Chapter 2
Automatic Design for Matheuristics

2.1 Introduction

To organize a matheuristic, we can use a large variety of techniques. These organize the search intensification and diversification and they guide in their way the problem-specific components. Traditionally, matheuristics design and development is involved in a manual, experimental approach that is mostly guided by the experience of the algorithm designers. This way the process is guided by overgeneralizations from the designers previous experience and by implicit independence assumptions between parameters and algorithm components. In addition to these, the manual process has a number of other disadvantages. Some are that it limits the number of designs that would be involved, it hides the efforts that have been dedicated to their developement, it loses the number of design alternatives that are involved, or it simply makes the design process irreproducible.

The implementation effort that is associated with the design of matheuristic algorithms is very variable. Basic versions that consist of few numerical parameters can be, in principle, designed conventially with little effort and reach good performance. However, when very high performance is required, the design of matheuristics profits strongly from the exploitation of problem-specific knowledge, the time invested in designing and tuning the algorithms and the use of insights into algorithm behavior and the interplay with problem characteristics. With this they go along, at least in tendency, with the importance of the parameters. The matheuristics can be seen as a rather general technique, varying their parameters as categorial or ordinal parameters (more on it in Section 2.2.1) together with specialized algorithm components usually in the form of numerical parameters.

There have been a number of methods that have adopted numerical and also categorical and ordinal techniques. A first one is the determination of numerical parameters. For a few parameters it is the fractional design or variants of it, but it has the disadvantages that it is related to the chosen parameter settings. A different approach is to configure numerical parameters with numerical optimization algorithms. One can use fractional design techniques also if one has categorical or ordinal parameters, yet they turn infeasible if the parameters are many. Then, instead, there are several methods one can choose from, be it heuristics search techniques, statistical methods or model-based techniques. Here we can cite ParamILS and gender-based genetic algorithm as heuristics search methods, F-race and iterated F-race as statistical racing techniques, and sequential model-based configuration. These techniques deal with the stochasticity of the algorithms and have successfully dealt with tens or sometimes hundreds of parameters.

A first approach to these automatic algorithm configuration techniques consists in the use of numerical techniques for the configuration of the parameter setting of matheuristics algorithms. This includes algorithms that are fully specified according to their numerical distributions and alternative algorithm components, and a numerical parameter may be substituted if its value is zero. Yet, although the numerical results may already improve significantly the performance of the algorithm, the effective importance of the automatic algorithm configuration techniques is in their essential role of making the categorical and ordinal choices of a matheuristic. It is an alternative design method that relies on a design paradigm: defining an appropriate design space which encodes alternative algorithm design choices and numerical parameter. Then a computation intensive process explores this design space to find high-performing algorithm instantiations. In a variety of research efforts, this method has been shown to be feasible as we may see in a few examples in this chapter.

The chapter is organized as follows. First, in Section 2.2, we highlight the settings of parameters that leads to the parameter configuration problem and leads to an increasing automatization of design. In Section 2.3, we give a number of successful examples that highlight the potential of an automated design of methods. We then go through an example with the irace software package in Section 2.4, showing with a short example with iterated local search for the Generalized Assignment Problem (GAP), how it works.

2.2 Parameter configuration

In this section, we give an overview of the type of parameter settings that we have in algorithm deployments and define what the configuration of parameters is. Then we go through the developments of automatic parameter configurators and show three examples: ParamILS, SMAC and irace.

2.2.1 Parameters

The available choices for the parameter configuration problem depend on the type of the parameters. On one side we have discrete options available. These are *categorical* parameters if no details are given to allow distinguishing the value of the choices. These can be the choice of a simulated annealing or a tabu search as local search procedure or the choice of different perturbation types in an iterated local search. Sometimes it may be possible to give to the values an ordering without having a clear distance measure available. This gives place to *ordinal* parameters, with an example being $\{$small, medium, large$\}$. In the GAP, it could be the size of a neighbourhood or the value of a lower bound. Finally, there are parameters which are numerical in nature. These can be real-valued parameters, such as the temperature in simulated annealing or the evaporation rate in ant colony optimization, or integer ones, such as the population size in memetic algorithms or the tabu list length in tabu search. These numerical parameters one can have as algorithm-wide parameters if they depend on the search method. For example, this is the case with the population size for an memetic algorithm. However, a parameter is *conditonal,* if it depends on others that are to be set usually as a categorical or an ordinal one. For example, whether having a temperature parameter depends on whether one has a simulated annealing as local search or a tabu search.

Another issue is the size of the parameter settings. For many cases like the categorical and the ordinal parameters these will be not addressed, as it is commonly the same for each of them. However, for the numerical parameters need a range whose size and precision impacts the search. In case of doubt, one may opt for ranges than are maybe too wide. They require more experiments than smaller ranges, but they can give better results. Only if the experiments are too few, it could be problematic and one would be too demanding with the size or the precision.

2.2.2 Parameter configuration problem

We can now describe the parameters of an algorithm in more detail. The performance-optimizing algorithm configuration can be described by the set of variables of any type, that is, they can be numerical, ordinal, and categorical. Every parameter $\theta_i, 1, \dots, N_p$, is given an associated type t_i and a domain D_i, so that we have a parameter vector $\theta = (\theta_1, \dots, \theta_{N_p}) \in \Theta$, where Θ is the space of possible parameters settings.

The goal in the design of algorithms then is to optimize for some problem Π the performance of an algorithm for a specific instance distribution $\mathscr{I}$ of problem Π. In formal ways, when applied to a problem instance π_i, the performance of the algorithm is the cost measure $\mathscr{C}(\theta, \pi_i) : \Theta \times \mathscr{I} \rightarrow R$. Usually, this will be a random variable with a typically unknown distribution, if during the search the algorithm involves stochastic decisions. Yet by executing an algorithm on a specific instance, one can measure realizations of the random variable. Another element of stochasticity is the fact that each instance π_i can be seen as being obtained from some random instance distribution $\mathscr{I}$. The performance $F_{\mathscr{I}}(\theta) : \Theta \rightarrow R$ of a configuration is then defined as function for an instance distribution $\mathscr{I}$.

The usual approach is to define $F_{\mathscr{I}}(\theta)$ with respect to the expected cost $E[\mathscr{C}(\theta, \pi_i)]$ of θ if applied to a instance distibution. The definition of $F_{\mathscr{I}}(\theta)$ determines how to compare configurations over a set of instances. If cost values across different instances are not comparable on a same scale, rank-based measures such as the median or the sum of ranks may be more meaningful. The precise value of $F_{\mathscr{I}}(\theta)$ can generally not be computed in an analytic way but it can be estimated by sampling. This means that one obtains realizations $c(\theta, \pi_i)$ of the random variable $\mathscr{C}(\theta, \pi_i)$ by running an algorithm configuration θ on instances that have been sampled according to $\mathscr{I}$.

We can then say that the algorithm configuration problem is to identify an algorithm configuration θ^* where it holds

$$\theta^* = \arg\min_{\theta \in \Theta} F_{\mathscr{I}}(\theta). \tag{2.1}$$

In summary, the algorithm configuration problem is a stochastic optimization problem where the types of variables are numerical, categorical, ordinal or of any other type. Each of these variables has a specific domain of distribution and a specific type of constraints. The stochasticity of the configuration task is determined by the types of the stochasticity of the algorithm and the stochasticity that stems from the sampling of the problem instances.

