search moves after the last escape move.

TabuSolutions: Here we maintain the current length of the first tabu list which forbids recently visited solutions to be revisited.

TabuVariables: Anologous to the previous dynamic search features, in this value we record the current length of the second tabu list, which prevents specific variables that were recently flipped, from being flipped again.

AverageQualityThisRegime: Starting with the quality of the initial solution after the last escape move, we keep track of the average solution quality within the current regime.

AverageBestQualityThisRegime: In this statistic, we maintain the average quality of the best solution found within the current regime. That is, for every tabu search move

after the last escape action, we make an entry of the best quality that was found within the current regime until that move. This dynamic search feature is the average of this list of values.

BestQualityThisRegime: Here we record the best solution seen after the last escape move.

AverageBestQualityAtTheEndOfRegimes: Right before each escape move, we consider the best solution quality that was found within the regime that is now ending. This statistic maintains the average over these values for all completed regimes.

Note that the search features listed above may vary a lot from instance to instance, because of differences in instance sizes, different objective scales etc. Consequently, to allow offline training (see below), we need to normalize these features.

To normalize the number of overall best solution updates, we divide that number by an estimate of the total number of tabu search moves we will be able to conduct within the given time limit. This estimate is updated based on the actual observed time for the tabu search moves. This same estimate is also used to normalize the three following features which all count numbers of tabu search moves. The tabu list lengths values are divided by the number of binary variables in the problem instance to be solved.

Finally, to normalize all quality features we maintain the moving average of the quality of all accepted solutions as well as the quality of the current overall best quality found. The quality features are normalized by dividing the respective quality by the difference between average quality and best quality found so far.

4.1.3 Hyper-Parameterization

Given the set of dynamic search features defined in the previous section, the next core decision is how these features should determine the setting of the tabu search parameters. More formally, we need to define algorithmically how the search features as inputs ought to be used to compute the output: the current search parameters. Note that, historically, this was the task of the algorithm designer. Consider RTS from [16], for example. Here, the mechanism by which the length of the tabu list is increased or decreased is prescribed by the authors, whereby the decision to increase or decrease the length is dependent on search characteristics, such as the average cycle length observed during the most recent search.

The paradigm shift introduced in [19] is to let the machine *learn* this functional dependency between search features and search parameters. The decision that is left to the algorithm designer is merely what class of functions ought to be considered by the machine. Given the tremendous success of deep learning, it is tempting to propose a neural network structure that transforms features into parameters, in which the weights in the network would be learned during a training phase. The problem with this approach is the number of weights that would need to be considered as hyper-parameters. Note that, in the context of a local search metaheuristic, we do not have a fixed training set with supervised labels available to us. In fact, earlier search decisions affect the distribution of search decisions that need to be taken later (this problem was also identified when trying to devise search guidance for systematic search approaches, see for example [66]). Moreover, the quality of

the decisions taken can only be judged by running the search multiple times and possibly on a whole set of problem instances.

That is why we use an algorithm configurator to "learn" the hyper-parameters that ultimately determine how search features are transformed into search parameters. However, local search performance is not differentiable, and therefore there is no simple gradient decent approach available to us that would provide high-quality parameters very quickly. Consequently, the search for superior hyper-parameters is very tedious (thankfully only for the machine and not the user) and the search becomes more difficult the more hyper-parameters need to be tuned.

To keep the search for hyper-parameters manageable, we therefore need to choose a functional dependency between search features and search parameters that requires few hyper-parameters. A simple concept class consists of logistic regression functions which were also used in [19]. For each search parameter p, using k search features $f_1, \ldots, f_k$, we define a function $p \leftarrow \frac{1}{1+e^{w_0^p + \sum_{i=1}^k w_i^p f_i}}$, in which $w_0^p, \ldots, w_k^p$ denote the hyper-parameters that govern the relationship between the k search features and parameter p. Note that, for each search parameter, this choice requires just one more hyper-parameter than there are features. Moreover, the resulting values naturally cover the continuous interval $[0, 1]$ that all our parameters fell into.

4.2 Hyper-Reactive Tabu Search for MaxSAT

To evaluate the hyper-reactive meta-heuristic framework that we developed, it is important for us not to change the general framework to accommodate the particular problem class at hand. Consequently, the only calibration we allow for the particular application is the provisioning of an efficient way how to evaluate the objective function of a new assignment and, incrementally, the objective value change when one variable is flipped.

For MaxSAT, to enable fast incremental objective updates, we pre-compute once in the beginning which variables occur in which clauses and we maintain, for each clause, the set of variables that currently support the clause, if any. Then, when we flip a variable, we can easily consider only those clauses that the variable appears in and determine whether the variable is now, or no longer is, supporting the clause. By looking at the cardinality of the set of supporting variables, we can then quickly determine for which clauses we must add or subtract its penalty value to or from the previous objective value due to the variable change.

4.3 Experimental Results

We now report on the results of using the generic HRTS approach described above for tackling the MaxSAT problem. We first assess our contribution by comparing HRTS to a static RTS strategy. We then examine how the parameters change over time in HRTS when solving several MaxSAT instances to better understand the inner workings of HRTS. Finally, we show how HRTS parameterizations can be used to augment maxroster [67], the winning solver at MSE'17 [68].

4.3.1 Benchmark and Evaluation Metric

We use data from the annual MaxSAT Evaluations [24, 68]. In particular, we consider the scenario of a user who wants to build a superior MaxSAT solver after MSE'17. We train HRTS on the benchmarks from MSE'16. We tune multiple HRTS parameterizations using the GGA++ algorithm configurator [18, 45], splitting the training data into 16 *families* for tuning based on the filenames of the instances. All tuning was performed with a 60 second target algorithm timeout due to limited computational resources. Testing is all performed with a 300 second timeout.

We build a portfolio of HRTS parameterizations in combination with maxroster. To test the performance of the resulting portfolios we evaluate on the MSE'17 data. Note that all solvers in the portfolios are built and tuned on pre-2017 benchmarks.

We use the same performance metric as at MSE'17, which considers the average gap in quality. That is, for each instance we compute the ratio of best known solution to an instance divided by the final assignment cost of the respective solver for that instance. As the best known solution, we use the better of the two solutions found when comparing two solvers head-to-head, or the best quality published by MSE results [24, 68].

4.3.2 Tuning Setup

When optimizing the hyper-parameters, we need to define a function that establishes which parameterization is better than another when both have been evaluated on some MaxSAT instances. In the evaluation, we only run the algorithm to the best known solution quality and consider an instance "solved" if the solution was reached before the timeout. Parameterizations are first compared on average gap in quality. If that is equal, we next compare them based on the average time it took them to solve instances. Any remaining ties are broken randomly during tuning. As in [19], the surrogate model of GGA++ is trained by using relative ranks based on these pairwise comparisons.

We use a distributed version of GGA++ with 7 machines with 8 cores each and a memory limit of 32 GB each. The tuning uses a population size of 100 individuals and runs for 100 generations. In order to reduce experimental variance, each MaxSAT instance is evaluated with a common random seed.

To train the algorithm selector which picks the parameterizations at runtime, we use the cost-sensitive hierarchical clustering methodology (CSHC) [35]. We use the same 37 features from [69] and build 1,000 trees, whereby each is based on a sub sampling of the training instances with replacement with probability 0.7 and a subset of 6 features.

Experiments were run on a cluster containing Intel Xeon CPU E5-2670 processors at 2.6GHz running Scientific Linux 7.2.

4.3.3 Comparison with a Statically Tuned RTS

We create a static reactive tabu search variant, called SRTS, in which we only tune the full list of hyper-parameters that guide the reactive tabu lengths of the two

Family	Instances	HRTS		SRTS		ramp	dsat/wpm3
		Score	Solved	Score	Solved	Solved	Solved
auctions_auc-paths	20	100%	**20**	99.91%	12	18	17
auctions_auc-scheduling	20	100%	**20**	100%	**20**	**20**	19
frb	34	99.97%	29	99.98%	29	**32**	15
maxcut	48	100%	**48**	100%	**48**	**48**	47
min-enc_warehouses	18	77.64%	0	83.31%	1	1	**3**
spot5	42	99.01%	15	98.24%	13	28	**30**
Total	182	96.10%	132	96.91%	123	**147**	131

Table 4.1: Average score and number of instances solved for HRTS and SRTS on MSE'16 instance families after 300 seconds. We also give number of solved instances by local search solver ramp and the dsat/wpm3 portfolio from MSE'16.

tabu lists used in our approach. For all other tabu search parameters, we only tune the corresponding hyper-parameter w_0^p, which means that the corresponding search parameter is tuned yet stays fixed throughout the tabu search with no dynamically reactive behavior. This deprives the tuner from access to more dynamic search strategies, but on the other hand it has the advantage that the number of hyper-parameters to be tuned is lowered significantly.

Table 4.1 provides a comparison of HRTS and SRTS on several families of MSE'16 instances where local search traditionally works well. We consider an instance to be solved if the best known solution is reached. All runs are performed using a 300 second timeout.

Since the MSE'16 instances correspond to the training set, we use the results to assess the potential of the methods. We observe that HRTS provides a better or approximately the same score (the average ratio of best known quality over quality found by the solver) on five out of the six groups we consider. Overall, it therefore appears worthwhile to open more than just the tabu lengths for dynamic updates, even though the resulting tuning problem is much harder. As we see on family min-enc_warehouses, at times the tuning problem for the full HRTS may be so vast that an inferior parameterization is found. Overall, the existing state-of-the-art tuners (in this case GGA++ [45]) appear able to effectively tune even a fully hyper-parameterized version of tabu search, although we note that having enough instances available to tune is critical. With too few instances, HRTS is very prone to overfitting, although it is unfortunately not possible to exactly define "too few".

The configurations found for HRTS on these families are very competitive. On most families, HRTS outperforms the hyper-parameterized dialectic search [19] based portfolio in MSE'16 and works comparably well as human-developed MaxSAT solvers, like, e.g., ramp, the local search solver that maxroster employs. Note that the search guidance in HRTS knows nothing about MaxSAT itself, but is nonetheless competitive with state-of-the-art approaches.

4.3.4 Runtime Log Analysis

We now investigate how the parameters change over the course of a single run of HRTS, and split our analysis into looking at differences in the same parameters over different instances, and different parameters on the same instance. For the following

experiments we use a 250 second timeout.

Dynamic Search Behavior Using the Same Parameterization:

The scpnre instances are based on the well-known set covering problem. Figure 4.1 tracks eight different values over the course of solving three scpnre instances (SCPNRE1 (blue), SCPNRE2 (green), and SCPNRE3 (red)), when using the *same* hyper-parameterization of HRTS.

We observe that, on all three instances, this parameterization makes quick progress in improving the solution quality, using a large neighborhood size for the first-improvement tabu greedy steps that are only diversified by making 5-7% of recently flipped variables and practically no individual solutions tabu in the beginning. This small diversification pressure is even further reduced by a very generous aspiration policy. After the initial phase of fast improvements, the aspiration criterion is made gradually more restrictive until only best solution improvements are allowed to override the tabu lists. In turn, however, the tabu variables list length decreases until this list is practically not used anymore at around half of the available computation time. Instead, the tabu solutions list length is increased, so that it has a total length of about 0.8 times the number of variables.

In the second half of the optimization, this list length is adjusted very dynamically, the aspiration is slightly relaxed again, and the probability for escape moves is increased, whereby the escape path length is controlled carefully. 0.4% of variables corresponds to roughly 20 variables on these instances. Furthermore, the average escape size (in two of the three instances) slowly ratchets up before smoothly declining. As a result of these changes in search behavior, we begin to see new and some abysmally bad solutions as evidenced by spikes in the average solution quality. This shows that the method is diversifying effectively, and the success of this strategy shows in the best solution quality: two of the three runs find improving solutions during this second half of the optimization.

Overall we see that the search is reactive, yet the overall search strategy (if one can call it that given its origins in tuning) is comparable on all three instances.

Comparison Of Multiple Parameterizations On the Same Instances:

Having just compared the dynamic search behavior of the same parameterization on different instances, in Figure 4.2 we now compare two *different parameterizations* (solid and dashed lines) which we both run on the *same* instance, SCPNRG2. The dashed parameterization finds good solutions faster, as seen in the plot of the best value, although the solid parameterization does end up finding similar solutions later in its search.

The example shows beautifully how flexible the hyper-parameterized tabu search framework is. Let us look at the solid lines first. This is almost a pure tabu search strategy: The number of recently flipped variables that are tabu is kept firm at 5%, the tabu solutions list is not used. Furthermore, there are no escape moves at all, and almost all variables are open for flipping at any tabu best neighbor step. The only difference to a very traditional tabu search approach is the use of the aspiration

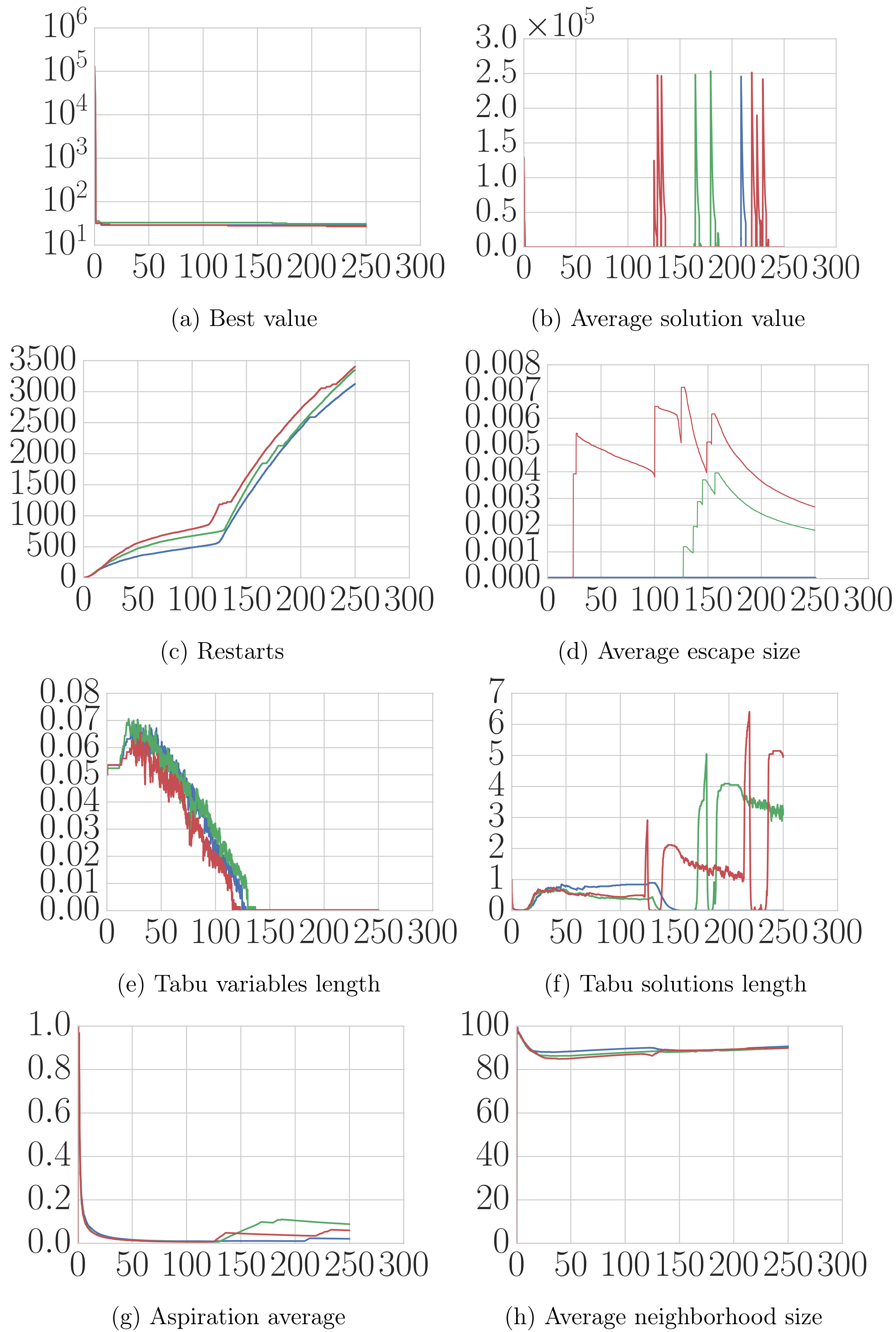


Figure 4.1: Normalized characteristics over the course of running three SCPNRE instances, with seconds on the x axis.