Due to the stochastic nature of the configuration process, it is of importance to let this process to generalize to unseen instances. As a result, this process is done in two phases. In a first *training* phase, a high-performing configuration is searched ideally by a large training size. Clearly, it is important that this training set it also in shape and in size similar to the test phase. This is also ensured by a specific application context that has, for example, the available computation times in which the algorithms is employed. In a second *test* phase, this configuration is then evualated as the true configuration on an independent test set. This generalization to the test instances is the real setting. This also reflects the nature of the setting. In a first phase, the training phase, an algorithm is designed and trained for a specific target application and in a second phase, the test phase, the algorithm is deployed to solve new, previously unseen instances.

2.2.3 Automatic algorithm configuration

While in the old matheuristics literature the design of parameters is pre-set and mainly presented as a manual trial-and-error process, in the current literature the automatic algorithm processing is given a distinctive place and gives rise to a computationally intensive processes. A configurator gets as input the name of the variables, the type of the variables, and the values of the parameters. For the categorical and ordinal parameters it gets the discrete sets of values, such as $\{x, y, z\}$ and $\{small, medium, large\}$, and for the numerical parameters the real-valued variables, for example $[0, 1]$, or the integers, for example $\{0, 1, 2, \ldots, 10\}$. Additionally, the configuration tools may receive information about which parameters depend on each

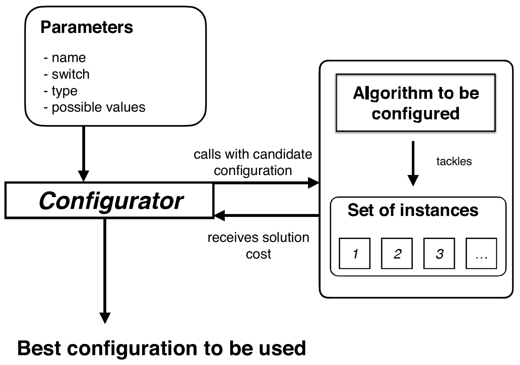

Fig. 2.1 Generic view on the main interaction of an automatic configuration technique with the configuration scenario.

other, that is, the conditional parameters, and which parameters have forbidden values, and any other detailed information that may be helpful.

The configurator sets these parameter values to one or several configurations, depending on whether it is a population-based algorithm or a single search-based configurator. These configurations are evaluated on training instances and the evaluations of the candidate configurations are returned to the configurator. This process of returning the configurations and solving them by the software is repeated until some measure of interest is achieved. The most common measures are the amount of computational experiments or a measure of the computation time. The first is commonly a measure that is used for metaheuristics and consists in the number of experiments executed. The second, the computation time, be it CPU or wall-clock time, is usually the time that is allowed for a specific computation. Independent of the specific termination, the best configuration is returned at the end of the process. Usually, also some other information on the configurations process is returned such as other high-quality configurations or statistics of the configuration.

A general overview of these interaction between the configurator, the parameter definition, the calls of the software and the return of the solution costs is given in Figure 2.1.

We can now come to highlight some of the main approaches. We can classify them as experimental design techniques, continuous optimization techniques, heuristic search techniques, surrogate-model based configurators, and non-iterated and iterated racing approaches.

Experimental design techniques. To avoid the immediate pitfalls of trial-and-error processes, various researchers have adopted statistical techniques, such as hypothesis testing for evaluating the statistical significance of performance differences, or experimental design techniques such as factorial or fractional factorial designs and response surface methodologies. While often experimental design techniques such as ANOVA have been applied using manual intervention, several efforts have been made to exploit such techniques to make them more automated. An example in this direction is the CALIBRA approach, which applies Taguchi designs and a refinement local search to tune five parameters. A different approach is based on the exploitation of response surface methodologies. Yet, a common disadvantage of these methods is that many of these are limited to few variables and that many of these have the variables fixed.

Continuous optimization techniques. A different approach is numerical optimization in the form of continuous optimization. It first has to be mentioned that in this case it is often feasible to integrate integer and continuous function optimization by a continuous relaxation approach in form of truncation and rounding. Independent of this, such things are often feasible if the algorithm either has only numerical parameters or if the algorithm has already been fine-tuned by setting all the right value of categorical or ordinal parameters. One of the first is the mesh-adaptive direct search (MADS) mechanism, which features a mathematical program for the search and a large set of functions for the heuristic evaluations. About the same time, Nannen and Eiben have done some work on the REVAC algorithm. In a computational study of various instances among which were BOBYQA, CMAES, Iterated F-race, and MADS, they found that for small instances with two to four parameters the best performance was obtained by BOBYQA, while beyond that parameters the best performances were obtained by a modified version of CMAES. This performance was further improved by post-selection in automated configuration, where first a subsets of configurations is defined and afterwards it undergoes a more careful evaluations.

Heuristic search techniques. If instead one has to make also categorical or ordinal parameters choices, one has a number of heuristic choices available. One of the first choices was the meta-genetic algorithms that was proposed by Grefenstette in the year 1986; but there the performance was too weak to be impressive. It was in the first and in the second decades of this century that these methods became promising. One of the first ones was the ParamILS algorithms; we will go in more detail in Section 2.2.4. More recent is the gender-based genetic algorithm that is rather good for exact algorithms and which has also been adapted in recent years to make use of surrogate models. Other approaches are linear genetic programming algorithms that evolve evolutionary algorithms, the OPAL system that extends MADS to include categorial and binary parameters, the EVOCA evolutionary algorithm, and the heuristic oriented racing algorithm (HORA).

Surrogate model-based configurators. The evaluation of configurations is typically the most computationally demanding part of an automatic configuration method, since it requires actually executing the target algorithm being tuned. Several methods aim to reduce this computational effort by using surrogate models to predict the cost value of applying a specific configuration to one or several instances. Based on the predictions, one or a subset of the most promising configurations are then actually executed and the prediction model is updated according to these evaluations. Among the first surrogate based configuration methods is sequential parameter optimization (SPOT). A more general method also using surrogate models is the sequential model-based algorithm configuration (SMAC). In fact, this is currently one of best configurators and we will revise it in some more details in Section 2.2.5.

(Iterated) racing approaches. Finally, other systems investigate racing methods for selecting one or various configurations among a (huge) sets of candidate configurations. They use F-race or other sequential statistical testing methods to do the racing. A race can be organized based on design of experiments techniques, randomly generated configurations or based on problem-specific information. If the race is iterated, this iterations can be based on problem-instance based methods. For example, iteration i is based on the best configuration of the previous race $i-1$. In Section 2.2.6 we will see an example of a configurator based on the iterated racing approach.

Independent of the automated algorithm configuration, the usage of such a system has some advantages. A first is that one has the parameters for the system. Here it is important that one needs not only the "usual" parameters, but one can extend those parameters beyond those that are in use. We further may distinguish between categorical and ordinal parameters on the one side, and between numerical parameters on the other. The categorical and ordinal ones are the normal parameters, which discriminate major parts of the algorithm that before where in many designs pre-given. The numerical parameters are those that are available such as the temperature parameter in simulated annealing. However, when one has a automatic configuration system available, one can also deal with numerical parameter that otherwise would be preset. One can also distinguish between conditional parameters, which only may play an essential role for some parameters of a system.

Apart from the parameters, one has the further advantage that there is a clear separation between the training instances and test instances. Ideally, there is only the distinction between training and test instances in the sense that there are differences only in, say, random seeds. A further advantage is the clear role of the configurator and the algorithms that are configured. This comprises the budget, special features of the configurator and so on. Last but not least, a final advantage is that with improvements of the configurators we may also improve performance of the configured algorithm.

2.2.4 ParamILS

We are now able to present in more details some popular configurators. We start with ParamILS, which is an ILS algorithm in the search space of the parameters. A first difference from the other parameter configurators is that it is a fully categorical parameter system. In other words, real and integer parameters have to be converted into discrete values. This can be done by either introducing a grid or, if there is some knowledge available, by introducing a specific parameter space, or simply at random, which is often better than grid-based.

As initialization of ParamILS, it is required a default configuration. In addition to this, r random inital configuration are determined and the best of this $r+1$ configurations builds the initial solutions. This is done as random configurations may be better than the default configuration.

The local search in ParamILS is a first-improvement local search with a random-improvement basis. Each time this random-improvement scheme finds a new improvement, it takes it and continues with the random-improvements scheme which is randomly initialized. This is continued until all the neighborhood is examined or the budget is exhausted.

When a local search is terminated, the better of the two local minima is kept and a perturbation is done. With a probabilty of $1 - p_R$ this is done by actually taking the better of the two and introducing a k randomly chosen parameter to obtain a new θ' as the solution. By default, $k = 3$ random parameters are set. Yet with a probability of p_R, the perturbation consists in a random contribution that is generated from which the next ILS iteration is started. By default, p_R is set to 0.01.

Of course, deciding which configuration is better is not trivial. ParamILS offers two variants: BasicILS and FocusedILS. BasicILS chooses the best of two configurations after having them evaluated them on n_e instances, where n_e is a given set on instances. Such setting can be approximately right or rather be very wrong. In other words, it is simply not a trivial task to set n_e: too low it will risk not to provide enough estimates; too high, it will risk to waste enough candidates to be seen. The FocusedILS version compares two configurations increasingly, until one configuration dominates the other. Dominance between two configurations is established as follows. Let the two configuration θ_1 and θ_2 be evaluated on $n_1 > n_2$ instances. Then configuration θ_1 dominates θ_2 if we have that $\hat{F}(\theta_1, n_2) \leq \hat{F}(\theta_2, n_2)$, where $\hat{F}(\theta, n)$ gives the cost estimate of a configuration θ on n instances. Hence, a configuration dominates another one if it has been used on as many instances as the other and it has a lower cost estimate. If θ_1 is not dominating the other one, the number of instances on which the second one will be evaluated is redone. When a new instance improves over the old one, the number of instances will also be increased.

Often, ParamILS is used to configure for run-time. For example, this is very often the case when configuring exact algorithms such as CPLEX, Gurobi or FICO Xpress for specific tasks. Therefore, ParamILS implements a procedure that is called adaptive capping which prunes search early for the evaluation of potentially poor performing configurations. When it is combined with the above dominance criterion, adaptive capping can strongly reduce the computing time.

ParamILS is publically available at http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/, the version at the time of writing is 2.3.8. It has been shown to excel in a number of configuration tasks and, in particular, it led to speed-ups of several orders of magnitude for mixed-integer programming or propositional satisfiability problem.

2.2.5 SMAC

Sequential model-based algorithm configuration (SMAC) is another configurator proposed by some of the same authors of ParamILS. It is a configurator, which uses a surrogate-model search of the parameter space and, as opposed to ParamILS, handles both real and integer parameters and categorical parameters. This means that it does not need only discrete numerical parameters as ParamILS.

SMAC starts from an initial configuration, which is typically the algorithm default. When no algorithm default is available, a random one can be given. With this input, the procedure loops through the followin steps. First, a random forest model is learned. Following this, a set of candidate configurations is determined by following n_{LS} elite configurations. Each of them is the candidate of the n_{LS} elite configurations, which are determined as the best-improvement local searches on the configuration space; each time an algorithm is selected according to the expected improvement criterion. In addition, n_r randomly generated configurations are generated, where each one is evaluated according to the expected improvement criterion. Then, SMAC sorts the $n_{LS} + n_R$ configurations according the their expected improvement and executes them in thi order. The process of evaluating the instances is the same as for FocusedILS. It uses the dominance criterion, the adaptive capping, and increases the number of instances on which the incumbent is evaluated. This evaluation process is terminated after a specific termination time. Then the next overall iteration of SMAC is invoked which starts first with the learning of the new random forest.

While the high-level search process that is implemented by SMAC follows these stages, there are a number of rather intricate steps which make the process much more involved. Some of these are special treatments of censored data, which can improve the predictions of SMAC. Another one is the inclusion of instance features for improving the predictions. All in all they make SMAC currently one of the top-performing configurators.

SMAC is available online. The SMAC v2.08 in Java and the version SMAC v3 is written in Python3 and is continuously tested in Python3.5 and Python3.6. The versions of SMAC v2.08 and SMAC v3 are virtually the same, but the version SMAC v2.08 gives slightly better performance. The SMAC v3 is free software, which one can redistribute or modify it under the terms of the 3-clause BSD license.

2.2.6 irace

Last but not least is the irace system that implements an iterated racing for the determination of the best configuration. It is based on three subsequent iterations. First, it generates candidate configurations from a statistical model. Second, it selects the best algorithm candidate configurations by a statistical racing method that is either F-race or paired Student t-test. Third, it updates the probabilistical model of the system to give the most promising candidates a higher probability of being used.

In the first step, irace has to sample new configurations. In the first iterations, when no results are available, it has various possibilities. If nothing is known the first generation may be generated by random sampling, that is, all the first generation is random. Of course, one or also various candidates may be given as default configurations. From the second iteration, each of the surviving configurations, called elites, forms a (partial) model. Each of these elites will have numerical, ordinal and categorical parameters. If it is numerical and ordinal, then the probabilistic parameters are drawn from a truncated normal distribution (μ_i^j, σ_i^j) where μ_i^j is the mean value of the i-th parameter of the j-th elite configuration and σ_i^j is the standard deviation. The truncation is happening because the parameter is in a range $[x_l, x_u]$, where x_l and x_u are the lower and the upper boundries of the parameter. If this is an integer or an ordinal parameter, then it is rounded as $\text{round}(x-0.5)$. If instead with X_d we have a categorical parameter with levels $X_d \in \{x_1, x_2, \ldots, x_{n_d}\}$, then a new value is sampled from $\mathscr{P}^{j,z}$. This distribution is set to the uniform distribution at the beginning and then at each iteration is set to the sampling as follows:

$$\mathscr{P}^{j,z}(X_d = x_j) = \mathscr{P}^{j-1,z}(X_d = x_j) \cdot \left(1 - \frac{j-1}{N^{iter}}\right) + \Delta\mathscr{P} \tag{2.2}$$

where

$$\Delta\mathscr{P} = \begin{cases} \dfrac{j-1}{N^{iter}} & \text{if } x_j = x_z \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where N^{iter} is the estimated number of iteration, j is the current iteration, and x_z is the current elite. This way, the entries of the elite probability for a categorical parameter is always the one with highest probability.