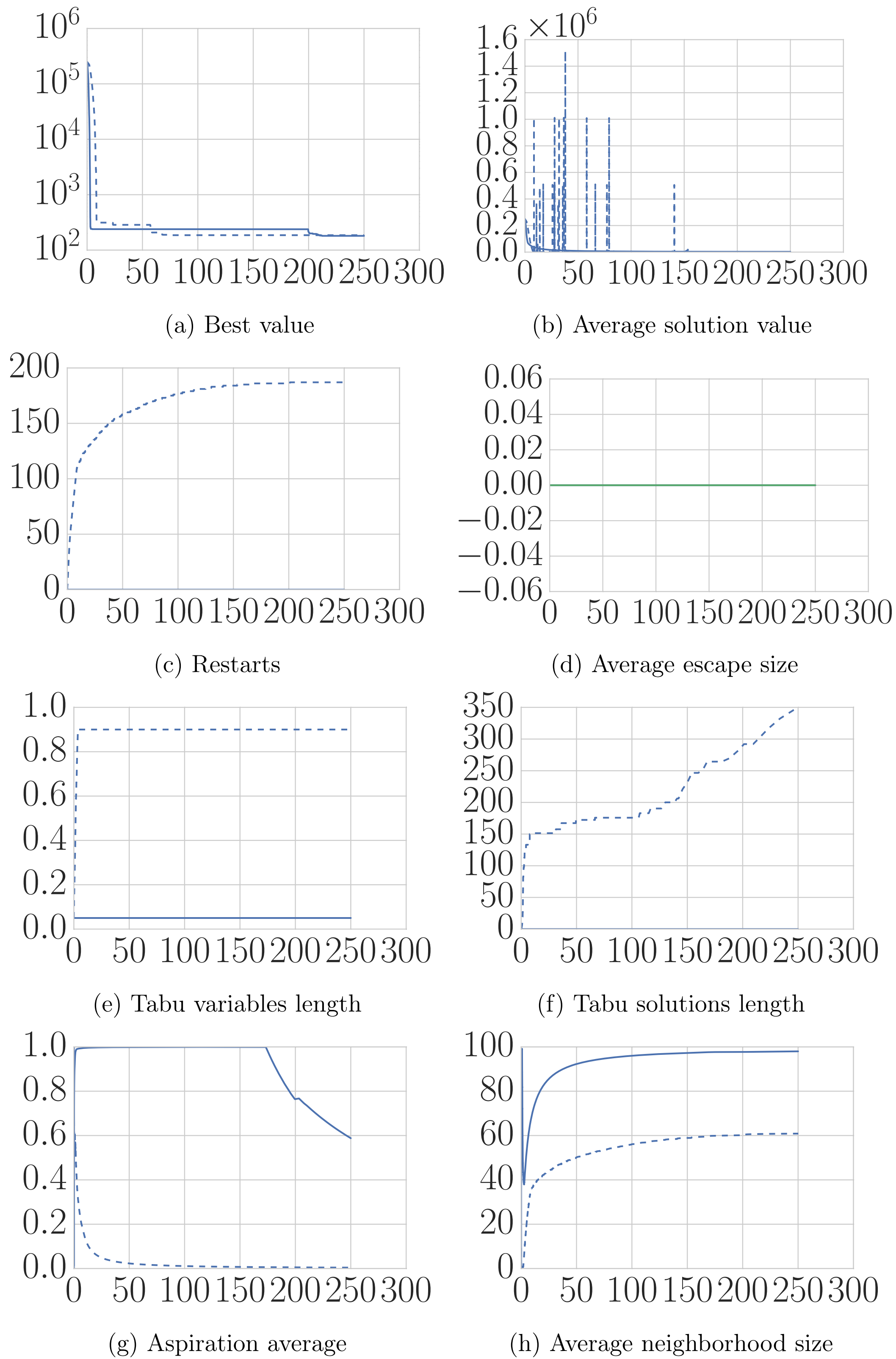


Figure 4.2: Normalized characteristics over the course of running the SCPNRG2 instance with two parameterizations, with seconds on the x axis.

criterion which allows practically any best neighbor to override the tabu variables list as long as the resulting quality is still below average cost. Only after about two thirds of the available runtime are exhausted, the aspiration criterion is made more restrictive.

Contrast this strategy with the one that leads to the dashed lines in our plots. Here, we observe that both tabu lists are used excessively: About 90% of recently flipped variables are tabu, and the history of "recently" visited solutions that is kept on file grows from 150 times the number of variables to 350 times the number of variables. Note further that this massive diversification pressure is not alleviated by a lax aspiration policy either. At the same time, only 50-60% of variables are considered for flipping during first improvement tabu steps. Obviously, this very aggressive, "go after the best you can find that is new" policy may lead to dead ends quickly. Consequently, the search restarts quickly, whereby, curiously, the method does actually *not* diversify at this point by using the hash table to find a good new starting point. The escape size is held firmly at zero. Merely, the restarts are used to clear the tabu lists only.

This is the quintessential take-away of hyper-parameterization: It allows the machine to drive the search in many different ways, explore vastly differing strategies, more than humans have considered and bothered to implement and test. Within the context of tuning and algorithm portfolios, it is *diversity* that makes all the difference. Any individual strategy is no longer required to work robustly across many different applications. As long as a certain policy can excel sometimes, it can become a valuable member of a portfolio of parameterizations.

4.3.5 Comparison With the State Of The Art

Finally, we show what effect the hyper-parameterized tabu search framework can have in a domain where regular solver competitions have driven the development of incomplete methods for many years. We apply our framework to MaxSAT, trained on the MSE'16 benchmarks, and with automatically tuned hyper-reactive tabu parameterizations added, like the ones studied earlier, to a portfolio of solvers. In Table 4.2 we show the results when adding our parameterizations to maxroster, the winning solver at MSE'17, and compare to the virtual best solver (VBS).

In Table 4.2, we can see that entirely self-tuned tabu search hyper-parameterizations, which realize vastly differing search strategies, can effectively complement the current best MaxSAT solver, maxroster. The score (average ratio of the best known solution over solver quality) improves from 0.9721 to 0.9771 on the training set (we use the best quality in a portfolio of maxroster, wpm3 and HRTS parameterizations to define the instance best) and from 0.8311 to 0.8321 on the test set. This is achieved by improving the quality on 7 out of the 156 test instances, or about 4.5% of all instances, without ever decreasing it.

This leads us several important conclusions: First, hyper-parameterized tabu search can be tuned effectively. This is true even when the training set is not 100% indicative of the test set (the benchmark used at MSE'17 differed significantly from prior MaxSAT evaluations). Second, hyper-parameterized tabu search can realize search strategies that differ significantly from existing, human-incepted search

		maxr	maxr+HRTS
Training (961 instances)	Score	97.21%	**97.71%**
Test (156 instances)	Percent VBS	99.89%	**100%**
	Head-to-Head Wins	0	**7**
	Score	83.11%	**83.21%**
	Best Found	69	69

Table 4.2: Head-to-head comparison of maxroster and a maxroster+HRTS portfolio.

methods, and thus offer the opportunity for improvements within a portfolio approach. This conclusion is analogue to the results presented in [56]. And third, as a consequence of the previous two points, even in entrenched, highly researched domains, hyper-parameterized tabu search can be applied effectively to achieve performance improvements automatically, without any need of the user to provide any domain knowledge. In our case, HRTS has access to an incremental cost evaluator for MaxSAT. That is all that the framework can exploit. Everything that regards search guidance, how to dynamically adapt, and what overall strategy to run, is entirely driven by the search experiences that are gathered during training, by monitoring the search features that we described in this chapter and correlating these with the overall search performance.

4.4 Conclusions

We introduced a framework for the hyper-parameterization of tabu search. Our framework opens tabu search parameters such as escape rules, neighborhood sizes, and aspiration criterion for the dynamic adaptation during search, on top of the length of tabu lists which had been pioneered in [16]. The dynamic self-adaptation is based on various search features that we introduced. Experimental results show that portfolios of hyper-reactive tabu search parameterizations work competitively with human devised local search solvers for MaxSAT. Moreover, these parameterizations can even improve the most cutting edge MaxSAT solver maxroster in a joint portfolio approach.

Chapter 5

Boosting Evolutionary Algorithm Configuration

AI research has created powerful tools, programming libraries as well as cloud and device services, which are now publicly available to help human decision makers find information, semantically link it to other information, provide forecasts, and, most importantly, *prescribe* favorable courses of action. The paradigm of declarative programming consists in the idea that a user could be liberated from the task of *how* to find such prescriptions. Rather, the user only *declares* what properties so-called "solutions" should have. *Solvers* are key for provisioning this desirable capability of liberating the user from having to program the actual control flow of a machine's reasoning. The solver is the engine that combines search and intelligent reasoning to find solutions that exhibit the properties the user declared.

Consequently, solvers that effectively tackle hard search problems have been developed in all communities that study these problems. From logic programming to constraint programming to mathematical programming, from satisfiability to SAT modulo theories, from quantified Boolean formulae to maximum satisfiability: in all communities one of, if not: the core quest is the development of more efficient solvers that push the boundary of what kind of problem instances can still be solved in practically feasible computation times before, ultimately, the asymptotic impossibility of provisioning solutions for NP-hard problems in affordable time overpowers the machine.

Notably, it was found that different algorithmic approaches exhibit strong complementary strengths in their ability to efficiently solve problem instances that exhibit different characteristics [70, 71, 72]. An instance that is solved in seconds by one approach may take days and months by another - and vice versa! In fact, it is often observed that the same is true for different parameterizations of the same algorithm.

In light of this situation, consider the scenario where a company or a research group has devised a new combinatorial algorithm, a solver, like CPLEX for solving mathematical programs, or SparrowToRiss, for solving logic programs. Not knowing the instances their solvers will ultimately be run on by the users of these solvers, the developers of these tools have three options: They can either aim to find good default parameters for their solver that work reasonably well, by hand. They can try

to write a users' manual that explains the solver parameters to the users so that these can search for good parameterizations for their respective applications. Or they can employ an automatic solver configurator that finds decent default parameters, as well as allows users to tune the solver for their respective instance after deployment.

Algorithm tuners were developed that tune and customize solvers to work better on specific sets of problem instances [18, 30, 32]. Moreover, managerial dispatch technologies were invented that choose, at runtime, which solver to run for a given instance [35, 37, 73, 40, 74]. And finally, combining self-tuning and selection technologies, methods have been invented that choose a solver parameterization instance-specifically at runtime [33, 75].

In this chapter, we aim to improve the automatic solver configurator GGA [18] in three steps: First, we present a forward simulator that helps improve the performance of the configurator on a parallel machine. Second, we introduce the idea of an "elite mini-tournament" of solver parameterizations to improve the tuning process itself. And third, we show how the parameterizations encountered during the course of the configuration process can be utilized to build a solver portfolio.

In the next sections we introduce our new ideas in detail, and eventually evaluate and compare the new solver configurator in a series of practical experiments.

5.1 Improving Parallel Efficiency Using an Evolution Simulator

Due to the extremely high computational costs of automatic solver configuration, these algorithms must run on parallel machines to achieve acceptable configuration times. In its current form, GGA has a strict synchronization point at the end of each generation, which limits its parallel efficiency. In this section, we explain how we can parallelize GGA and obtain the exact same results as GGA, but with significantly improved parallel efficiency.

Key to this improvement is a forward simulation of the parallel execution which allows us to *safely interleave evaluations from consecutive generations.* At first, this idea may sound speculative, similar to how a processor anticipates which computations may have to be conducted next, for example, by running the "then"-block of an "if"-statement before the result of the conditional jump is actually known. Obviously, this avoids idleness, but it may lead to computations that the original version of the algorithm would have never executed. Consequently, speculative computations may keep everyone busy, and yet still not lead to good speed-ups over the original algorithm.

Instead, we suggest to pre-compute which evaluations will *surely* have to take place in future generations and only execute those computations that the original version of GGA would also have conducted. To realize this vision, we need to somehow pre-compute which future GGA target algorithm computations will have to be carried out no matter how the currently unfinished mini-tournaments will turn out. To this end, we introduce the idea of an *evolution simulator.*

The idea of the simulator is to determine all random decisions as early as possible. Similar to a sports tournament, or the data flow graphs used in Spark [76], we can

decide that the winner of some mini-tournament will mate with a specific subset of non-competitive individuals, and pre-determine the gender of their offspring. Also, we can decide upfront which inputs will be used in which generation. Then, as soon as the concrete parents are determined, we can create the offspring. If that offspring has the competitive gender, we know which target algorithm evaluations need to be conducted, and we can do it even before all mini-tournaments in the previous generation are completed.

When we pre-pone the random decisions, we can create an *evolution simulator* based on a *precedence graph* which tells us which target algorithm evaluations need to be conducted before others. Nodes in this graph are either *virtual genomes* or *virtual mini-tournaments*. Genome nodes would represent individuals in the population and tournament nodes represent the (unknown) winners of a mini-tournament. Nodes are virtual, because the actual genomes will only be known at runtime. The arcs in the graph express which other nodes need to be actualized before a node can be evaluated.

Genome nodes receive two input arcs from their parents (except for the initial randomly created genomes which are immediately actualized). A competitive genome node has as many output arcs as mini-tournaments it competes in. A non-competitive genome node has as many output arcs as it will have offspring. A tournament node has as many input arcs as there are competitive genomes it races and, as it represents the winner in its actualization, as many output arcs as the winner has offspring.

Algorithm 8 Evolution Simulator

Input: Target Algorithm A, Parameter Space Θ, Instances Π, Number of Generations NG, MiniTournament Size N
Output: A precedence graph that represents the trace of the GGA algorithm
1: graph, pop $\leftarrow$ initPopulation(Θ)
2: **for** $g = 1$ **to** $g \leq NG$ **do**
3: inst $\leftarrow$ selectInstances(Π)
4: mini_t $\leftarrow$ createMiniTournamentNodes(graph, A, pop.comp, inst, N)
5: offspring $\leftarrow$ createCrossoverNodes(graph, pop.noncomp, mini_t, Θ)
6: pop $\leftarrow$ agingAndDeath(pop) $\cup$ offspring
7: **return** graph

The simulator, shown in Algorithm 8, follows the exact same structure as the original GGA (Alg. 1), with the necessary changes to accommodate the creation of the precedence graph. First, *initPopulation* creates the initial population nodes (g_x) and initializes the precedence graph. These initial nodes form the active population (*pop*). At Line 4, instead of running the mini-tournaments, it creates the mini-tournaments nodes (MT_x) and sets the appropriate arcs. Similarly, Line 5 creates new nodes that represent individuals simulating the crossover between the non-competitive nodes and the recently created mini-tournaments nodes. Finally, at Line 6, *pop* is updated with the nodes that survive to the next generation and the offspring from the crossover operation.