After the configurations are generated, they undergo a race. In a race, all the configurations are evaluated on a first instance, then on a second one and so on. As soon as the configurations are deemed to be poor, they are eliminated. There are different criteria how this is managed, which in part depends on the objective function, which can be quality-wise or time-wise. If it is quality-wise then the usual thing is to take a test. This is done in irace for this case. In fact, one has either the iterated F-race, that is based on the non-parametric Friedman's two-way analysis of variance by ranks or the paired Student's t-test with or without p-value corrections. The first one, the F-race, can be used especially if the non-parameteric test is wortwhile, such as when the quality-wise setting is very different. For the second choice, we would recommend to do the test without the p-value corrections, since it eliminates much more quickly the poor candidates. If the test is time-wise, then in the version irace 3.0 one has an extension that better handles run-time minimization. In particular, it handles a dominance criterion among configurations. A configuration θ_1 dominates θ_2 if it has more instances N_1 than N_2 and it holds that cost $c(\theta_1, 1, \ldots, N_2) \leq c(\theta_2, 1, \ldots, N_2)$. This maintains the trajectory and it works very well together with the Student's t-test.

After a race is terminated, either because the minimum elite number of configurations is reached or the computational budget is reached, one makes the update step of the elite configurations. This is done by numerical and ordinal parameters centering these at the point of their mean and diminishing their standard deviations as it is foreseen in the irace software. For categorical parameters the elite probability and the other probabilities are set as indicated above.

The irace software is publicly available at http://iridia.ulb.ac.be/irace/ or at https://cran.r-project.org/web/packages/irace. Currently, it is available as version irace 3.4.

2.3 Design of algorithms

In what follows, we claim that advantages of automatic configuration go from an automatic configuration of existing algorithm, to integration of an algorithm engineering process and advanced uses of the automated configuration software. It is especially this last one where the systematic use of automated configuration software has real advantages.

2.3.1 Automated configuration of existing algorithms

One of the first uses of automated configuration software is to determine parameter settings that would improve performance. This is feasible with relatively minor efforts. The reason is that the default setting can always be used if no advanced parameter setting is found. Yet, this is only rarely the case and often one can find parameter settings that substantially improve over it. There are various reasons for this. One is that the default values are for a very different instance distribution and, hence, the new parameters are very different from the old ones. Another reason is that running times of the new applications scenarios are very different from the one that had been assumed. Yet another reason may be that the default parameter settings are not well attached with the available instances.

There has been a number of papers about this, especially in the initial steps of the automated configuring methods. One of the first papers in this direction is the paper of Hutter et al. on the boosting of a verifications procedure in which they observed speedups of up to 500 time the default. Similar examples were obtained when employing them in the parametric case study on mixed-integer programming methods such as CPLEX. Other examples include the REVAC method, which reached improvements on the solution quality or the irace methods which found solutions quality improvements for various methods.

When we want to compare algorithms, we have to do the same thing: all of them should ideally be run on the same parameters setting for each algorithm. For this task the automatic configuration methods make good performance available for all algorithms. In particular, they make the parameter setting, if they are equally good of course, less an issue of the particular system designer than intrinsic to the system.

That is, the features of the system designer is not or, at least, less critical. All the algorithm parameters then have the same advantages as the single algorithms: the tuned versus the default parameters, the very different time or solution quality parameters, or the poor integration with the instances. Another advantage may be that an old algorithm may be surprisingly good when offering new parameters.

2.3.2 Integration into an algorithm (re-)engineering process

One way to use automatic configuration methods is to apply them only at the end of the development. But it may be much better to use automatic configuration methods already in each step of the engineering or re-engineering of the algorithms. It is in the steps where a major change is examined when it is most useful to see whether it actually improves the algorithm or not. A first approach in this direction was the design of an existing particle swarm optimization algorithm for a large-scale function optimisation competition. This approach went through a six steps engineering set which included a local search, call and control strategy of the local search, the particle swarm optimisation rules which were set to the original ones, the bound constraint handling when calling the particle swarm optimisation, the stagnation handling and finally the restarts. Here iterated F-race was used at each step to configure up to ten parameters of the algorithm. Although the configuration was done on only 19 function of dimension ten, it was found that the scaling was valid up to function dimension 1000. In a similar undertaking, Hartung and Hoos did an engineering of a parameterized algorithm used for clustered editing and Balaparash et al. developed some high-performing algorithm for the probabilistic traveling salesman problem and a single vehicle routing problem with stochastic demands and customers.

2.3.3 Advanced uses of configurators

A major research area is how to design new algorithms. This can be done at least in two ways: top-down approaches or bottom-up approaches. Both approaches use the same set of algorithm components and follow rules how to combine these components. Yet they differ in how they deal with the problem.

In the top-down approaches there is always one single basic algorithm template but different algorithm components are used. One of the first top-down approaches is the SATenstein approach. This approach could initialize five algorithmic blocks which go from a perform diversification, over walksat-based algorithms, dynamic local search algortithms, over G^2WSAT variants to updates of the data structures. While with SATenstein a new state-of-the-art solver for SAT local search algorithms could be generated, it is also clear that the local search is heavily optimised. The group of IRIDIA has explored several metaheuristics. These include several meta-

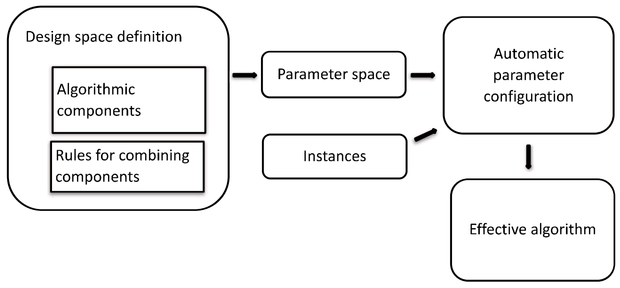

Fig. 2.2 General approach for deriving configurable algorithm frameworks.

heuristics which are single objective ones such as a framework for continuous optimization for ACO. In this case, they have shown that the numerical optimizer is outperforming the best previously known ACO. The same was obtained for the generalized artificial bee colony framework. Another research area is multi-objective optimization, which originated in 2010 with the help of multi-objective ant colony optimization and then went on with the TP+PLS framework and with an automatic design of multi-objective evolutionary algorithms. The most recent works explore an automated multi-objective evolutionary algorithm (AutoMOEA) that could instantiate a Pareto-based, an indicator-based or a weight-based MOEA. This was possible through a preference-relation which was set-partitioning, quality, and diversity based. In its latest setting, AutoMOEA could generate algorithms to solve problems with two, three, five and ten objectives that outperformed the best of the self-tuned algorithms.

A second method is the bottom-up approach, that tries to make it potentially more complex algorithms based on low-level rules frequently in the form of grammars. A first bottom-up approach is again for the SAT problem, based on a grammar-based approach. In fact, this is the approach that is commonly followed in the hyper-heuristic setting. In principle, it is a methodolgy that wants to generate constructive and perturbation heuristics and often these method are based on grammatical evolution and gene expression programming. A different approach was followed by the group of Stützle and followed by few others. They noticed that with standard software like irace they could replace grammatical evolution, a well established variant of grammar-based genetic programming. In fact, they noticed that this could be replaced by irace in a much more directed way. They first made a version that was based on Paradiseo. Later this system was designed from scratch and so far the results are very promising, producing state-of-the-art algorithms for a number of permutation flow-shop problems.

2.4 An example: irace

Let us continue this chapter with an example on irace configuring a simple iterated local search (ILS), which you can see also in section 3.4 of this book, for the generalized assignment problem (GAP). For this we assume that we have installed irace in R and we have a first ILS algorithm for the GAP. Now it is the time to improve this simple algorithm by configuring some of the parameters.

First, we have to notice that we do have infeasible and feasible solutions. Here we deal with it by generating all feasible solutions by counting infeasible ones with a penalty of 100 per integer infeasibility step. In other words, if we have say 27 infeasibilties, then we count them as $27 \cdot 100 = 2700$ in addition to the feasible part that is as usual in the GAP.

We have a few parameters of the ILS that we can change. The first part is the initial solution. Here we assume that the initial solution is a random one.

Next, we can address the acceptance criterion of our ILS. In the first place we have a better criterion, where we only accept improving solutions. As a slight change we may also accept the same solution quality as the best one. This may be better if we have many solutions in the neighbourhood that are of the same quality. A rather different acceptance criterion would be one that accepts every new solution as the current one. An intermediate to this may be a temperature-based one. One common choice is to use the simulating annealing one; see Section 3.2 for details of simulated annealing. It always accepts a solution if it is better or equal to the current one. It instead accepts with a probability $\exp(-\delta/T_{Tcurrent})$ a worsening solution, depending on the worsening δ and the temperature $T_{Tcurrent}$. Here the temperature is kept constant throughout the run of the algorithm. Hence, we have two parameters of the acceptance options. The first, is a categorical acceptance criterion with 4 criteria. The second is a conditional parameter for the temperature $T_{Tcurrent}$, set only if the simulated annealing acceptance criterion is chosen as the first acceptance parameter.

As the next item we have the perturbation. Here we implement two types of perturbation. In the first case, we do a fixed setting that is kept constant independent of problem size, that is, perturbation is equal to p, with $p \in (2,98)$. In the second case, we do a variable size perturbation in the style of variable neighborhood search (VNS); see Section 3.5 on it. We use a value 2, which is the smallest possible value larger than the local search step, and configure the upper limit p, bounded to be at most 98. Hence, we have also two parameters for the perturbation: the first is the type of perturbation, fixed or variable, and the second is the value p.

As the last one we can choose the way we use the local search. As first, we can choose no local search at all. This is fast, but we cannot assume it to be good enough. Hence, we use three more instances with local search. The first is a kind of first-improvement local search. It first examines the local search improvements that can be done for a specific server and if it is an improvement it applies it. Otherwise it goes to the next server. Once it has searched the server for each client, it stops the local search. In addition, it does a local search only if the initial solution is found to be feasible. However, notice that if a solution is found infeasible but better than the previous best one, it anyway replaces it. The two other local searches also perform a local search iteration when they are in a infeasible solution. The first is the very basic version of the second one and it does one time the local search for each server when the infeasibility counts as one if present. In the second one an infeasible solution gets an *alpha* penalty, where alpha is an integer. In the case of the second implementation a true local minima is identified, that is, a server is visited more than once except if the solution is already a local optima. We know that we could identify more local searches as, for example, we could apply a two-opt one neighborhood. With this, also the final solution would probably improve. Yet, we leave this for further work and we have two parameters as the local search: one is the type of local search, which is a categorical one with four parameters; the second is the value of *alpha* if the second local search is chosen.

With this, we can define the parameter file. This file has several inputs for each parameter, which are listed as

```
<name> <switch> <type> <range>  [ | <condition>].
```

Each entry then has the following meaning.

name	This is the name of the parameter as an alphanumeric string.
switch	A label for the parameter, which is a string that is used in the target runner of the system. In the default provided in the package it would be “-l “ or similar. Note that here the user has to take care of giving the right size. Especially, the separator is important here and has to be how the code expects it.
type	This is the type of the parameter. It can be either real, integer, ordinal, or categorical by r, i, o, and c. Numerical parameter can be also sampled as r,log or as i,log for real and integer.
values	This is the range or the set of values. It is given as (0,1) or as (a, b, c, d).
condition	If the condition is false, then no value is assigned to the parameter. If the condition is true, then the condition must be in the same syntax as the valid R logical expression. The condition may use the same name or also use different one as long as it does not have any cycles.

An example of the parameter file for the ILS can be seen in Figure 2.3. We have six parameters, of which four are independent (two parameters for the perturbation, one for the choice of the acceptance criterion, and one for the choice of the local search). In addition there are two conditional parameter, one for the temperature parameter and one for the *alpha* parameter. Recall that the temperature is a real parameter, which is as default 4 digits after the comma, and that the *alpha* is an integer.

```
# name            switch  type  values       [conditions (using R syntax)]
p_paramater        "-p "   i     (2, 98)
p_fixed_variable  "-z "   c     (0, 1)
acceptance         "-y "   c     (0, 1, 2, 3)
temperature_value "-f "   r     (0.0, 100.)  | Acceptance %in% c(3)
local_search       "-l "   c     (0, 1, 2, 3)
alpha              "-m "   i     (1, 100)     | Local_search %in% c(3)
```

Fig. 2.3 Parameter file (parameters.txt) for configuring ILS-GAP.

The `targetRunner` is the auxiliary program that actually runs the algorithm. The `targetRunner` is by default a minimization program. In the case of the GAP, where the default can be either a minimization or a maximization, it would have, if it is a maximization, returned to irace the result multiplied by -1. Yet, here we assume it is minimization. Then the `targetRunner` is made out this

```
<id.configuration> <id.instance> <seed>
           <instance> [bound] <configuration>
```

The information is given as follows.

id.configuration	unique identifier of the configuration.
id.instance	unique identifier of the instance.
seed	seed for the random number generation; ignore it for a deterministic algorithm.
instance	instance to be used.
bound	optional execution time bound.
configuration	the pairs of the label and values of the parameters that describe the configuration.

So, for example, for us the program to run is ./ilsgap and then comes the fixed parameters of the candidate configuration, the instance ID, the seed of the parameter setting, the instance that is run. Next, there would be a bound if it was an exact algorithm; but in this case it is not provided by irace as the target algorithm is a metaheuristic. Then comes the part of the parameters. An important part is that the program (./ilsgap in our case) has to put a real number as the output, which corresponds to the real cost measure in our case. The working directory of the targetRunner then is the running directory set by the option `execDir`, which allows to run different runs of irace if necessary. The target runner is in Figure 2.4 here in terms of a shell script.