Figure 5.1 shows an artificial example of a simulation graph describing the prece-

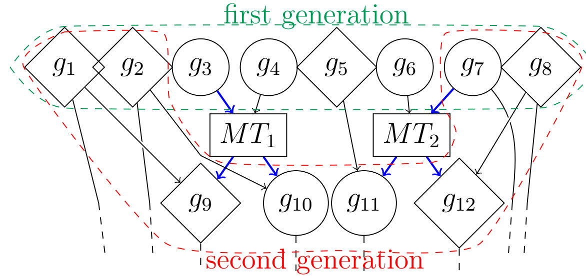


Figure 5.1: Evolution simulation graph

dence relations that would generate the GGA algorithm. In the first generation, there are eight genomes which are randomly split into four non-competitive virtual genomes $\{g_1, g_2, g_5, g_8\}$ (the diamond-shaped nodes) and four competitive virtual genomes $\{g_3, g_4, g_6, g_7\}$ (the circle nodes). The age they initially have, randomly assigned from 1 to 3, is $\{2, 1, 3, 3, 3, 3, 1, 1\}$, respectively.

The competitive nodes are randomly selected to be raced in two mini-tournaments $\{MT_1, MT_2\}$ (rectangular nodes) of size two each. The mini-tournament nodes represent the winners. Therefore, MT_1 will represent, in its actualization, either g_3 or g_4. A priori, we do not know which genome will win, but we know that each winner has the right to reproduce. Then, the winner is paired at random with some non-competitive genomes. In our example, the death age limit is 3. Therefore, genomes g_3, g_4, g_5, g_6 are replaced by the new offspring $g_9, g_{10}, g_{11}, g_{12}$, which are randomly labeled competitive or non-competitive with a 50% probability as done in GGA. At this point, we have all the information to extend the simulation graph with nodes of the second generation.

There is one complication: The overall best competitive winner does not die, no matter its age, as long as it performs best. However, which competitive individual is the overall best and therefore excluded from age termination can only be identified at runtime. To address this issue, among the offspring of every mini-tournament, at least one offspring of each gender is forced to exist. In the actualization, if the tournament winner exceeds the age limit but is the overall most competitive individual, then the forced competitive offspring will become a clone of the parent, at age 0.

In Figure 5.2, we showcase the result of our efforts. The figure shows two block diagrams of the parallel execution of genomes g on various training instances i. The bottom of the figure shows the real execution of two generations of the original GGA using 4 cores, the top shows the execution of the newly proposed algorithm. Note that we tune the target algorithm for runtime, consequently, the width of the individual executions of the same instance i may vary with the algorithm parameterization g that is used to tackle it.

As we see, the evaluations of the second generation start when the evaluation of genome g_4 on instance i_2 has finished, leaving three cores idle. Notice that, in GGA, once an evaluation (with a given timeout) has been initiated, it cannot be canceled on demand. Moreover, since a given genome can be executed on the same instance several times across its life period, it can be more efficient to let the evaluation finish

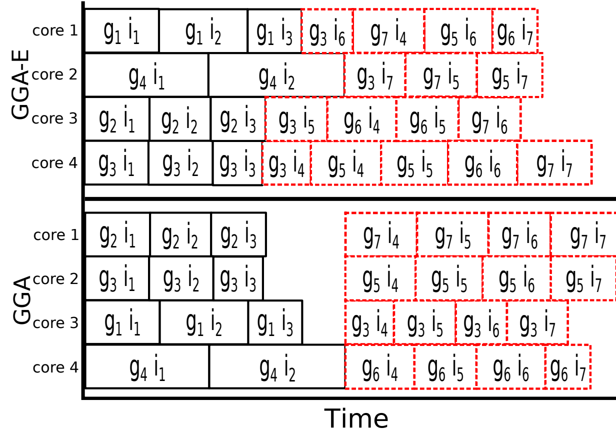


Figure 5.2: Timeline of evaluations in GGA-E and GGA

and access the cache for later re-evaluations.

At the top, we see the new algorithm (referred to as GGA-E) executing the same mini-tournaments. Notice that GGA-E allocates work opportunistically, i.e., GGA-E uses as many cores as possible, therefore individual parameterizations are no longer tied to just one core. For example, genome g_3 is executed on different instances on cores 2, 3, and 4.

5.1.1 Experimental Results

Here and in the remainder of this chapter, we run our experiments in a computer cluster with nodes equipped with two octo-core Intel Xeon Silver 4110 @ 2.10 GHz processors and 96 GiB of RAM. We tune satisfiability (SAT) and mixed-integer programming (MIP) solvers on standard algorithm configuration benchmark sets that consist of train/test splits of SAT and MIP instances.

We configure the award-winning SAT solver SparrowToRiss on the industrial benchmarks used in [77]: Bounded Model Checking (BMC), Circuit Fuzz (CF) and IBM-Hardware Verification, and crafted benchmarks: Graph Isomorphism (GI), Low Autocorrelation Binary Sequence (LABS) and N-Rooks, all from the algorithm configuration library AClib [78].

Furthermore, we tune the commercial state-of-the-art integer programming solver CPLEX on the Assortment [79], MIRPLib [80] and RCW [81, 82] benchmarks.

Unless otherwise noted, for the configuration phase, the total wall-clock time is 2 days, the evaluation timeout for each (parameterization/instance) pair is 300 seconds, and the memory limit is 6 GB. GGA and all its variants are run with population size 100 and mini-tournament size 8.

For the test phase, we use a time limit of 300 seconds for SAT instances and 1,800 seconds for CPLEX, which reflects the relative difficulty of the respective benchmarks. We report the PAR10 performance and, in parentheses, the number of instances that were fully solved (including a proof of unsatisfiability, if applicable, or optimality, in case of optimization problems) within the time limit. The PAR10 metric is defined as follows: When an instance is fully solved within the time limit, we record the time that was needed to solve the instance. If the time limit is reached

before the instance is solved, then the run is penalized with a value of 10 times the time limit. We then aggregate and average over all test instances for which at least one approach of the ones we compare was able to solve the instance within the time limit. That is, the PAR10 penalty for an unsolved instance is 0 provided no solver or solver parameterization considered are able to solve the instance within the time limit. For each benchmark, we note in bold the best overall approach on the test set.

In Table 5.1, we first compare the original GGA and the new version GGA-E that uses the evolution simulator. We show the number of generations after two days elapsed time tuning SparrowToRiss on benchmarks CF, IBM, GI, BMC and N-Rooks, as well as CPLEX on benchmarks Assort, MIRP, and RCW. As we can see, except for MIRP, GGA-E is able to complete more generations than GGA thanks to the more efficient desynchronized parallelization. In particular, for LABS, GGA-E performs around 32,000 evaluations, compared to 17,000 evaluations that GGA is able to conduct in the same time using the same number of processors. This marks an 88% improvement, which is strong evidence to support that the evolution simulator can help reduce idle times very significantly.

Moreover, on all benchmarks tested, we found that the system load during the configuration process was always a little above 16 for GGA-E, which corresponds almost exactly to the 16 workers, plus some overhead from the operating system. This implies that, on benchmarks where we do not see a great improvement in efficiency, for example on MIRP, the original GGA is already making maximal use of the available compute power. However, whenever this is not the case, we see that GGA-E has effectively reduced idle times close to zero.

Table 5.1: Number of generations after 2 days elapsed time.

	SparrowToRiss						**CPLEX**		
	CF	IBM	GI	BMC	LABS	N-Rooks	Assort	MIRP	RCW
GGA	19	12	11	8	11	27	15	**12**	4
GGA-E	**23**	**14**	**12**	**10**	**15**	**51**	**20**	**12**	**6**

5.2 Improving GGA

5.2.1 Instance Selection Strategy

The original strategy used by GGA is to randomly select the (increasing number of) instances that competitive genomes are compared on at the beginning of each new generation. While, in GGA, two consecutive generations can work on a completely different set of instances, in IRACE and ParamILS [44]/SMAC the most recent set is a randomly augmented super-set of the previous set. We investigate whether this strategy also works for GGA: We randomly shuffle the instances only before the first generation. Then, at the beginning of each new generation, we select instances in that static random order until the (increasing) size of the training subset is reached. That is to say, in consecutive generations, a super-set of the instances used in the

previous generation are being considered for fitness evaluations, just as it is done in ParamILS/SMAC. This experiment is of interest since the racing mechanism are different in ParamILS/SMAC, IRACE, and GGA.

Our hypothesis is as follows: Randomly choosing a completely newly sampled subset of instances in each generation may help with population diversity, as competitive parameterizations with complementary strengths on different instances have the chance to parent new offspring. On the other hand, using a monotonically increasing set of instances (a strategy that we denote with 'M') may help focus the optimization on promising parameterizations. Moreover, GGA's cache for avoiding re-evaluations may be far more impactful when reusing instances from the previous generation.

5.2.2 Elite Mini-tournament

When running GGA, we found that several parameterizations, that used to be overall most competitive at one point during the configuration process, would have been also the best parameterization at the end of the entire configuration process, but had vanished in an earlier generation when they were outperformed by a competitor when run on a specific subset of training instances. To prevent GGA from losing high-performing genomes, we therefore experiment with an "elite" mini-tournament that we add in each generation where all overall winners from previous generations compete in one champions league ('C'). The downside is that the elite mini-tournament could possibly cause undue computational overhead.

However, when analysing the additional effort carefully, we can see that the additional effort is very affordable, and this is not in small part due to the previously introduced modifications: First, notice that all new champions have already been evaluated on all instances of the previous generations, as mini-tournament champions are by definition never interrupted by a competitor that runs faster. Therefore, thanks to the monotonic ordering of instances and the cache, their evaluation is mostly limited to the new instances added in the current generation and consequently very fast. Second, champions are high-performing parameterizations. Consequently, the winner of winners, which determines the runtime for the elite tournament thanks to the parallel racing in GGA, can be expected to run very fast. Finally, to prevent that the fast elite tournaments run too far ahead of the progression of other generations, we schedule outstanding tournaments from earlier generations before we run mini-tournaments on younger generations.

Putting all three improvements together: We refer to the version of GGA that uses the evolution simulator with GGA-E, the monotone instance selection GGA-M, and the champions league tournaments GGA-C. The new version that uses all three strategies is named GGA-EMC, and analogously for any subset of the new strategies introduced above.

5.2.3 Experimental Results

In Table 5.2, we repeat the assessment of parallel efficiency from Section 5.1.1 by tuning SAT solver SparrowToRiss and MIP solver CPLEX for various benchmarks

Table 5.2: Number of generations after 2 days elapsed time.

	SparrowToRiss						**CPLEX**		
	CF	IBM	GI	BMC	LABS	N-Rooks	Assort	MIRP	RCW
GGA	19	12	11	8	11	27	15	12	4
GGA-E	23	14	12	10	15	51	20	12	6
GGA-EC	22	14	11	9	15	50	20	11	6
GGA-EM	**39**	**22**	**15**	**16**	**23**	**53**	**31**	**19**	**9**
GGA-EMC	**39**	**22**	**15**	**16**	**23**	**53**	**31**	**19**	**9**

Table 5.3: PAR10 performance (# solved instances).

		GGA	GGA-E	GGA-EM	GGA-EMC
SAT	CF Test	121 (280)	121 (280)	119 (280)	**89 (283)**
	CF Train	68 (284)	57 (285)	58 (285)	**48 (286)**
	IBM Test	21 (231)	23 (231)	21 (231)	**10 (232)**
	IBM Train	**34 (298)**	53 (296)	52 (296)	41 (297)
	GI Test	108 (315)	150 (310)	159 (309)	**91 (317)**
	GI Train	117 (922)	161 (907)	176 (902)	**73 (937)**
	BMC Test	159 (268)	**134 (270)**	192 (265)	171 (267)
	BMC Train	121 (547)	**92 (552)**	132 (545)	133 (545)
	LABS Test	200 (255)	275 (248)	**197 (255)**	275 (248)
	LABS Train	267 (255)	290 (253)	**193 (262)**	267 (255)
	N-Rooks Test	6.6 (351)	**5.7 (351)**	6.5 (351)	6.3 (351)
	N-Rooks Train	5.7 (484)	**4.8 (484)**	5.9 (484)	5.6 (484)
MIP	Assort Test	3030 (50)	**393 (59)**	708 (58)	718 (58)
	Assort Train	353 (49)	117 (53)	**65 (54)**	**65 (54)**
	MIRP Test	**3340 (70)**	5822 (58)	4583 (64)	5164 (61)
	MIRP Train	652 (60)	764 (57)	**614 (61)**	724 (58)
	RCW Test	557 (844)	435 (851)	516 (845)	**344 (855)**
	RCW Train	304 (698)	406 (671)	**238 (713)**	265 (708)

and report the number of generations competed within the configuration time limit for each variant of GGA. We observe that GGA-EMC consistently completes more generations than GGA-E. Even though GGA-E had practically eliminated all idle time that GGA exhibited, it is forced to conduct more work per generation as there are a lot of new instances in the early generations, rendering the cache less effective. The use of the caching mechanism in combination with monotone instance selection clearly allows GGA-EM to work through more generations faster. We also see that the use of the additional elite mini-tournament 'C' does not affect this result, as the results for GGA-EM and GGA-EMC are the same in terms of the number of generations that can be computed in a given amount of time. This supports our earlier analysis that adding the elite tournament does not cause major computational overhead.

Of course, at the end of the day what matters is not how many generations the tuner can work through, but how good the resulting parameterizations are. In particular, in Table 5.3, we report the train and test quality achieved by the different variants of GGA. The table gives the PAR10 score and, in brackets, the number of instances solved within the time limit. We observe that performance improves the

more of the new strategies we add to GGA. Due to the stochastic nature of the evolutionary algorithm, the behavior is not strictly monotone, and what leads to an improvement on one tuning benchmark may be detrimental on another. Overall, we can observe that GGA-EMC provides the best test performance on four out of the nine benchmarks considered. Moreover, GGA-EMC achieves an average rank of 1.61, clearly outperforming the other variants (GGA-E 2.56, GGA-EM 2.61, and GGA 2.78). What this data shows is that the performance of GGA-EMC is not the result on any one individual modification, but their synergy effect when applied in concert.