Once we have the algorithm running, we need some training instances and some test instances. First, we have the training instances. The two options for irace are `trainInstancesDir` and `trainInstancesFile`, which is, however,

```
#!/bin/sh
CONFIG_ID=$1
INSTANCE_ID=$2
SEED=$3
INSTANCE=$4
CONFIG_PARAMS=$*

FIXED_PARAMS=" -t 3 "
STDOUT="c$CONFIG_ID-$INSTANCE_ID.stdout"
/home/stuetzle/Algorithms/GAP/GAP_new/ilsgap $FIXED_PARAMS \
                        -i $INSTANCE $CONFIG_PARAMS > $STDOUT
COST=$(tail -n 1 $STDOUT)
echo "$COST"
exit 0
```

Fig. 2.4 Target runner that configures the GAP for the ILS-GAP.

empty as a default. In our case, we take the `trainInstancesDir` which is `./GAP_instances/GAP_training`. We have generated instances of type C and D from Chu and Beasley, each of the instance classes with 100 and 200 clients for I with servers for 5, 10, and 20 for J. These are considered not too trivial for our purposes. For the C instances, we have generated the r_{ij} as integers from $U(5,25)$ and the c_{ij} as integer from $U(10,50)$ and as the bound have been generated as $b_i = 0.8 \cdot \sum_{j\in J} r_{ij}/m$. For the D instances, we have generated instances of r_{ij} as integers from $U(1,100)$ and c_{ij} are integers from $111 - r_{ij} + e$, where the e are integers from $U(-10,10)$. We again use $b_i = 0.8 \cdot \sum_{j\in J} r_{ij}/m$. For each size we have generated 5 instances so that we have 60 instances. We verified that all instances were feasible for the best of our training elements. As test instances, we have used the instances of type A, B, C, and type D that are available. We have not taken the instances A and B as training instances since they are known to be rather easy. As such, it is the 100 and 200 for I and the 5, 10, 20 for J. Note, that the size of the instances is the same; this could have also been made different for the training set but we leave this for later.

The irace has a number of further options. One is to have a text file that contains one or several initial configurations. In the case of the ILS on the GAP we can do so with starting from a random solution, iteration-best candidate with the better acceptance criterion, do a perturbation with 5 random moves, and a first-improvement local search with an alpha. Hence, we would have the following ILS-GAP scenario as in Figure 2.5.

```
## Initial candidate configuration for irace
initial  local search  perturbation  acceptance alpha
random   3             5             0          10
```

Fig. 2.5 Parameter file for specifying an initial configuration for ILS-GAP.

Other options would be to forbid configurations or the repair of configurations, which are however not needed in our case. Something that should be useful is to run irace in parallel, which we do here by imposing 30 runs in parallel. Another thing that is useful in irace is the testing of the configuration. Here it may be the final best configuration `testNbElites = 1`, which is also the default that is returned. An alternative maybe is the first best configuration of every iteration of irace, which is done by `testIterationElites = 1`. With this we can now run irace.

In the end, Figure 2.6 gives the outcome of the execution of irace. The figure presents the first part of the output file up to the end of the first iteration and the last part comprising the last iteration and the final result. We note that the parameters and the details are both understandable. In the lower part of the figure, it is given the best configuration found, which is configuration 692 with the perturbation parameter 3 with variable perturbation which varies between 2 to 3 perturbations, an acceptance criterion that is better or equal and a local search that is doing the first improvement local search. The two parameters temperature and alpha are not used. If one wants to do more analysis with irace, this can be done by the analysis of the results.

2.5 Related literature

In this chapter we have seean how to perform parameter configuration for a matheuristics. We first argue that many people have been aware that categorical and numerical parameters are important (Birattari et al., 2002; Hutter et al., 2009, 2011; López-Ibáñez et al., 2016) in addition to only the numerical ones (Audet and Orban, 2006; Nannen and Eiben, 2006; Yuan et al., 2012). We then have seen the parameter configuration problem in a first version as it was done by Birattari in his PhD thesis (Birattari, 2004). We have classified the field in five classes. First of all, we used experimental design techniques which have used by Coy et al. (2001); Montgomery (2012); Ridge and Kudenko (2010); Ruiz and Maroto (2005) among others. Out of these, the CALIBRA approach (Adenso-Díaz and Laguna , 2006) and the Coy et al. (2001) are well known and next we highlighted the MADS (Audet and Orban, 2006) and the REVAC ones as exemplary continuous techniques (Nannen and Eiben, 2006, 2007; Smit and Eiben, 2010). In our own work we have found CMAES to be among the best (Yuan et al., 2012, 2013). The majority of the systems fall in what we call heuristic search techniques. The first we know is that of Grefenstette from 1986. However, the mayority belongs to the first and second decade from this century and we may highlight ParamILS (Hutter et al., 2007b, 2009), gender-based genetic algorithms (Ansótegui et al. , 2009), linear genetic programming (Oltean, 2005), the OPAL system (Audet et al. , 2010), or the EVOCA evolutionary algorithm (Riff and Montero, 2013). In addition to these, we differentiate two final classes. The first are the surrogate-model based configurators of which we detail the SPOT system (Bartz-Beielstein et al., 2005) and the currently best performing SMAC system (Hutter et al., 2011). The second are methods that use some racing methods. The first is the F-race approach (Birattari et al., 2002), which is then developed into the

```
#------------------------------------------------------------------------------
# irace: An implementation in R of (Elitist) Iterated Racing
# Version: 3.4.1.9fcaeaf
# Copyright (C) 2010-2020
# Manuel Lopez-Ibanez     <manuel.lopez-ibanez@manchester.ac.uk>
# Jeremie Dubois-Lacoste
# Leslie Perez Caceres    <leslie.perez.caceres@ulb.ac.be>
#
# This is free software, and you are welcome to redistribute it under certain
# conditions.  See the GNU General Public License for details. There is NO
# WARRANTY; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# irace builds upon previous code from the race package:
#     race: Racing methods for the selection of the best
#     Copyright (C) 2003 Mauro Birattari
#------------------------------------------------------------------------------
# installed at: /home/stuetzle/R/x86_64-redhat-linux-gnu-library/3.5/irace
# called with: --exec-dir=execdir6 --parallel 32 --mpi 1 --seed 1
Warning: A default scenario file './scenario.txt' has been found and will be read
# 2020-09-03 11:33:14 CEST: Initialization
# Elitist race
# Elitist new instances: 1
# Elitist limit: 2
# nbIterations: 4
# minNbSurvival: 4
# nbParameters: 6
# seed: 1
# confidence level: 0.95
# budget: 5000
# mu: 5
# deterministic: FALSE

# 2020-09-03 11:33:14 CEST: Iteration 1 of 4
# experimentsUsedSoFar: 0
# remainingBudget: 5000
# currentBudget: 1250
# nbConfigurations: 208
# Markers:
     x No test is performed.
     c Configurations are discarded only due to capping.
     - The test is performed and some configurations are discarded.
     = The test is performed but no configuration is discarded.
     ! The test is performed and configurations could be discarded but elite configurations are preserved.
     . All alive configurations are elite and nothing is discarded

+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
| |   Instance|      Alive|       Best|      Mean best| Exp so far|  W time|  rho|KenW|  Qvar|
+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
|x|          1|        208|        180|    3667.000000|        208|00:06:40|   NA|  NA|    NA|
|x|          2|        208|        169|    3224.000000|        416|00:13:00|+0.93|0.97|0.0399|
|x|          3|        208|          2|    4428.333333|        624|00:03:16|+0.78|0.85|0.5011|
|x|          4|        208|        180|    6688.000000|        832|00:12:45|+0.78|0.84|0.4350|
|-|          5|        123|        180|    5596.800000|       1040|00:13:09|+0.73|0.78|0.4833|
|-|          6|        103|        180|    6929.500000|       1163|00:03:41|+0.69|0.74|0.4941|
+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
Best-so-far configuration:         180    mean value:     6929.500000
Description of the best-so-far configuration:
    .ID. perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha .PARENT.
180  180                      8                           1          0            1               NA    NA       NA


...


|.|         23|          4|        537|    6033.607143|         12|00:00:00|-0.02|0.01|0.7727|
|.|         16|          4|        537|    5923.172414|         12|00:00:00|-0.02|0.02|0.7678|
|.|          6|          4|        537|    6172.466667|         12|00:00:00|-0.02|0.01|0.7712|
+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
Best-so-far configuration:         692    mean value:     6171.900000
Description of the best-so-far configuration:
    .ID. perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha .PARENT.
692  692                      3                           1          1            1               NA    NA      605

# 2020-09-03 17:46:39 CEST: Elite configurations (first number is the configuration ID; listed from best to worst
according to the mean value):
    perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha
692                      3                           1          1            1               NA    NA
709                      3                           1          1            1               NA    NA
537                      2                           1          0            1               NA    NA
605                      3                           1          1            1               NA    NA
# 2020-09-03 17:46:39 CEST: Stopped because there is not enough budget left to race more than the minimum (4)
# You may either increase the budget or set 'minNbSurvival' to a lower value
# Iteration: 13
# nbIterations: 13
# experimentsUsedSoFar: 4972
# timeUsed: 0
# remainingBudget: 28
# currentBudget: 28
# number of elites: 4
# nbConfigurations: 4
# Best configurations (first number is the configuration ID; listed
from best to worst according to the mean value):
    perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha
692                      3                           1          1            1               NA    NA
709                      3                           1          1            1               NA    NA
537                      2                           1          0            1               NA    NA
605                      3                           1          1            1               NA    NA
# Best configurations as commandlines (first number is the configuration ID; same order as above):
692  -p 3 -z 1 -y 1 -l 1
709  -p 3 -z 1 -y 1 -l 1
537  -p 2 -z 1 -y 0 -l 1
605  -p 3 -z 1 -y 1 -l 1
```

Fig. 2.6 Parameter file (parameters.txt) for configuring ILS-GAP.

irace systems (López-Ibáñez et al., 2016; Pérez Cáceres et al., 2017). Of these methods we give some details of ParamILS (Hutter et al., 2009), SMAC (Hutter et al., 2011) and of irace (López-Ibáñez et al., 2016).

In section 2.3, we have classified the advantages of the automatic algorithm configuration. We first have shown that (Hutter et al., 2007a) have excellent performance for a verifications procedure and the same for mixed-integer programming procedures (Hutter et al., 2010). Similar results were obtained by F-race (Birattari et al., 2002), REVAC (Nannen and Eiben, 2007; Smit and Eiben, 2010), or with irace (Balaprakash et al., 2010; Liao et al., 2013, 2011; Pérez Cáceres et al., 2015). Other advantages are that sometimes old algorithms are getting surprisingly good results (Franzin and Stützle, 2016). When doing algorithm (re-)engineering, automatic algorithm configurations software can also be very useful (Montes de Oca et al., 2011; Hartung and Hoos, 2015; Balaprakash et al., 2010, 2015). The last effort is also the most used one. We divide this into top-down and bottom-up approaches. One of the first is the SATenstein approach (KhudaBukhsh et al., 2009, 2016). There is a lot also done with single objective algorithms like in Liao et al. (2014); Aydın et al. (2017); Franzin and Stützle (2019) or also multi-objective approaches (López-Ibáñez and Stützle, 2010, 2012; Dubois-Lacoste et al., 2011; Bezerra et al., 2016, 2018, 2020). This top-down approach is feasible, if the algorithm to be used is rather straight-forward. As an alternative, one can use the bottom-up approach. This was done again with the SAT approach first (Fukunaga, 2004, 2008). In general, it is common if following the hyper-heuristic settings (Burke et al., 2013, 2019) as can be seen with a few settings (Burke et al., 2012; Sabar et al., 2015). In a different approach, we used irace to do this grammar-settings. In a first approach, we used Paradiseo as algorithmic framework (López-Ibáñez et al., 2017). This system was redesigned and currently it has the best results for a number of flowshop problems (Pagnozzi and Stützle, 2019; Alfaro-Fernández et al. , 2020).

Finally, we used irace to configure a small example for an ILS for the GAP. For more details on the irace package, we refer to the irace user guide.

References

B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114, 2006.

P. Alfaro-Fernández, R. Ruiz, F. Pagnozzi, and T. Stützle. Automatic algorithm design for hybrid flowshop scheduling problems. *European Journal of Operational Research*, 282(3):835–845, 2020.

C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming, CP 2009*, volume 5732 of *LNCS*, pages 142–157. Springer, 2009.

C. Audet and D. Orban. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization*, 17(3):642–664, 2006.

C. Audet, C.-K. Dang, and D. Orban. Algorithmic parameter optimization of the DFO method with the OPAL framework. In K. Naono, K. Teranishi, J. Cavazos, and R. Suda, editors, *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, pages 255–274. Springer, 2010.

D. Aydın, G. Yavuz, and T. Stützle. ABC-X: a generalized, automatically configurable artificial bee colony framework. *Swarm Intelligence*, 11(1):1–38, 2017.

P. Balaprakash, M. Birattari, T. Stützle, and M. Dorigo. Estimation-based metaheuristics for the probabilistic travelling salesman problem. *Comput. Oper. Res.*, 37(11):1939–1951, 2010.

P. Balaprakash, M. Birattari, T. Stützle, and M. Dorigo. Estimation-based metaheuristics for the single vehicle routing problem with stochastic demands and customers. *Computational Optimization and Applications*, 61(2):463–487, 2015.

T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *IEEE CEC*, pages 773–780, Piscataway, NJ, September 2005. IEEE Press.

L. C. T. Bezerra, M. López-Ibáñez, and T. Stützle. Automatic component-wise design of multi-objective evolutionary algorithms. *IEEE Trans. Evol. Comput.*, 20(3):403–417, 2016.

L. C. T. Bezerra, M. López-Ibáñez, and T. Stützle. A large-scale experimental evaluation of high-performing multi- and many-objective evolutionary algorithms. *Evol. Comput.*, 26(4):621–656, 2018.

L. C. T. Bezerra, M. López-Ibáñez, and T. Stützle. Automatically designing state-of-the-art multi- and many-objective evolutionary algorithms. *Evol. Comput.*, 28 (2):195–226, 2020.

M. Birattari. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, IRIDIA, Université Libre de Bruxelles, Belgium, 2004.