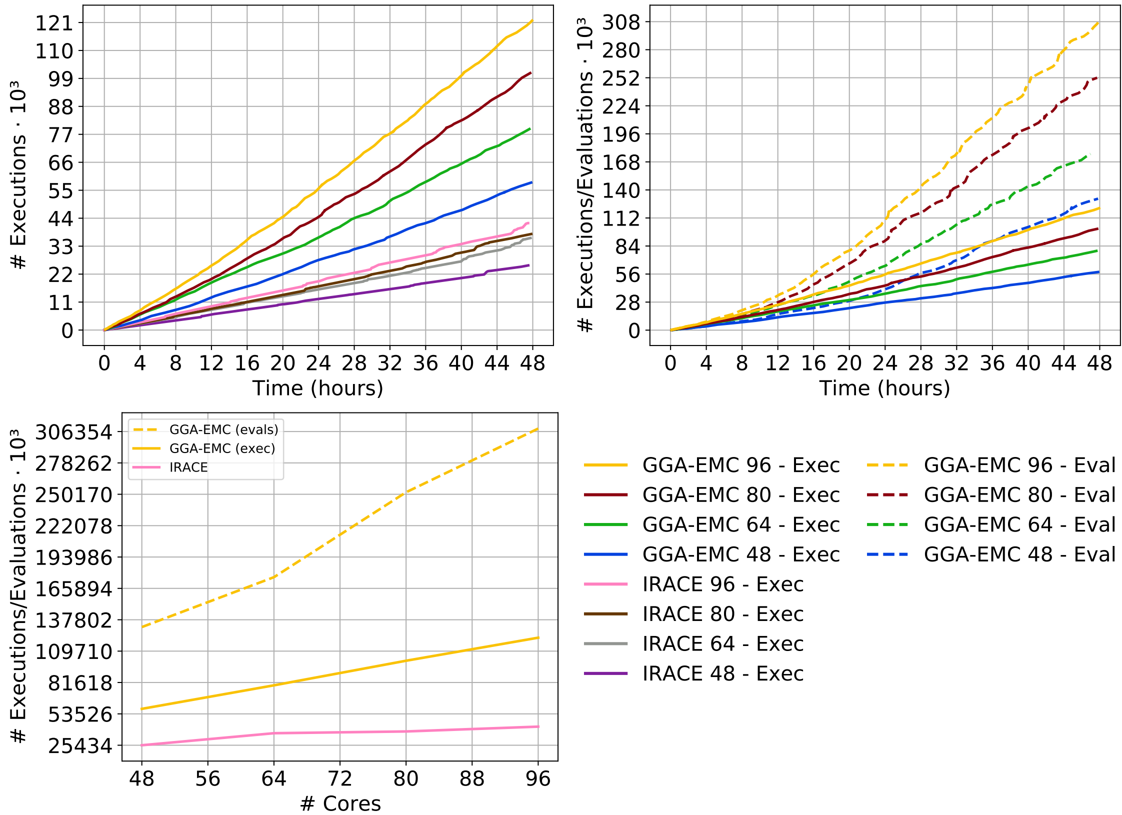


Figure 5.3: # of Evaluations and Executions on IBM.

We conducted an additional experiment to analyze cache and processor utilization as we scale the number of cores. We tuned the SAT solver SparrowToRiss with GGA-EMC and IRACE v3.0 on the IBM benchmark.[1] In Figure 5.3 (top-left), we plot the number of executions as a function of time, for different numbers of cores: $48, 64, 80$ and 96. We see that, in terms of processor utilization, GGA-EMC is more efficient than IRACE. After 2 days on 96 cores, GGA-EMC has performed over 121,000 runs of the target algorithm, while IRACE only conducted a little over 42,000, roughly one third of the executions run by GGA-EMC.

[1]Note that SMAC conducts independent Bayesian optimization runs on as many workers as available which trivially causes SMAC's CPU utilization to scale linearly in the number of workers. At the same time, though, this trivial parallelization scheme leads to an ever diminishing value of each additional CPU core for the task of solver tuning.

On the top-right, we see that GGA-EMC performs many more evaluations than actual executions (with *execution* we refer to a *real* run of the target algorithm on a given input, while an *evaluation* is either a real execution or a cache look up). This shows that GGA-EMC revisits the same parameterization/input pairs rather frequently, as discussed previously. Thanks to the evaluation cache, these redundant evaluations no longer require a re-run of the target algorithm. This consequently leads to a much more effective compute utilization. The total number of evaluations considered by GGA-EMC during 2 days on 96 cores comes close to 310,000, 2.5 times the number of actual target algorithm executions. On the bottom, we plot the numbers of evaluations and executions as a function of the number of cores for GGA-EMC and IRACE. As we see, the utilization advantage of GGA-EMC becomes ever more pronounced the more cores are available.

5.3 Instance-Specific Parameter Selection

We have introduced strategies to make GGA run faster by reducing idle times, and have shown how to improve its ability to find a better solver parameterization for a given set of reference instances. While instance-oblivious tuners greatly improve the performance of solvers, they still do not address the issue that parameterizations, even very good default parameterizations, do not work evenly well across heterogeneous sets of instances. One fixed parameterization simply does not fit all instances equally well.

Therefore, we now consider the last step of solver configuration, and that is the ability of provisioning a solver parameterization that is particularly suited for the problem instance at hand, at runtime. That is to say, we tune a solver offline, but search not only for *one* parameterization, but for a whole *variety* of different parameterizations. Then, we provide a method to pick one of the parameterizations created offline after the concrete instance to be solved in known, at runtime.

This approach obviously offers theoretical potential for further improvements, provided that we can manage to select a solver parameterization at runtime that works well for the instance at hand. That is to say: If we had an oracle that could provide the best parameterization for tackling a given instance, then instance-specific solver configuration would be a great idea.

The problem is, we do not have such an oracle. We could, for example, try to learn a surrogate that forecasts solver performance given a parameterization and instance properties. Then, at runtime, we could search for superior parameterizations given the instance features observed [83]. However, it has been found that the surrogates we can learn offline are simply not accurate enough for this approach to work robustly.

Instead, the current state of the art is to produce, at training time, a small, manageable set of few parameterizations that exhibit complementary strengths in solving subsets of different problem instances in the training set. At runtime, the task of the oracle is now reduced to picking one of these relatively few parameterizations, based on the characteristics of the concrete instance at hand.

One approach that realizes this methodology is ISAC [33]. It is based on the

assumption that the instance features, which will be used at runtime to select the best parameterization, ought to give us a good idea already as to which instances are similar in nature. Consequently, by clustering instances based on their features, we should be able to group instances together that can be efficiently solved by the same parameterization. Therefore, ISAC first clusters the training instances described by a set of computed features and runs an instance-oblivious tuner (like GGA-EMC) for each cluster of instances to produce as many parameterizations as there are clusters in the training set (ISAC automatically determines the number of clusters based on g-means clustering [34]). At runtime, ISAC then simply computes which cluster of training instances is closest (as measured by Euclidean distance to the feature vector of the cluster center) to the runtime instance, and uses the parameterization that was specifically trained for this cluster.

In a later iteration, ISAC++ [69] employed an algorithm selector (portfolio), namely cost-sensitive hierarchical clustering (CSHC) [35], which lead to performance improvements. In particular, the portfolio is built on the result of executing the parameterizations found by ISAC (as many as clusters) on the whole training set.

Another method to pre-compute a set of parameterizations offline and then select one of these at runtime is Hydra [75]. At training time, Hydra first searches for a good overall default parameterization. Then, Hydra re-runs the instance-oblivious tuner, but this time under the assume that the default parameterization that was just found will already handle all instances for which the new parameterization cannot improve upon. That is to say, in the second tuning, the performance of all instances is now assumed to be at least as good as the performance of the initial parameterization found.

This process is repeated, and in each new iteration we already assume that the portfolio of already generated parameterizations will handle those instances for which the new parameterization under construction provides worse performance than the best parameterization already in the portfolio. In this way, Hydra gives newly created parameterizations the opportunity to improve for some training instances, at the cost of doing worse on others, which drives the overall portfolio to exactly the kind of diversity that is required to handle different instances well.

Seeing that algorithm selection technology has greatly advanced since the original inception of Hydra and ISAC, the question arises if we could not afford to build a much larger set of parameterizations at runtime from which the algorithm selector may choose from, without running the risk of overwhelming the selector to the point where it is no longer able to choose a superior parameterization consistently. This is the starting point for our study below. Put simply, we propose to run a tuning algorithm and to collect a whole set of high-performing parameterizations that were encountered during the optimization process.

5.3.1 A Portfolio of All Parameterizations

Rather than building a large set of parameterizations in a guided approach (like, e.g., Hydra follows), in this staretgy we harness the *accidental variance of parameterizations found during the course of instance-oblivious solver configuration*. More precisely, the idea is the following: We run GGA-EMC (or another tuner) and use

its performance cache of all competitive parameterizations encountered during the configuration process. For all instances that a parameterization was never evaluated on, we assume it would have timed out. Equipped with this imputed data, we then employ CSHC (or another algorithm selector) to create a dispatch front-end, which, based on the features of the instance at hand, chooses one out of these very many parameterizations at runtime. We refer to this method as AVP-All ("AVP" for "accidental variance portfolio").

5.3.2 A Portfolio of Selected Parameterizations

The danger in the AVP-All approach is that the many parameterizations encountered while configuring a solver may pose a big challenge for the algorithm selector to pick the right parameterization. There is a danger that the selector may badly overfit the training data as it has too many classes to choose from. We therefore consider a modification of AVP-All where we reduce the set of parameterizations before training the algorithm selector: For each instance in the training set, there is one parameterization that has the best performance for that instance. A perfect oracle, a so-called virtual best solver (VBS), that chooses the best from the available parameterizations for each instance, would choose that parameterization for the respective instance. Consequently, by reducing the set of parameterizations to only those that are best for at least one training instance, our best possible *training* performance remains the same. However, we may have greatly facilitated the task for the algorithm selector to find the right instance, and may therefore boost training performance in this way. Our hope is that test performance will also improve as the selection procedure may now work more robustly as well. In our second approach, named AVP-VBS, we build an algorithm selector based only on parameterizations that are best for at least one instance in the training set.

5.3.3 Selecting the Portfolio

We now have three different strategies: The first is to train one parameterization for the training instances and to stick to it for all test instances. Alternatively, we can use the AVP-All or the AVP-VBS portfolios. Which method is best for a given solver and benchmark if instances will depend on how homogeneous the training instances are, and how well the instance features allow us to predict performance, as this determines how well the algorithm selectors can work with many constituent parameterizations. Consequently, as our last variant, we propose to build all portfolios at training time, and then choose the portfolio that exhibits the best performance on the training set. For IRACE, SMAC and GGA-EMC, we refer to this approach as Meta-IRACE, Meta-SMAC and Meta-GGA-EMC, respectively.

5.3.4 Experimental Results

In Table 5.4, we show the impact of building a portfolio from the parameterizations encountered during the tuning of a solver with three different solver configurators: IRACE v3.0, SMAC v3 0.8.0, and GGA-EMC. For these experiments, we extend the

Table 5.4: PAR10 Performance (# solved instances). Def (Default). Solved (ratio of all solved instances).

SAT	Solved	Def	IRACE	AVP$_{All}$	AVP$_{VBS}$	Meta$_{IRACE}$	SMAC	AVP$_{All}$	AVP$_{VBS}$	Meta$_{SMAC}$	GGA-EMC	AVP$_{All}$	AVP$_{VBS}$	Meta$_{GGA-EMC}$
BMC Test	281/302	346 (262)	166 (267)	168 (267)	146 (269)*	166 (267)	**121 (271)***	139 (270)	136 (270)	136 (270)	171 (267)	140 (270)*	140 (270)*	140 (270)*
BMC Train	566/684	446 (507)	143 (542)	149 (541)	153 (540)	143 (252)	97 (550)	93 (552)	91 (552)	91 (552)	132 (545)	100 (551)	119 (547)	100 (551)
CF Test	290/302	297 (276)	119 (280)	112 (281)*	130 (279)	119 (280)	89 (283)*	121 (281)	102 (282)	102 (282)	89 (283)	73 (285)	**72 (285)***	**72 (285)***
CF Train	289/299	305 (274)	68 (284)	79 (283)	90 (282)	68 (284)	67 (284)	57 (286)	37 (287)	37 (287)	48 (286)	38 (287)	37 (287)	37 (287)
IBM Test	232/302	113 (232)	22 (231)*	23 (231)	36 (230)	22 (231)*	21 (231)*	23 (231)	36 (230)	21 (231)*	**10 (232)***	11 (232)	34 (230)	**10 (232)***
IBM Train	300/382	108 (300)	43 (297)	53 (296)	43 (297)	43 (297)	32 (298)	35 (298)	32 (298)	32 (298)	41 (297)	52 (296)	51 (296)	41 (297)
GI Test	326/351	247 (307)	1330 (182)*	2104 (98)	2057 (103)	2104 (98)	167 (308)	173 (308)	163 (309)*	167 (308)	91 (317)	84 (318)	**75 (319)***	**75 (319)***
GI Train	957/1032	268 (898)	1468 (490)	1357 (526)	1469 (490)	1357 (526)	172 (937)	175 (904)	176 (903)	172 (937)	73 (937)	63 (941)	57 (943)	57 (943)
LABS Test	272/351	303 (250)	297 (246)	308 (245)	288 (247)*	288 (247)*	**230 (252)***	264 (249)	252 (250)	252 (250)	275 (248)	244 (251)	243 (251)*	243 (251)*
LABS Train	279/350	289 (258)	320 (250)	321 (250)	300 (252)	300 (252)	173 (264)	78 (273)	78 (273)	78 (273)	267 (255)	173 (264)	164 (265)	164 (265)
N-Rooks Test	351/351	116 (348)	116 (338)*	125 (338)	116 (338)*	116 (338)*	**5.8 (351)***	91 (351)	5.9 (351)	5.9 (351)	6.3 (351)	6.6 (351)	6.2 (351)*	6.2 (351)*
N-Rooks Train	484/484	145 (476)	48 (477)	64 (476)	49 (477)	48 (477)	4.9 (484)	88 (484)	4.5 (484)	4.5 (484)	5.6 (484)	5.5 (484)	5.3 (484)	5.3 (484)
MIP														
Assort Test	60/90	2429 (52)	2437 (52)	1332 (56)*	1335 (56)	1335 (56)	2150 (53)*	2158 (53)	2156 (53)	2150 (53)*	718 (58)	**712 (58)***	1885 (54)	718 (58)
Assort Train	55/89	50 (55)	342 (54)	128 (54)	78 (54)	78 (54)	117 (55)	122 (53)	122 (53)	117 (55)	65 (54)	69 (54)	71 (54)	65 (54)
CLS Test	50/50	3.04 (50)	2.99 (50)*	5.96 (50)	5.82 (50)	2.99 (50)*	2.49 (50)*	248 (50)	5.59 (50)	2.49 (50)*	**2.01 (50)***	7.66 (50)	6.27 (50)	**2.01 (50)***
CLS Train	50/50	3.14 (50)	3.1 (50)	6.12 (50)	5.83 (50)	3.1 (50)	2.4 (50)	257 (50)	5.11 (50)	2.4 (50)	1.96 (50)	7.2 (50)	4.54 (50)	1.96 (50)
COR-LAT Test	1000/1000	37 (999)	56 (998)*	59 (998)	59 (998)	56 (998)*	13 (1000)*	29 (1000)	23 (1000)	23 (1000)	**7 (1000)***	11 (1000)	11 (1000)	11 (1000)
COR-LAT Train	1000/1000	60 (984)	77 (979)	80 (979)	83 (979)	77 (979)	20 (998)	23 (999)	16 (999)	16 (999)	202 (946)	11 (1000)	11 (1000)	11 (1000)
MIRP Test	85/150	4995 (62)	8474 (45)	7845 (48)*	7845 (48)*	7845 (48)*	4984 (62)	5011 (62)	4808 (63)*	5011 (62)	5164 (61)	4970 (62)	**3357 (70)***	**3357 (70)***
MIRP Train	76/150	844 (55)	1149 (47)	922 (53)	922 (53)	922 (53)	727 (58)	574 (62)	608 (60)	574 (62)	724 (58)	689 (58)	646 (60)	646 (60)
RCW Test	863/915	562 (844)	702 (836)*	2198 (778)	2370 (769)	702 (836)*	730 (835)	702 (837)*	868 (828)	730 (835)	**344 (855)***	536 (845)	379 (853)	**344 (855)***
RCW Train	757/916	297 (700)	281 (702)	1301 (448)	1284 (453)	281 (702)	261 (708)	296 (703)	294 (703)	261 (708)	265 (708)	280 (705)	280 (705)	265 (708)
RCW2 Test	495/495	40 (495)	46 (495)*	463 (486)	330 (490)	46 (495)*	37 (495)*	51 (495)	42 (495)	37 (495)*	**32 (495)***	40 (495)	37 (495)	**32 (495)***
RCW2 Train	490/495	119 (475)	120 (475)	393 (435)	400 (434)	120 (475)	57 (485)	71 (485)	63 (485)	57 (485)	28 (490)	35 (490)	34 (490)	28 (490)

set of benchmarks with three MIP benchmarks from the AClib: BCOL-CLS, CORLAT and RCW2, each using the train/test splits provided as part of the benchmark.