M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editors, *Proceedings of GECCO 2002*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

E. K. Burke, M. R. Hyde, and G. Kendall. Grammatical evolution of local search heuristics. *IEEE Trans. Evol. Comput.*, 16(7):406–417, 2012.

E. K. Burke, M. Gendreau, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu. Hyper-heuristics: A survey of the state of the art. *J. Oper. Res. Soc.*, 64(12):1695–1724, 2013.

E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches: Revisited. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, chapter 14, pages 453–477. Springer, 2019.

S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil. Using experimental design to find effective parameter settings for heuristics. *J. Heuristics*, 7(1):77–97, 2001.

J. Dubois-Lacoste, M. López-Ibáñez, and T. Stützle. Automatic configuration of state-of-the-art multi-objective optimizers using the TP+PLS framework. In N. Krasnogor and P. L. Lanzi, editors, *Proc. of GECCO 2011*, pages 2019–2026. ACM Press, New York, NY, 2011.

A. Franzin and T. Stützle. Exploration of metaheuristics through automatic algorithm configuration techniques and algorithmic frameworks. In T. Friedrich, F. Neumann, and A. M. Sutton, editors, *GECCO (Companion)*, pages 1341–1347. ACM Press, New York, NY, 2016.

A. Franzin and T. Stützle. Revisiting simulated annealing: A component-based analysis. *Comput. Oper. Res.*, 104:191 – 206, 2019.

A. S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In K. Deb et al., editors, *Proc. of GECCO 2004, Part II*, volume 3103 of *LNCS*, pages 483–494. Springer, 2004.

A. S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evol. Comput.*, 16(1):31–61, March 2008.

J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proc. of FMCAD'07*, pages 27–34, Austin, Texas, USA, 2007a. IEEE Computer Society, Washington, DC, USA.

F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In R. C. Holte and A. Howe, editors, *Proc. of AAAI '07*, pages 1152–1157. AAAI Press/MIT Press, Menlo Park, CA, 2007b.

F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *J. Artif. Intell. Res.*, 36:267–306, October 2009.

F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In A. Lodi, M. Milano, and P. Toth, editors, *Proc. of CPAIOR 2010*, volume 6140 of *LNCS*, pages 186–202. Springer, 2010.

F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In C. A. Coello Coello, editor, *LION 5*, volume 6683 of *LNCS*, pages 507–523. Springer, 2011.

S. Hartung and H. H. Hoos. Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing. In C. Dhaenens, L. Jourdan and M.-E. Marmion, editors, *LION 9*, volume 8994 of *LNCS*, pages 43–58. Springer, 2015.

A. R. KhudaBukhsh, Lin Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In C. Boutilier, editor, *Proc. IJCAI-09*, pages 517–524. AAAI Press, Menlo Park, CA, 2009.

A. R. KhudaBukhsh, Lin Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT Solvers from Components. *Artificial Intelligence*, 232:20–42, 2016.

T. Liao, M. A. Montes de Oca, and T. Stützle. Tuning parameters across mixed dimensional instances: A performance scalability study of Sep-G-CMA-ES. In N. Krasnogor and P. L. Lanzi, editors, *GECCO (Companion)*, pages 703–706, New York, NY, 2011. ACM Press.

T. Liao, M. A. Montes de Oca, and T. Stützle. Computational results for an automatically tuned CMA-ES with increasing population size on the CEC'05 benchmark set. *Soft Computing*, 17(6):1031–1046, 2013.

T. Liao, T. Stützle, M. A. Montes de Oca, and M. Dorigo. A unified ant colony optimization algorithm for continuous optimization. *Eur. J. Oper. Res.*, 234(3): 597–609, 2014.

M. López-Ibáñez and T. Stützle. Automatic configuration of multi-objective ACO algorithms. In M. Dorigo et al., editors, *ANTS 2010*, volume 6234 of *LNCS*, pages 95–106. Springer, 2010.

M. López-Ibáñez and T. Stützle. The automatic design of multi-objective ant colony optimization algorithms. *IEEE Trans. Evol. Comput.*, 16(6):861–875, 2012.

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

M. López-Ibáñez, M-E. Kessaci, and T. Stützle. Automatic design of hybrid metaheuristics from algorithmic components. Technical Report TR/IRIDIA/2017-012, IRIDIA, Université Libre de Bruxelles, Belgium, 2017.

M. A. Montes de Oca, D. Aydın, and T. Stützle. An incremental particle swarm for large-scale continuous optimization problems: An example of tuning-in-the-loop (re)design of optimization algorithms. *Soft Computing*, 15(11):2233–2255, 2011.

D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, New York, NY, eighth edition, 2012.

V. Nannen and A. E. Eiben. A method for parameter calibration and relevance estimation in evolutionary algorithms. In M. Cattolico et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2006*, pages 183–190. ACM Press, New York, NY, 2006.

V. Nannen and A. E. Eiben. Relevance estimation and value calibration of evolutionary algorithm parameters. In M. M. Veloso, editor, *Proceedings of IJCAI-07*, pages 975–980. AAAI Press, Menlo Park, CA, 2007.

M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evol. Comput.*, 13(3):387–410, 2005.

F. Pagnozzi and T. Stützle. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. *Eur. J. Oper. Res.*, 276:409–421, 2019.

L. Pérez Cáceres, M. López-Ibáñez, and T. Stützle. Ant colony optimization on a limited budget of evaluations. *Swarm Intelligence*, 9(2-3):103–124, 2015.

L. Pérez Cáceres, M. López-Ibáñez, H. H. Hoos, and T. Stützle. An experimental study of adaptive capping in irace. In R. Battiti, D. E. Kvasov, and Y. D. Sergeyev, editors, *LION 11*, volume 10556 of *LNCS*, pages 235–250. Springer, Cham, Switzerland, 2017.

E. Ridge and D. Kudenko. Tuning an algorithm using design of experiments. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 265–286. Springer, Berlin, Germany, 2010.

M-C. Riff and E. Montero. A new algorithm for reducing metaheuristic design effort. In *Proceedings of CEC 2013*, pages 3283–3290. IEEE Press, Piscataway, NJ, 2013.

R. Ruiz and C. Maroto. A comprehensive review and evaluation of permutation flowshop heuristics. *Eur. J. Oper. Res.*, 165(2):479–494, 2005.

N. R. Sabar, M. Ayob, G. Kendall, and R. Qu. Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems. *IEEE Trans. Evol. Comput.*, 19(3):309–325, 2015.

S. K. Smit and A. E. Eiben. Beating the 'world champion' evolutionary algorithm via REVAC tuning. In H. Ishibuchi et al., editors, *Proceedings of CEC 2010*, pages 1–8. IEEE Press, Piscataway, NJ, 2010.

Z. Yuan, M. A. Montes de Oca, T. Stützle, and M. Birattari. Continuous optimization algorithms for tuning real and integer algorithm parameters of swarm intelligence algorithms. *Swarm Intelligence*, 6(1):49–75, 2012.

Z. Yuan, M. A. Montes de Oca, T. Stützle, H. C. Lau, and M. Birattari. An analysis of post-selection in automatic configuration. In Christian Blum and Enrique Alba, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2013*, pages 1557–1564. ACM Press, New York, NY, 2013.