IRACE and GGA-EMC have built-in paralelization and were run with 16 cores each, using seed 123456. For SMAC, to exploit the 16 parallel cores, we conducted 16 different tuning runs using different start seeds, as recommended by the SMAC developers in [84]. We choose the seeds consecutively starting at 123456. Additionally, to run SMAC with its full potential, we provided the instance features from the AClib and used the tool from `http://www.cs.ubc.ca/labs/beta/Projects/EPMs/` to compute the features for Assortment and MIRPLib.

After the individual tuning runs are completed, we then evaluated all 16 parameterizations returned by SMAC on all training instances (notice that during the tuning not necessarily all the parameterizations are executed on all the training instances) and selected the configuration with the lowest training PAR10 score to be run on the test set presented in the table. Note that this means that SMAC was given significant additional computation time when compared to the other approaches.

For each tuner, we noted with '*' the best of its variants on the test set, and, for each benchmark, we noted in bold the best overall approach on the test partitions. In the second and third column we give the number of instances for which at least one parameterization could find a solution within the time limit, as well as the total number of instances in the train and test partitions. For example, for the BMC test partition, 281 instances can be solved by SparrowToRiss if we chose dynamically the right parameterization for each instance out of the total set of parameterizations considered by any of the three tuners. With 302 instances in this partition, this means that 21 instances cannot be solved within the given time by SparrowToRiss, no matter which parameterization we use out of any that were ever considered by any tuner.

In the next column, we show the performance of the default parameterizations, first measured as PAR10 score over all instances for which at least one parameterization allows us to solve the instance, and, in brackets, the number of instances the default allows us to solve within the time limit.

In the following columns, broken down in three times four columns for each tuner, we show the performance of IRACE, SMAC, and GGA-EMC, as well as the portfolios derived from each tuner.

As the table shows, SMAC and GGA-EMC are effective tuners. Both provide substantial speed-ups over the default solver parameterizations for most benchmarks and solve more instances within the given time limit. This clearly indicates the value of configuration tools for solvers. To calibrate the impact, consider, for example, the CF benchmark. The default parameterization of SparrowToRiss solves 276 test instances within the time limit. Aspirationally, we may hope to solve 290 instances using this solver, as is, just by tuning. SMAC manages to tune SparrowToRiss to a point where it can solve 283 instances, the AVP-VBS portfolio generated by GGA-EMC even solves 285 instances. In other words, merely by tuning the solver and selecting a parameterization instance-specifically at runtime, we can solve 9 instances more than the default configuration, closing 64% of the default solver performance and its realistic maximum capability. At the 2020 SAT competition, a difference of 9 instances corresponds to ranking third or thirteenth in the competition. From our

own experience, we can state that the effect of tuning is roughly the same as one full year of algorithmic solver development time, which is very significant.

In Table 5.4, we can see that GGA-EMC performs on par with SMAC and slightly better than IRACE on SAT benchmarks. On MIP benchmarks, GGA-EMC is already clearly superior, even without building a portfolio. In terms of the portfolio versions of each tuner, we observe that SMAC benefits less from building a portfolio and only sees some slight boost on GI, MIRP and RCW. This result is somewhat surprising, as GGA (on BMC, CF, GI, LABS, N-Rooks, Assort, MIRP) and IRACE (on BMC, CF, LABS, Assort, MIRP) both clearly benefit from the ability to switch solver parameterizations at runtime. Moreoever, SMAC has been used in the past to build predictive ensembles when configuring the hyper-parameters of machine learning algorithms.[85]

One possible explanation for these results is that SMAC explores many more different parameterizations than GGA-EMC, but for each parameterization the experience of how it fares across the varying instances is more limited. GGA sees fewer parameterizations, but gains more experience on different instances with each one. This makes choosing one best parameterization at runtime more problematic for SMAC, as there are very many to choose from and each one is burdened with more imputed data for a lot of instances.

The reason why SMAC portfolios perform well in ML [85], on the other hand, is because the results from all parameterizations are aggregated rather than selected. That is to say, in an ML application, a portfolio is inherently associated with a stacking or bagging process, where an ensemble of weak learners can make good predictions. This is, however, not the case when configuring solvers. We have to choose and run exactly *one* highly efficient solver parameterization. In this context, using SMAC-provided parameterizations does not work well, and it is best to use SMAC's final parameters rather than switching these based on instance features.

Considering IRACE, we could expect a higher impact of building portfolios on top of this tuner since GGA-EMC and SMAC show there is more room for improvement. We recall that IRACE is built on the idea of omitting evaluations when a parameterization is already expected to perform worse than another on a given set of instances. However, the whole idea of a portfolio is to identify parameterizations that have niche high performance on some instances, while they may not work as robustly across sets of instances. When aiming to build a portfolio, IRACE suffers from the fact that it never gets to see the excellent niche performance of some parameterizations on some instances. Consequently, training a portfolio on this information is inherently limited by the data available to the portfolio trainer.

In terms of choosing whether to use always the same static, tuner-recommended parameterization or one of the two dynamically choosing portfolios, we see that the training performance is indeed an excellent criterion for making the final determination for GGA-EMC. Meta-GGA-EMC performs best on the training and test set on all but the Assort and COR-LAT benchmarks. Moreover, on these two benchmarks, Meta-GGA-EMC performs very close to the best GGA-EMC variant.

Encouragingly, Meta-GGA-EMC also shows a substantial speed-up on the challenging task of configuring CPLEX on the MIRPLib benchmark. Our tuner produces an approach that, when compared with the default in terms of PAR10 score, pro-

vides a 5 times performance improvement: 470 (with 70 instances solved within the time limit) versus 2431 (with only 62 instance solved). In comparison, IRACE is not able to find a parameterization that comes close to the quality of the default parameters, and SMAC is not able to improve on the CPLEX default performance.

Overall, if we consider it to be similar performance when the difference in solved instances is two or less, META-GGA-EMC performs best on all twelve benchmarks, whereby it is tied with SMAC on two thirds of them. On the remaining third, the difference in performance is striking:

- RCW: The default configuration solves 844 instances, the maximum we may realistically hope for on this test set with this solver is 863. If we consider these 19 instances to represent our total tuning entitlement, our META-GGA-EMC achieves almost 58% of the tuning potential. SMAC cannot find a parameterization that can compete with the default and actually loses over 47% of the potential by solving only 835 test instances within the time limit.
- MIRP: META-GGA-EMC realizes close to 35% of the tuning potential, SMAC none.
- Assort: META-GGA-EMC realizes 75% of the tuning potential, SMAC a mere 12.5%.
- GI: META-GGA-EMC realizes over 63% of the potential offered by tuning, SMAC a little over 5%.

Furthermore, META-GGA-EMC always achieves at least the performance of the default parameters of the given solver[2], while SMAC is unable to do so for 17% of the benchmarks we tested. We conclude that META-GGA-EMC exhibits a level or reliability and performance that tuners could not provide before.

Table 5.5: Comparison with state-of-the-art, test performances.

SAT	IRACE	Meta_{IRACE}	SMAC	Meta_{SMAC}	ISAC++	GGA-EMC	$\text{Meta}_{GGA-EMC}$
CF	119 (280)	119 (280)	89 (283)	102 (282)	89 (283)	89 (283)	**72 (285)**
IBM	22 (231)	22 (231)	21 (231)	21 (231)	21 (231)	**10 (232)**	**10 (232)**
BMC	166 (267)	166 (267)	**121 (271)**	136 (270)	147 (269)	171 (267)	140 (270)
GI	1330 (182)	2014 (98)	167 (308)	167 (308)	161 (309)	91 (317)	**75 (319)**
LABS	297 (246)	288 (247)	**230 (252)**	252 (250)	**230 (252)**	275 (248)	243 (251)
N-Rooks	116 (338)	116 (338)	**5.8 (351)**	5.9 (351)	39 (347)	6.3 (351)	6.2 (351)
MIP							
Assort	2437 (52)	1335 (56)	2150 (53)	2150 (53)	2714 (51)	**718 (58)**	**718 (58)**
MIRP	8474 (45)	7845 (48)	4984 (62)	5011 (62)	5607 (59)	5164 (61)	**3357 (70)**
RCW	702 (836)	702 (836)	730 (835)	730 (835)	571 (844)	**344 (855)**	**344 (855)**

This claim is further substantiated by a comparison of the new configurators GGA-EMC and Meta-GGA-EMC with the most competitive solver configurators

[2]Note that the tuners were not given access to these default settings and had to rediscover parameters that perform equally well or better, from scratch. This is the realistic scenario when developing a new tuner or after a substantial algorithmic update.

to date: IRACE, SMAC, GGA, their Meta-versions, and ISAC++[3]. We compare on benchmark sets where the default solver parameterization is unable to solve a significant number of instances in the test set. Table 5.5 shows that the new solver configuration tool works markedly better than the state of the art for the benchmarks considered in this chapter. Particularly for MIP problems, we see a significant improvement in performance. We believe that this makes Meta-GGA-EMC the currently best tool for configuring constraint satisfaction and optimization solvers.

5.4 Conclusions

We presented the combination of several contributions to boost the performance of GGA for combinatorial solvers significantly. Experiments on twelve challenging benchmarks from SAT and MIP demonstrated the effectiveness of the new tool, called META-GGA-EMC, that exhibits a level of reliability that could not be achieved before. On every benchmark we tested, the tuner was able to match, and usually outperform, the default parameterizations of the respective solver, while the best performing competitor was unable to do so on 17% of the benchmarks. Moreover, on 33% of the benchmarks tested, the improved tuning tool outperformed even the closest competitor, while it was tied in performance on the remaining benchmarks.

[3]Note that ISAC++ outperforms ISAC and, according to [86], Hydra and ISAC are comparable. We can therefore use ISAC++ to compare with ISAC and Hydra as well.

Chapter 6

PyDGGA: Distributed GGA for Automatic Configuration

In previous chapters, we have seen how setting parameters automatically dramatically reduces manual efforts and can result in orders of magnitude improvements in performance. Over the past decade there have been a number of methods developed for tuning parameters automatically, such as CALIBRA [43], ParamILS [44], I/F-Race [87], SMAC [30], ReACT/ReACTR [88, 89] and CPPL [90]. In chapter 5 we focused on GGA, and while previous work has explored parallel algorithm configuration in the context of ParamILS, SMAC [91] and grid search [92], we showed that the inherent potential for parallelization [93] of genetic algorithms makes them a good candidate to tackle AAC.

In this chapter we present PYDGGA, a distributed version of GGA written in Python, adapted to exploit the resources of High Performance Computing (HPC) clusters. PYDGGA implements all the features that improved the original GGA in Chapter 5. In what follows, we first describe the distributed architecture and present other implementation details not introduced previously. Then, we provide brief instructions on how to use PYDGGA. Finally, we experiment on SAT problems and conclude.

6.1 Distributed Architecture

To adapt GGA to a distributed computing architecture while preserving the core algorithm as close as possible to the original description, PYDGGA is implemented using an event-driven architecture, which is known to be good for horizontal scalability. The events represent steps from the original GGA, such as the generation of new offspring or the evaluation of a genome on an instance. Each event has the necessary information attached to it to perform its associated action and triggers the next event in a way so as to maintain the original GGA execution logic. To exploit the available computing resources to the fullest extent, PYDGGA uses a master-worker architecture, shown in figure 6.1, to distribute the genome-instance evaluations across several machines. The master runs the event-based core, and the workers wait for parameters and instance data to evaluate and return the re-

sult. The workers have none of GGA's logic and can be added or removed at any time. The master will simply use the workers still available and rollback incomplete evaluations, if necessary.

One can find examples of evolutionary algorithms based on a steady-state model that make maximal use of many parallel processors using a master-worker architecture. However, it has been shown that they tend to be biased towards regions of the search space [94]. Our contribution preserves GGA's generational scheme, which has been shown to be effective for automatic configuration. Moreover, we decided to use this approach instead of just relying on a batch-queuing system, such as SGE or SLURM, because these systems are used by multiple users concurrently and their tasks are interleaved, which add an almost negligible delay when only a handful of tasks are to be executed. But, since PyDGGA tries to run as many evaluations as possible, this delay ends up being a burden. The worker approach lets PyDGGA run on any distributed environment regardless of the batch-queuing system, as long as there is a shared file system. This ensures computing resources are reserved for a longer period and allows the user to terminate and re-submit more workers later to release resources for other jobs temporarily.

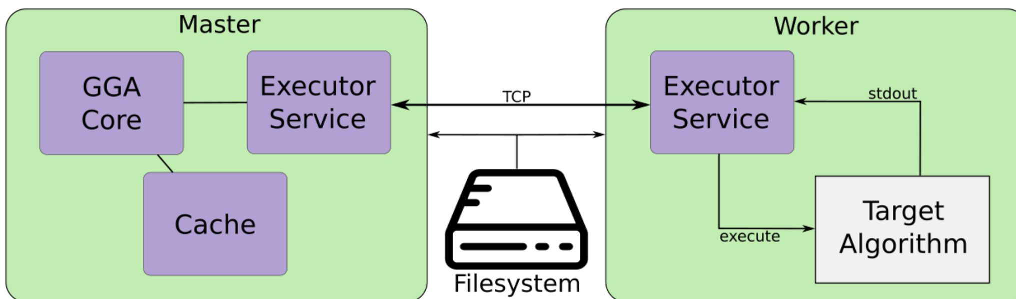


Figure 6.1: Master-Worker architecture

6.2 Scheduling & Canceling

A problem arises when we can run evaluations of different generations distributively at the same time. Which should be run first? It makes sense to run the evaluations in an order relative to the generation they belong to. This way we keep on fulfilling the dependencies of later generations, which trigger more evaluations, thus the hardware rarely idles. However, this static order may break the efficiency of the original GGA racing scheme.

GGA uses mini-tournaments with size equal to the number of CPU cores on a single machine, and run the evaluation of each individual on a different core. Then, as soon as an individual could be declared the winner it simply cancels the evaluations of the other individuals in the mini-tournament that have not been started yet.

PyDGGA can handle many more resources than GGA and can evaluate entire mini-tournaments at the same time, which means that it will waste time running evaluations that GGA would have skipped. To tackle this issue, PyDGGA keeps

a dynamic priority value that determines the next evaluation to run. How this value is to be computed to maximize the efficiency is still an open question, but so far the implementation tries to mimic the racing scheme behaviour. Finally, since the scheduling is not perfect, we know that PyDGGA will start some evaluations that will end up being unnecessary. To mitigate this, we also implement a way for PyDGGA to terminate running evaluations.

6.3 Tool Enhancements

On top of the more profound changes commented above, we also introduce some additional modifications to make PyDGGA more user-friendly.

- **Stop/Resume**: PyDGGA keeps a cache of all the evaluations performed so far inside the scenario directory. If the same scenario is used again it will reuse the cache whenever possible, which has the effect of resuming the search from wherever it was stopped as all the evaluations in the simulation graph up to that point are resolved instantly.
- **Enhanced configurations constraints**: GGA allows the user to specify combinations of forbidden values, but it could only express constraints, such as $a = 10$ & $b = 5$ as forbidden, which forced the user to write the Cartesian product of all the forbidden parameter-value combinations. PyDGGA uses Python's abstract syntax tree module, which lets the user write Python logical expressions that must be satisfied (True) by all valid configurations, for example: $10 <= a < 20$ and b in $[5, 6, 7]$.
- **Abort Search**: GGA only supports two possible evaluation results: SUCCESS and CRASHED. The first denotes that the evaluation was successful and the second captures cases where A failed, but are not critical, for example because it run out of memory. While CRASHED works fine in most situations, there are others that leave the user waiting for the algorithm to finish just to realize at the end that all the evaluations CRASHED. As an example, imagine that the instances or the target algorithm binary are moved while PyDGGA is running, or imagine that the user decides to abort if the program detects something wrong happened that may indicate that the result is not correct and the same error may arise in the rest of the executions. For these situations, we add the evaluation result ABORT, which stops PyDGGA immediately.
- **Objective function**: GGA was designed with runtime tuning in mind. PyDGGA extends this to support a different type of objective function. Namely, the user can pass any value as the evaluation metric (including the runtime) and PyDGGA will try to configure the target algorithm for that metric.

6.4 Using PyDGGA

PyDGGA is available as a command-line tool from `https://ulog.udl.cat/?page_id=30`. There one can download a pre-built binary, the user manual and some

examples. For the sake of brevity, we do not describe the details of the whole process, instead, we explain what a scenario is and show that running PyDGGA locally or in a distributed environment is quite similar. We encourage the reader to follow the complete example on how to tune the SAT solver glucose 4 in the user manual.

A configuration scenario for PyDGGA is just a directory with some special files that contain the information required to configure the target algorithm. These files are:

- **conf.xml:** This file describes the parameter structure of A, as a tree. Additionally, it may also contain the so-called seed genomes, i.e, the default solver parameter's values, and constraints to filter forbidden parameterizations.
- **instances.txt:** A simple text file that contains the instances that form Π. Each line of the file contains the path to an instance and the seed that the target algorithm should use to initialize the pseudo-random number generator of A when evaluating that instance.
- **settings.txt:** The configuration of PyDGGA itself, such as the number of generations, the size of the population, etc. It also contains the name of the wrapper file.
- **wrapper file:** This could be the target algorithm A or a script that acts as the interface between PyDGGA and A.

Once a scenario is set up, running or testing it is as simple as running the following command to start PyDGGA locally:

```
pydgga gga -s "/path/to/scenario_dir"
```

If the scenario works locally, it is almost ready for use in a distributed environment. The only additional element is a script that PyDGGA will invoke any time it needs to start a new worker. For example, to run it on an environment that uses **qsub** to submit jobs, the script could be:

```
1 #!/usr/bin/env sh
2
3 QUEUE="yourqueue.q"          # System specific configuration
4 PENV="smp"
5 MEM_LIMIT="35840M" # 35 GB
6 RT_LIMIT=172800 # 2 Days
7
8 name=${1}  # session name      | Extract fixed parameters
9 slots=${2} # number of slots   | passed by pydgga
10 shift 2    # remove 'name' and 'slots' from ${@}
11
12 olog="/path/to/stdout/directory"
13 elog="/path/to/stderr/directory"
14
15 echo "pydgga dggaw ${@}" | qsub -V -cwd -pe ${PENV} ${slots}\
16     -l h_vmem=${MEM_LIMIT} -l h_rt=${RT_LIMIT} -q ${QUEUE}   \
```

```
17     -N ${name} -o "${olog}" -e "${elog}"
18
19 exit 0
```

Then to run the same scenario distributed, one simply runs:

```
pydgga dgga -s "/path/to/scenario_dir" --worker-script "/path/to/script
        " --slots SLOTS_PER_WORKER --num-workers NUM_WORKERS
```

6.5 Experimental Results

In this section we conduct some experiments to showcase that PyDGGA can outperform the default parameters on several SAT scenarios. We focus on minimizing the runtime of a SAT solver. The experiments are conducted in a compute cluster with nodes equipped with two octo-core Intel Xeon Silver 4110 @ 2.10 GHz processors and 96 GB of RAM. The selected solver is the award-winning SparrowToRiss [1], which has a large configuration space with 222 parameters open for configuration. The instances come from the industrial and crafted benchmarks used in [77]: Bounded Model Checking (BMC), Circuit Fuzz (CF), IBM-Hardware Verification, Graph Isomorphism (GI), and N-Rooks, which are all available, including the train/test splits, in the algorithm configuration library [78].

To configure the solver, we let PyDGGA run for 2 days. In both the training and test phases we use a time limit of 300 seconds and 5 GB per evaluation. The results of our evaluation are shown in Table 6.1, which show that PyDGGA can find better parameteriztions than the default one for SparrowToRiss on all the evaluated SAT benchmarks. The cost metric employed is PAR10, which is defined as the time needed to solve the instance if solved within the time limit, otherwise the run is penalized with a value 10 times the time limit. We report the results using the PAR10 metric as well as the number of solved instances. Finally, to make the PAR10 value more readable we remove the constant value post hoc added by instances that are never solved by any configuration.

Table 6.1: PAR10 performance (# solved instances) on the test instances

	BMC	CF	IBM	GI	N-Rooks
Default	346 (262)	297 (276)	113 (232)	247 (307)	116 (348)
PyDGGA	**171 (267)**	**89 (283)**	**10 (232)**	**91 (317)**	**6.3 (351)**

6.6 Conclusions

PyDGGA is able to exploit the resources of a distributed computing environment. Experiments using the SAT solver SparrowToRiss demonstrated that it can boost the performance of an algorithm by automatically finding a parameterization that yields better results than the default one.

Chapter 7

OptiLog: A Framework for SAT-based Systems

Python [95] has emerged as one of the most preferred programming languages for rapid prototyping of applications because of its straightforward syntax and the great amount of established libraries that provide common functionality for researchers to readily use. We can find several of these libraries into diverse Artificial Intelligence disciplines like, for example, Numpy[96], Pandas[97], scikit-learn[98], Pytorch[99] or Keras[100].

In terms of performance, the core of the critical components of these systems is implemented with more efficient languages such as C++, although their interconnection is commonly materialized through Python.

Within the area of Constraint Programming, Python has also become quite popular. CPLEX[17], Gurobi[101], OR-Tools[102], COIN-OR[103], SCIP[104], Z3[105] and many others have Python bindings. In particular, in the SAT community there have also been several contributions. PySAT [106] was the first framework, to our best knowledge, to provide Python bindings for several SAT solvers.

Recently, there have been other contributions that can be queried from Python such as SAT Heritage [107], intended to serve as an archive and to easily compile and run all SAT solvers that have been released so far, or *cnfgen* [108], that produces *hard* SAT benchmarks coming from research in Proof Complexity.

Our contribution in OptiLog is two-fold. First, we provide a Python binding [109] for the PBLib [110] that allows to encode Pseudo Boolean (PB) constraints into SAT. This binding is also currently integrated into PySAT.

Second, we take a step further, easing both the integration of new C++ SAT solvers in OptiLog and their end usage into practical environments.

We isolate the development of C++ SAT solvers so that by implementing the iSAT C++ interface OptiLog gently incorporates the new SAT solver. In contrast, PySAT requires the user to write some ad-hoc additional Python code plus the Python bindings. The iSAT interface is inspired by the C interface IPASIR (Reentrant Incremental Sat solver API, in reverse) [111] and the PySAT interface.

To optimize the end-SAT-based system, the end-user is commonly forced to play by hand with a non-negligible amount of adjustable parameters coming from the solvers or encoders it uses. Automatic configurators should have to be used in this

context. Unfortunately, it takes a while to become familiar on how to create the configuration of the scenarios, which is usually a source of countless bugs. OptiLog get rids of all this complexity and automatically generates all the pieces needed for the configuration, delivering a ready-to-tune application.

There have been a number of methods developed for tuning parameters automatically, such as CALIBRA [43], ParamILS [44], I/F-Race [32, 87], SMAC [30] and GGA [18, 45]. OptiLog currently provides support for SMAC and GGA.

This chapter is structured as follows. First we present the OptiLog framework with detail about the most important implemented modules, and how a new SAT solver can be integrated into it. Then, we present a comprehensive example of the framework. Finally, we provide some closing thoughts.

7.1 OptiLog Framework Architecture

The general architecture of OptiLog is described in Figure 7.1. Four main modules compose the end-user OptiLog API, which we briefly describe in the following subsections: the Formula module, the SAT Solver module, the PB Encoder module and the Automatic Configuration (AC) module. Additionally, new C++ SAT solvers can be integrated into OptiLog by implementing the C++ iSAT interface. Full details can be found in the OptiLog manual accessible from [112].

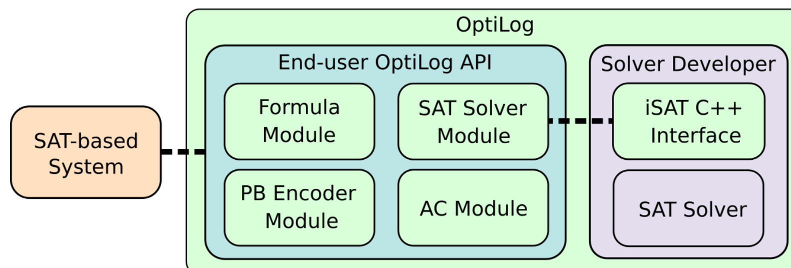


Figure 7.1: OptiLog's architecture.

7.2 Formula Module

The Formula module is designed to ease the implementation and manipulation of boolean formulas. As such, two specific classes are created: *CNF* (for the typical Conjunctive Normal Form) and *WCNF* (for the Weigthed CNF version). These Formulas have the common functionality of setting new variables, adding clauses and exporting to the DIMACS file format.

CNF: The CNF class provides the traditional representation of a Conjunctive Normal Form fomula, a conjunction of clauses defined as disjunctions of literals. In code, clauses are provided as lists of integers.

WCNF: The WCNF class provides the interface for partial and weighted partial CNF formulas. In this case, clauses can be added with a weight. If this is the case, these clauses are considered *soft*. *Hard* clauses are added without weight or by specifying the weight `INF_WEIGHT` that represents ∞.

As an example, CNF Formula $(x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$ and WCNF Formula $(x_1 \vee x_2, 1) \wedge (x_3 \vee \neg x_2, \infty)$ would be implemented as follows

```
1 from optilog.sat import CNF, WCNF
2 cnf = CNF()
3 cnf.add_clauses([[1, 2], [3, -2]])
4 wcnf = WCNF()
5 wcnf.add_clause([1, 2], weight=1)
6 wcnf.add_clause([3, -2]) # equivalent to weight=WCNF.INF_WEIGHT
```

Aside from the typical formula manipulation methods, OptiLog provides additional methods. In particular, it provides explicit functions `load_{cnf|wcnf}` from the `optilog.loaders` Python module. These functions allow to load the formula directly into a SAT solver.

7.3 SAT Solver Module

OptiLog is inspired on the interfaces of IPASIR [111] and PySAT. The behaviour of some functions can slightly change, see the manual [112] for details. The solvers currently integrated in OptiLog are: Cadical[113], Glucose 4.1 and Glucose 3.0 [114], Picosat[115], Minisat[116] and Lingeling 18 [117]. Not all the solvers implement all the methods in the iSAT API, the only one that fully does is a modified version of Glucose 4.1. Here, we briefly describe some of the additional methods that we incorporated into the iSAT API and that are currently supported by the modified version of Glucose 4.1 delivered with the OptiLog tool.

solver.set & solver.get: Used to set and get the value of parameters that modify the behaviour of the solver.

solver.set_decision_var: Used to set whether the input variable can be used as a decision variable.

solver.set_static_heuristic: Used to set an static decision heuristic.

solver.solve_hard_limited: Solves the current formula with a strict budget in terms of conflicts or propagations. This method does not wait for the current restart to end.

solver.learnt_clauses: This method returns the learnt clauses that are currently in the solver including learnt unit clauses.

7.4 PB Encoder Module

The PB Encoder module currently integrates the Python binding for PBLib we developed for this project, which provides the access to PB and Card encoders, some of them incremental. It also incorporates the Totalizer incremental encoder implemented in Python in PySAT. The user can transparently create PB/Card constraints that are automatically encoded through PBLib and PySAT Card functions into a set of SAT clauses. If all coefficients (weights) in the constraint are equal to 1, Card constraint encoders are applied.

```
1 from optilog.sat.pbencoder import IncrementalEncoder
2 L = [1,2,-3]
```

```
3 W = [4,3,3]
4 encoder, max_var, C = IncrementalEncoder.init(
5     lits=L, bound=7, weights=W, max_var=3, encoding="seqcounter")
```

Lines 2-5 in the above example show how to encode the PB constraint $4 \cdot x + 3 \cdot y + 3 \cdot \neg z \leq 7$ through an incremental encoder into SAT using OptiLog. Currently, we only support PB constraints with positive coefficients[1].

Function *IncrementalEncoder.init* takes as input the list of literals L, the bound, the list of weights W, the maximum variable and the encoding to be used. It returns an encoder object that can be used to refine the upper bound, the maximum variable used by the encoder and the list of clauses that encode the constraint C. In our example, to refine the upper bound to ≤ 6 we can use the command `max_var, C = encoder.extend(6)`, which returns the clauses C to force the new upper bound and the maximum variable used in C.

The possible encodings supported in PBLib for incremental encoding are *bdd* and *card* for cardinality constraints and *seqcounter* and *adder* for PB. PySAT Card supports *totalizer* for cardinality constraints. All these encodings are available in *IncrementalEncoder* through the parameter *encoding* in the *init* method. By default PBLib automatically overrides the user selected encoding when it detects it can generate too many clauses. In contrast, OptiLog always applies the encoding selected by the user.

7.5 Automatic Configuration (AC) Module

The AC module provides an API to generate configuration scenarios for AC tools. An AC tool searches for a setting, to the configurable parameters of a *target* function (algorithm), that optimizes some objective function or run time on a set of instances (data) under different seeds. We present the module features:

```
1 import random
2 from optilog.autocfg import ac, Bool, Int, Real, Categorical, CfgCall
3 from optilog.autocfg.configurators import SMACConfigurator
4 @ac
5 def func1(
6     x, data, p1: Bool() = True, p2: Real(-1.3, 2) = 0,
7     p3: Int(-5, 5) = 0, p4: Categorical("A", "B", "C") = "A"):
8     ...
9 @ac
10 def func2(
11     data, seed, l_func1: CfgCall(func1), n: Int(1, 10) = 1):
12     random.seed(seed)
13     res = n * l_func1(random.randint(20,30), data)
14     print("Result:", res)
15     return res
16
17 configurator = SMACConfigurator(
18     func2, global_cfgcalls=[func2], runsolver_path='./runsolver',
19     input_data=['path1', 'path2', 'path3'],
```

[1] We will add in short a normalization step for general PB constraints.

```
20     data_kwarg='data', seed_kwarg='seed',
21     run_obj='quality', cutoff=30, time_limit_sec=43200,
22     quality_regex=r"^Result: (\d+)$")
23 configurator.generate_scenario('./scenario')
```

Configurable parameters: Leveraging Python's type hints we can specify the type, domain and default value of the parameters to configure. For example, the AC module will recognize four configurable parameters in *func1* ($p1, p2, p3, p4$), where parameter $p3$ is of type `optilog.autocfg.Int`, and will collect the annotated information for creating the configuration scenario.

Configurable functions: The AC module allows to gather the configurable parameters of a configurable function (decorated with *@ac*). All calls to the same *global* CfgCall function will share the same values for the configurable parameters, while calls to *local* CfgCall functions can have different values. In the example, *func2* is *global* while *l_func1* is a *local* call to *func1*.

Configuration scenario: class *SMACConfigurator* is used to automatically generate the scenario for the SMAC configurator. It receives as parameters: (l. 18) the entry point $func2$ (i.e., the function that SMAC will call), the list of *global* configurable functions [$func2$] (notice that in our example $func2$ is itself configurable), the path to the *runsolver* tool, (l. 19) the list of input data (which is printed, item by item, to a text file and used by SMAC as the description of the set of instances where the function to be tuned will be evaluated), (l. 20) the parameters (*data_kwarg*, *seed_kwarg*) that will use the AC tool to send the data and seed to the entry point on which the current configuration will be evaluated, (l. 21) the objective is set to *quality* in order to minimize the result of the entry point (*runtime* is another possible objective), a set of parameters related to the automatic configuration process (*cutoff*, *time_limit_sec*), and (l. 22) the regular expression to extract the quality reported to the AC tool.

7.6 Adding SAT Solvers to OptiLog through iSAT Interface

OptiLog automatically generates bindings to C++ SAT solvers that implement the iSAT abstract interface. In order to integrate a new SAT solver, the solver source code has to be included into the compilation pipeline and an implementation to the abstract iSAT interface has to be provided.

The `Extern/sat` directory contains the source code of the SAT solver. For example, in `Extern/sat/glucose41` we find the source code for Glucose 4.1.

The `Module/sat` directory contains the implementation for the iSAT interface. In particular, the files `solver.{cpp|hpp}` define the implementation of the iSAT abstract interface. These files contain macros that will be used to automatically generate Python bindings. In the Glucose 4.1 example, the implementation of the interface is located in `Module/sat/glucose41`.

All the process described above is automatically performed by executing the `new_solver` script provided by OptiLog.

7.7 Using OptiLog

SAT-based MaxSAT algorithms reformulate the MaxSAT optimization problem into a sequence of SAT decision problems. Each SAT instance of the sequence encodes whether there exists an assignment with a cost $\leq k$, encoded as a PB or Card constraint depending on the weights of the soft constraints. SAT instances with a k less than the optimal cost are unsatisfiable, the others being satisfiable. In particular, the subclass of model-guided algorithms iteratively refine (decrease) the upper bound and guide the search with satisfying assignments (models) obtained from satisfiable SAT instances.

Left hand side of Program 1 shows an implementation of the Linear algorithm [118, 119], a SAT-based model-guided algorithm for Weighted MaxSAT formulas, with OptiLog. The *linear* function takes as parameters the path to the Weighted MaxSAT instance in DIMACS format and the seed (lines 7, 8). Lines 10-12 create the incremental SAT solver, set its seed and load the hard clauses directly into the solver while the soft clauses are stored in the WCNF formula f.

Lines 15-19 make a relaxed copy of the soft clauses (adding a new blocking variable per clause) that is added to the SAT solver. Line 22 creates an incremental PB constraint on the blocking variables B that uses as coefficients W, the weights of the soft constraints, and the initial upper bound ub as the independent term. It retrieves the set of initial SAT clauses C for the PB encoding (added to the SAT solver in line 24), the max_var auxiliary variable used in the encoding and the object *encoder* through which we will be able to generate additional SAT clauses to further restrict the constraint (see line 27).

Lines 26-32 conform the main loop of the algorithm. The new clauses to extend the incremental PB constraints are generated and added (lines 27,28). Line 29 calls the SAT solver and, if the current SAT instance is satisfiable, the model is retrieved using its cost to refine the upper bound (lines 30,31).

Right hand side of Program 1 shows how the definition of the linear function has to be changed so that it can be automatically configured. There are, in particular, two main configurable aspects: the SAT solver and the PB encoder to be used plus their respective adjustable parameters.

Instead of initializing the SAT solver in line 10, we use the configurable function *get_glucose41* that returns a configured *Glucose41* solver.

The other aspect to be configured is the incremental encoder that we are using. We add a configurable categorical parameter called *encoding* (line 8), which is passed to the *init* method of *IncrementalEncoder* in line 22.

The following lines show how the SMACConfigurator object is created. Line 7 is used to report the quality to the AC tool and line 8 is used to specify the default quality when there is a *crash* such as a system timeout or memout.

```
1 configurator = SMACConfigurator(
2     linear, runsolver_path="./runsolver", global_cfgcalls=[linear],
3     input_data=["inst1.wcnf", "inst2.wcnf", ..., "instN.wcnf"],
4     data_kwarg="instance", seed_kwarg="seed",
5     cutoff=30, memory_limit=6 * 1024,
6     wallclock_limit=43200, run_obj="quality",
7     quality_regex=r"^o (\d+)$",
```

Program 1 Linear MaxSAT algorithm implemented with OptiLog (left) and modifications required to the same implementation to enable its automatic configuration (right). The imports for *IncrementalEncoder* and *load_wcnf* are omitted in the automatic configuration example.

```
1  from optilog.sat import Glucose41
2  from optilog.sat.pbencoder import IncrementalEncoder
3  from optilog.loaders import load_wcnf
4
5
6  def linear(
7      instance,
8      seed
9  ):
10     s = Glucose41()
11     s.set('seed', seed)
12     f = load_wcnf(instance, s)
13     B, W, max_var = [], [], f.max_var()
14
15     for w, c in f.soft_clauses:
16         max_var += 1
17         s.add_clause(c + [max_var])
18         B += [max_var]
19         W += [w]
20
21     res, ub = True, f.top_weight()
22     encoder, max_var, C = IncrementalEncoder
23                             .init(B, ub, W, max_var)
24     s.add_clauses(C)
25
26     while res is True and ub > 0:
27         max_var, C = encoder.extend(ub - 1)
28         s.add_clauses(C)
29         res = s.solve()
30         if res is True:
31             ub = f.cost(s.model())
32             print("o", ub)
33
34     return ub
```

```
1  from optilog.autocfg import ac, Categorical, CfgCall
2  from optilog.autocfg.sat import get_glucose41
3  @ac
4  def linear(
5      instance,
6      seed,
7      init_solver_fn: CfgCall(get_glucose41),
8      encoding: Categorical('best', 'adder', 'seqcounter') = 'best'
9  ):
10     s = init_solver_fn(seed=seed)
11
12     f = load_wcnf(instance, s)
13     B, W, max_var = [], [], f.max_var()
14
15     for w, c in f.soft_clauses:
16         max_var += 1
17         s.add_clause(c + [max_var])
18         B += [max_var]
19         W += [w]
20
21     res, ub = True, f.top_weight()
22     encoder, max_var, C = IncrementalEncoder
23                             .init(B, ub, W, max_var, encoding)
24     s.add_clauses(C)
25
26     while res is True and ub > 0:
27         max_var, C = encoder.extend(ub - 1)
28         s.add_clauses(C)
29         res = s.solve()
30         if res is True:
31             ub = f.cost(s.model())
32             print("o", ub)
33
34     return ub
```

```
8     cost_for_crash=(2 << 64) - 1,  # Max sum WCNF weights
9 )
10 configurator.generate_scenario("./scenario")
```

7.8 Experimental Results

We experimented with the configurable version of the Linear algorithm on a computer cluster with 2.1GHz cores. As benchmarks, we used the set of 600 instances from the complete weighted track of the MaxSAT 2020 evaluation[120].

We executed SMAC in parallel with 32 runs (one of them with the default configuration of Glucose41 and PB encoder). In 5 out of the 32 runs, SMAC was able to find a better configuration than the default. These 5 runs provide suboptimal values for 446, 445, 443, 443 and 424 instances, while the default only on 388. Curiously, 2 out of the 5 best runs (443, 424) set the PB encoder to *adder* (default value is *best*). The rest of the changes are applied on the Glucose41 parameters. This is a sign of the benefit of using AC tools even on systems that combine several pieces that already have good default parameters.

7.9 Conclusions

The SAT community has generated amazing tools that we need to make more accessible to our and other communities. OptiLog contributes in this sense, easing the access to solvers and encoders, providing the iSAT interface that could become the basis for an standard SAT API., and the AC module that can potentially be applied to tune any Python function.

Chapter 8

Conclusion and Future Work

We introduced the hyper-configurable versions of the dialectic search and rective tabu search algorithms, and showed that automatic configuration could be used to configure them for the MaxSAT problem. The resulting algorithms worked robustly on random and crafted instances and were used to extend state-of-the-art solvers that excelled in the industrial category, providing a solution that worked robustly in all categories.

Inspired by the previous results, we then shifted towards improving the automatic algorithm configurator GGA. The proposed solutions were evaluated with satisfactory results, and our efforts boosted GGA's performance significantly. In addition, the proposed architecture based on simulating the evolution of the genetic algorithm could be used on other problems tackled using genetic algorithms.

Another attempt to boost the results of automatic configuration, lead us to explore how could we use the intermediate results explored by the automatic configuration algorithm, what we call the accidental variance of the search procedure. We provided a solution that uses simple imputation and filtering methods to build a portfolio of these configurations, and showed not only that it boosted the result but also that we could decide which of the different solutions should be used on the test data by looking at the results on the training data.

Focusing on the objective to bring the technology developed in this thesis to other communities and the industry, we made the improvements on GGA publicly available as PyDGGA, a new tool for automatic configuration specially suited to exploit the resources of high performance computing clusters.

Finally, continuing with our efforts to make automatic algorithm configuration more accessible, we presented OptiLog, a tool that eases the integration of SAT and automatic configuration technologies.

We have explored a variety of topics about algorithm configuration and answered many questions, but there remain open questions, as well as a significant amount of work to bring the technology to other research areas and the industry.

- Explore tuning other dynamic metaheuristic search features. The results proved that the configured search features were good candidates to guide the local search algorithms, but there could be other, even better, features that could improve the achieved results.

- Apply automatic configuration to other metaheuristic frameworks. In this thesis we obtained good results with dialectic search and reactive tabu search. But we could expect other frameworks, based on Hill climbing, Simulated annealing or Ant Colony optimization, to achieve similar or better results.

- Include a surrogate model into the new distributed approach for automatic algorithm configuration. One one hand, we distributed GGA and showed that the new version was superior. On the other, surrogate models have been used separately to improve the results of GGA and other algorithms. Hence, although distributing a surrogate model is a challenging task, we have reasons to believe that it would prove beneficial.

- Study other imputation methods to build better portfolios that exploit the accidental variability inherent to the search procedure of automatic configuration algorithms. In the machine learning literature we can find other, more sophisticated, imputation mechanisms that could be helpful for our approach.

- Try different algorithm selection algorithms to exploit the different configurations explored during the automatic algorithm configuration procedure. Our approach employed ISAC++, but other alternatives, for example from the Open Algorithm Selection Challenge [121], could yield other interesting results

- Improve PyDGGA usability by means of zero-configuration networking protocols. If the network configuration allows it, these protocols would let the tool find itself and free the user from the task of manually specifying ports and IPs

- Extend the OptiLog framework with additional solvers, i.e., MaxSAT solvers, and integrate random and crafted instance generators.

- Provide support for callback functions into OptiLog, as in Gurobi [101], that could be applied on critical points: restarts, pick literal decision, conflict analysis, etc.

Bibliography

[1] A. Balint and N. Manthey, "SparrowToRiss," in *Proceedings of SAT Competition 2014* (A. Belov, D. Diepold, M. J. Heule, and M. Järvisalo, eds.), vol. B-2014-2 of *Department of Computer Science Series of Publications B*, p. 77, University of Helsinki, Helsinki, Finland, 2014.

[2] X. Li, M. J. Garzarán, and D. A. Padua, "Optimizing sorting with machine learning algorithms," in *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pp. 1–6, IEEE, 2007.

[3] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001. New Trends in High Performance Computing.

[4] C. Audet and D. Orban, "Finding optimal algorithmic parameters using derivative-free optimization," *SIAM J. Optim.*, vol. 17, no. 3, pp. 642–664, 2006.

[5] J. Cavazos and M. F. P. O'Boyle, "Automatic tuning of inlining heuristics," in *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, p. 14, IEEE Computer Society, 2005.

[6] E. A. Brewer, "High-level optimization via automated statistical modeling," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995* (J. Ferrante, D. A. Padua, and R. L. Wexelblat, eds.), pp. 80–91, ACM, 1995.

[7] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, February 5-8, 2009 - Volume 1* (A. Ranchordas and H. Araújo, eds.), pp. 331–340, INSTICC Press, 2009.

[8] M. Feurer, K. Aaron, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems* (C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds.), vol. 28, Curran Associates, Inc., 2015.

[9] L. Zimmer, M. Lindauer, and F. Hutter, "Auto-pytorch tabular: Multi-fidelity metalearning for efficient and robust autodl," *CoRR*, vol. abs/2006.13799, 2020.

[10] M. Stillger and M. Spiliopoulou, "Genetic programming in database query optimization," in *Proceedings of the 1st Annual Conference on Genetic Programming*, (Cambridge, MA, USA), p. 388–393, MIT Press, 1996.

[11] Y. Diao, F. Eskesen, S. Froehlich, J. L. Hellerstein, L. Spainhower, and M. Surendra, "Generic online optimization of multiple configuration parameters with application to a database server," in *Self-Managing Distributed Systems, 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2003, Heidelberg, Germany, October 20-22, 2003, Proceedings* (M. Brunner and A. Keller, eds.), vol. 2867 of *Lecture Notes in Computer Science*, pp. 3–15, Springer, 2003.

[12] C. Thachuk, A. Shmygelska, and H. H. Hoos, "A replica exchange monte carlo algorithm for protein folding in the HP model," *BMC Bioinform.*, vol. 8, 2007.

[13] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu, "Boosting verification by automatic tuning of decision procedures," in *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pp. 27–34, IEEE Computer Society, 2007.

[14] B. A. Tolson and C. A. Shoemaker, "Dynamically dimensioned search algorithm for computationally efficient watershed model calibration," *Water Resources Research*, vol. 43, no. 1, 2007.

[15] S. Kadioglu and M. Sellmann, "Dialectic search," in *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings* (I. P. Gent, ed.), vol. 5732 of *Lecture Notes in Computer Science*, pp. 486–500, Springer, 2009.

[16] R. Battiti and G. Tecchiolli, "The reactive tabu search," *ORSA Journal on Computing*, vol. 6, no. 2, pp. 126–140, 1994.

[17] International Business Machines Corporation, "IBM ILOG CPLEX: High-performance software for mathematical programming and optimization," 2021.

[18] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings* (I. P. Gent, ed.), vol. 5732 of *Lecture Notes in Computer Science*, pp. 142–157, Springer, 2009.

[19] C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, "Reactive dialectic search portfolios for maxsat," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA* (S. P. Singh and S. Markovitch, eds.), pp. 765–772, AAAI Press, 2017.

[20] C. Ansótegui, B. Heymann, J. Pon, M. Sellmann, and K. Tierney, "Hyper-reactive tabu search for maxsat," in *Learning and Intelligent Optimization - 12th International Conference, LION 12, Kalamata, Greece, June 10-15, 2018, Revised Selected Papers* (R. Battiti, M. Brunato, I. S. Kotsireas, and P. M. Pardalos, eds.), vol. 11353 of *Lecture Notes in Computer Science*, pp. 309–325, Springer, 2018.

[21] C. Ansotegui, J. Pon, and M. Sellmann, "Boosting evolutionary algorithm configuration," *Annals of Mathematics and Artificial Intelligence*, 2021.

[22] C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, "PyDGGA: Distributed GGA for Automatic Configuration," in *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021*, Submitted for publication.

[23] C. Ansótegui, J. Ojeda, A. Pacheco, J. Pon, J. M. Salvia, and E. Torres, "OptiLog: A Framework for SAT-based Systems," in *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021*, Submitted for publication.

[24] J. Argelich, C. Li, F. Manyà, and J. Planes, "MaxSAT Evaluation 2016," 2016. `http://www.maxsat.udl.cat/16`.

[25] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA* (M. A. Harrison, R. B. Banerji, and J. D. Ullman, eds.), pp. 151–158, ACM, 1971.

[26] M. Vasquez and J. Hao, "A "logic-constrained" knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite," *Computational Optimization and Applications*, vol. 20, no. 2, pp. 137–157, 2001.

[27] H. Xu, R. Rutenbar, and K. Sakallah, "sub-SAT: a formulation for relaxed boolean satisfiability with applications in routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 814–820, 2003.

[28] S. Safarpour, H. Mangassarian, A. Veneris, M. Liffiton, and K. Sakallah, "Improved design debugging using maximum satisfiability," in *Formal Methods in Computer Aided Design*, pp. 13–19, IEEE, 2007.

[29] J. R. Rice, "The algorithm selection problem," *Adv. Comput.*, vol. 15, pp. 65–118, 1976.

[30] F. Hutter, H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, pp. 507–523, 2011.

[31] D. R. Jones, M. Schonlau, and W. J. Welch, “Efficient global optimization of expensive black-box functions,” *J. Glob. Optim.*, vol. 13, no. 4, pp. 455–492, 1998.

[32] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, “F-race and iterated F-race: An overview,” in *Empirical Methods for the Analysis of Optimization Algorithms*, pp. 311–336, 2010.

[33] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, “ISAC–Instance-Specific Algorithm Configuration,” in *ECAI* (H. Coelho, R. Studer, and M. Wooldridge, eds.), vol. 215 of *FAIA*, pp. 751–756, 2010.

[34] G. Hamerly and C. Elkan, “Learning the k in k-means,” in *In Neural Information Processing Systems*, p. 2003, MIT Press, 2003.

[35] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, “Algorithm portfolios based on cost-sensitive hierarchical clustering,” *IJCAI*, pp. 608–614, 2013.

[36] C. Gomes and B. Selman, “Algorithm portfolios,” *Artificial Intelligence*, vol. 126, pp. 43–62, 2001.

[37] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, “A portfolio approach to algorithm selection,” in *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003* (G. Gottlob and T. Walsh, eds.), p. 1542, Morgan Kaufmann, 2003.

[38] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, “SATzilla: portfolio-based algorithm selection for sat,” *JAIR*, vol. 32, no. 1, pp. 565–606, 2008.

[39] E. O’Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O’Sullivan, “Using case-based reasoning in an algorithm portfolio for constraint solving,” *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[40] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, “Algorithm selection and scheduling,” *CP*, pp. 454–469, 2011.

[41] L. Xu, F. Hutter, J. Shen, H. Hoos, and K. Leyton-Brown, “SATzilla2012: Improved algorithm selection based on cost-sensitive classification models,” 2012. SAT Competition.

[42] B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Fréchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, and J. Vanschoren, “ASlib: A benchmark library for algorithm selection,” *Artificial Intelligence*, vol. 237, pp. 41–58, 2016.

[43] B. Adenso-Diaz and M. Laguna, “Fine-tuning of algorithms using fractional experimental design and local search,” *Operations Research*, vol. 54, no. 1, pp. 99–114, 2006.

[44] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Intell. Res.*, vol. 36, pp. 267–306, 2009.

[45] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, and K. Tierney, "Model-based genetic algorithms for algorithm configuration," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015* (Q. Yang and M. J. Wooldridge, eds.), pp. 733–739, AAAI Press, 2015.

[46] L. Xu, H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," *AAAI*, pp. 210–216, 2010.

[47] C. Ansótegui, J. Gabàs, Y. Malitsky, and M. Sellmann, "MaxSAT by improved instance-specific algorithm configuration," *Artificial Intelligence*, vol. 235, pp. 26 – 39, 2016.

[48] J. Boyan and A. Moore, "Learning evaluation functions to improve optimization by local search," *Journal of Machine Learning Research*, vol. 1, no. Nov, pp. 77–112, 2000.

[49] B. Doerr and C. Doerr, "Optimal parameter choices through self-adjustment: Applying the 1/5-th rule in discrete settings," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 1335–1342, ACM, 2015.

[50] T. Stützle and M. López-Ibáñez, "Automatic (offline) configuration of algorithms," in *GECCO Companion*, pp. 795–818, 2016.

[51] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg, "Hyper-heuristics: An emerging direction in modern search technology," *Handbook of metaheuristics*, pp. 457–474, 2003.

[52] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[53] E. Özcan, M. Mısır, G. Ochoa, and E. K. Burke, "A reinforcement learning: Great-deluge hyper-heuristic," *Modeling, Analysis, and Applications in Metaheuristic Computing: Advancements and Trends: Advancements and Trends*, vol. 34, 2012.

[54] M. Mısır, K. Verbeeck, P. De Causmaecker, and G. V. Berghe, "An intelligent hyper-heuristic framework for chesc 2011," in *Learning and Intelligent Optimization*, pp. 461–466, Springer, 2012.

[55] B. Doerr and C. Doerr, "Optimal static and self-adjusting parameter choices for the $(1 + (\lambda, \lambda))(1 + (\lambda, \lambda))$genetic algorithm," *Algorithmica*, Aug 2017.

[56] A. KhudaBukhsh, L. Xu, H. Hoos, and K. Leyton-Brown, "SATenstein: Automatically building local search sat solvers from components," *IJCAI*, pp. 517–524, 2009.

[57] S. Kadioglu and M. Sellmann, "Dialectic search," in *CP*, pp. 486–500, 2009.

[58] H. Lourenço, O. Martin, and T. Stützle, "Iterated local search," in *Handbook of metaheuristics*, pp. 320–353, Springer, 2003.

[59] F. Glover, M. Laguna, and R. Marti, "Fundamentals of scatter search and path relinking," *Control and Cybernetics*, vol. 39, pp. 653–684, 2000.

[60] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. Hoos, and K. Leyton-Brown, "The configurable SAT solver challenge (CSSC)," *arXiv preprint arXiv:1505.01221*, 2015.

[61] C. Gomes, H. Kautz, A. Sabharwal, and B. Selman, *Satisability Solvers*. Elsevier B.V., 2008.

[62] C. Ansótegui, F. Didier, and J. Gabàs, "Exploiting the structure of unsatisfiable cores in maxsat," in *IJCAI*, pp. 283–289, 2015.

[63] R. Battiti, M. Brunato, and F. Mascia, *Reactive search and intelligent optimization*, vol. 45. Springer Science & Business Media, 2008.

[64] F. Glover and M. Laguna, "Tabu search," in *Handbook of Combinatorial Optimization*, pp. 3261–3362, Springer, 2013.

[65] R. Martí, M. Laguna, and F. W. Glover, "Principles of tabu search," in *Handbook of Approximation Algorithms and Metaheuristics* (T. F. Gonzalez, ed.), Chapman and Hall/CRC, 2007.

[66] D. Leventhal and M. Sellmann, "The accuracy of search heuristics: an empirical study on knapsack problems," *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 142–157, 2008.

[67] T. Sugawara, "Maxroster: Solver description," *MaxSAT Evaluation 2017*, p. 12, 2017.

[68] C. Ansótegui, F. Bacchus, M. Järvisalo, and R. Martins, "MaxSAT Evaluation 2017," 2017. `http://mse17.cs.helsinki.fi`.

[69] C. Ansótegui, J. Gabàs, Y. Malitsky, and M. Sellmann, "Maxsat by improved instance-specific algorithm configuration," *Artif. Intell.*, vol. 235, pp. 26–39, 2016.

[70] SAT-Competition, 2019. www.satcompetition.org.

[71] MaxSAT-Evaluations, "MaxSAT Evaluations," 2021. `https://maxsat-evaluations.github.io/`.

[72] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. Hoos, and K. Leyton-Brown, “The configurable sat solver challenge (cssc),” *Artificial Intelligence*, vol. 243, pp. 1 – 25, 2017.

[73] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Satzilla2009: an automatic algorithm portfolio for sat. solver description,” 2009. SAT Competition.

[74] L. Xu, F. Hutter, J. Shen, H. H. Hoos, and K. Leyton-Brown, “Satzilla2012: Improved algorithm selection based on cost-sensitive classification models,” 2012. SAT Competition.

[75] L. Xu, H. H. Hoos, and K. Leyton-Brown, “Hydra: Automatically configuring algorithms for portfolio-based selection,” *AAAI*, 2010.

[76] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, “Apache spark: A unified engine for big data processing,” *Commun. ACM*, vol. 59, pp. 56–65, oct 2016.

[77] M. Lindauer and F. Hutter, “Warmstarting of model-based algorithm configuration,” in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018* (S. A. McIlraith and K. Q. Weinberger, eds.), pp. 1355–1362, AAAI Press, 2018.

[78] F. Hutter, M. López-Ibáñez, C. Fawcett, M. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Stützle, “Aclib: A benchmark library for algorithm configuration,” in *Learning and Intelligent Optimization* (P. M. Pardalos, M. G. Resende, C. Vogiatzis, and J. L. Walteros, eds.), (Cham), pp. 36–40, Springer International Publishing, 2014.

[79] A. Şen, A. Atamtürk, and P. Kaminsky, “A conic integer programming approach to constrained assortment optimization under the mixed multinomial logit model,” Research Report BCOL.15.06, IEOR, University of California–Berkeley, October 2015.

[80] D. J. Papageorgiou, G. L. Nemhauser, J. Sokol, M.-S. Cheon, and A. B. Keha, “Mirplib – a library of maritime inventory routing problem instances: Survey, core model, and benchmark results,” *EJOR*, vol. 235, no. 2, pp. 350 – 366, 2014. Maritime Logistics.

[81] D. Sheldon, B. Dilkina, A. Elmachtoub, R. Finseth, A. Sabharwal, J. Conrad, C. P. Gomes, D. Shmoys, W. Allen, O. Amundsen, and B. Vaughan, “Maximizing spread of cascades using network design,” in *UAI-2010: 26th Conference on Uncertainty in Artificial Intelligence*, (Catalina Island, Avalon, CA), pp. 517–526, July 2010.

[82] K. Ahmadizadeh, B. Dilkina, C. P. Gomes, and A. Sabharwal, "An empirical study of optimization for maximizing diffusion in networks," in *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, pp. 514–521, 2010.

[83] F. Hutter and Y. Hamadi, "Parameter adjustment based on performance prediction: Towards an instance-aware problem solver," Tech. Rep. MSR-TR-2005-125, Microsoft Research, Cambridge, UK, December 2005.

[84] M. Lindauer, M. Feurer, K. Eggensperger, A. Klein, S. Falkner, and F. Hutter, "Parallel SMAC (pSMAC)," 2018.

[85] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 2962–2970, Curran Associates, Inc., 2015.

[86] H. H. Hoos, M. Lindauer, and T. Schaub, "claspfolio 2: Advances in algorithm selection for answer set programming," *Theory and Practice of Logic Programming*, vol. 14, no. 4-5, p. 569–585, 2014.

[87] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[88] T. Fitzgerald, Y. Malitsky, B. O'Sullivan, and K. Tierney, "React: Real-time algorithm configuration through tournaments," in *Proceedings of the Symposium on Combinatorial Search*, 2014.

[89] T. Fitzgerald, Y. Malitsky, and B. O'Sullivan, "Reactr: Realtime algorithm configuration through tournament rankings," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, Citeseer, 2015.

[90] A. El Mesaoudi-Paul, D. Weiß, V. Bengs, E. Hüllermeier, and K. Tierney, "Pool-based realtime algorithm configuration: A preselection bandit approach," in *International Conference on Learning and Intelligent Optimization*, pp. 216–232, Springer, 2020.

[91] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Parallel algorithm configuration," in *Proc. of LION-6*, pp. 55–70, 2012.

[92] P. Prettenhofer, "Parallel grid search for sklearn Gradient Boosting." `https://gist.github.com/pprett/3989337`. Accessed May, 2015.

[93] E. Cantu-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, vol. 10, 1998.

[94] E. O. Scott and K. A. D. Jong, "Evaluation-time bias in quasi-generational and steady-state asynchronous evolutionary algorithms," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016* (T. Friedrich, F. Neumann, and A. M. Sutton, eds.), pp. 845–852, ACM, 2016.

[95] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual.* Scotts Valley, CA: CreateSpace, 2009.

[96] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[97] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference* (Stéfan van der Walt and Jarrod Millman, eds.), pp. 56 – 61, 2010.

[98] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[99] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[100] F. Chollet *et al.*, "Keras," 2015.

[101] Gurobi Optimization, "Gurobi." `https://www.gurobi.com/`, 2021.

[102] Google, "Google OR-Tools." `https://developers.google.com/optimization`, 2021.

[103] COIN-OR Foundation, "Computational infrastructure for operations research." `https://www.coin-or.org/`, 2016.

[104] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig, "The SCIP Optimization Suite 7.0," ZIB-Report 20-10, Zuse Institute Berlin, March 2020.

[105] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[106] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, pp. 428–437, 2018.

[107] G. Audemard, L. Paulevé, and L. Simon, "SAT heritage: A community-driven effort for archiving, building and running more than thousand SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings* (L. Pulina and M. Seidl, eds.), vol. 12178 of *Lecture Notes in Computer Science*, pp. 107–113, Springer, 2020.

[108] M. Lauria, J. Elffers, J. Nordström, and M. Vinyals, "Cnfgen: A generator of crafted benchmarks," in *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings* (S. Gaspers and T. Walsh, eds.), vol. 10491 of *Lecture Notes in Computer Science*, pp. 464–473, Springer, 2017.

[109] Logic Optimization Group, "PyPBLib: PBLib Python3 bindings." `https://pypi.org/project/pypblib/`, 2019.

[110] T. Philipp and P. Steinke, "PBLib – a library for encoding pseudo-boolean constraints into CNF," in *Theory and Applications of Satisfiability Testing – SAT 2015* (M. Heule and S. Weaver, eds.), (Cham), pp. 9–16, Springer International Publishing, 2015.

[111] T. Balyo and contributors, "The standard interface for incremental satisfiability solving." `https://github.com/biotomas/ipasir`, 2014.

[112] Logic and Optimization Group, "Optilog official documentation," 2021.

[113] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions* (T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, eds.), vol. B-2020-1 of *Department of Computer Science Report Series B*, pp. 51–53, University of Helsinki, 2020.

[114] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, (San Francisco, CA, USA), p. 399–404, Morgan Kaufmann Publishers Inc., 2009.

[115] A. Biere, "Picosat essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, no. 2-4, pp. 75–97, 2008.

[116] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), (Berlin, Heidelberg), pp. 502–518, Springer Berlin Heidelberg, 2004.

[117] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," *Proceedings of SAT competition*, vol. 2013, p. 1, 2013.

[118] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, Jan. 2006. Publisher: IOS Press.

[119] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, Jan. 2010. Publisher: IOS Press.

[120] F. Bacchus, J. Berg, M. Järvisalo, and R. Martins, "MaxSAT Evaluation 2020: Solver and benchmark descriptions," 2020.

[121] M. Lindauer, J. N. van Rijn, and L. Kotthoff, "Open algorithm selection challenge 2017: Setup and scenarios," in *Proceedings of the Open Algorithm Selection Challenge* (M. Lindauer, J. N. van Rijn, and L. Kotthoff, eds.), vol. 79 of *Proceedings of Machine Learning Research*, (Brussels, Belgium), pp. 1–7, PMLR, 11–12 Sep 2017.