Adaptive and Multilevel Metaheuristics

1

Marc Sevaux, Kenneth Sörensen, and Nelishia Pillay

Contents

M. Sevaux (✉)
Université de Bretagne-Sud, Lab-STICC, CNRS, Lorient, France
e-mail: marc.sevaux@univ-ubs.fr

K. Sörensen
University of Antwerp, Antwerp, Belgium
e-mail: kenneth.sorensen@uantwerpen.be

N. Pillay
School of Mathematics, Statistics, and Computer Science, University of KwaZulu-Natal, Durban, South Africa
e-mail: pillayn32@ukzn.ac.za

R. Martí et al. (eds.), *Handbook of Heuristics*,
https://doi.org/10.1007/978-3-319-07124-4_16

Abstract

For the last decades, metaheuristics have become ever more popular as a tool to solve a large class of difficult optimization problems. However, determining the best configuration of a metaheuristic, which includes the program flow and the parameter settings, remains a difficult task. Adaptive metaheuristics (that change their configuration during the search) and multilevel metaheuristics (that change their configuration during the search by means of a metaheuristic) can be a solution for this. This chapter intends to make a quick review of the latest trends in adaptive metaheuristics and in multilevel metaheuristics.

Keywords

Metaheuristics · Multilevel · Adaptive · Configuration · Hyper-heuristics

Introduction

Metaheuristics are flexible frameworks that can be used to design heuristics for virtually any combinatorial optimization problem. This flexibility also comes at a cost: many researchers in metaheuristics spend a large amount of time to properly design and tune their algorithm in a trial-and-error fashion. As mentioned and observed in many published papers, designing an efficient metaheuristic is an art, requiring a lot of intuition on the part of the metaheuristic designer. There is no doubt, however, that the parameters and the structure of the metaheuristic may influence the performance of the solution approach and the quality of the results in the end.

In the design of a metaheuristic, a large fraction of the time is usually spent on determining the *control flow*, i.e., the order in which the different components of the metaheuristic are used and the optimal levels of the various parameters of the metaheuristic. A more structured approach than the one commonly used is wanted. The purpose of this chapter is to discuss one of the ways to alleviate this problem, through adaptive and multilevel metaheuristics.

Our overview is necessarily very short and cannot replace the many years of research, the large number of books and papers that have tried to clarify the topic. We encourage the reader to address the existing work that we point out and explore the references that we may have missed here. This chapter is not a complete review of all papers in the field of adaptive or multilevel metaheuristics but rather a set of good practices that can be done when designing metaheuristics.

In a first section, we will attempt to clearly define some concepts to come to a definition of the terms *adaptive* and *multilevel* (section "Definitions"). We will also discuss the impact of heuristic parameters on the behavior of a metaheuristic algorithm (section "Configuring a Metaheuristic") and how we can reduce the number of parameters or how we can automatically tune some parameters. Adaptive metaheuristics are discussed in section "Adaptive Metaheuristics", and multilevel metaheuristics and hyper-heuristics in section "Multilevel Metaheuristics and Hyper-heuristics". Section "Conclusion" will conclude the chapter.

Definitions

The terms *adaptive* and *multilevel* do not have a single, generally accepted definition within the metaheuristics community. The aim of this section is to develop clear definitions for both. Both *adaptive* and *multilevel* refer to the evolution of the *configuration* of a metaheuristic algorithm during the optimization process. We must therefore first define the term *configuration*. Our definitions will be intuitive rather than formal.

Each metaheuristic algorithm consists of several *components*, i.e., parts that form a more or less logical and atomic unit. Examples are a local search operator in a variable search algorithm, a tabu list in a tabu search algorithm, a crossover operator or a selection operator in an evolutionary algorithm. Each of these components can exist more or less independently of the rest of the metaheuristic algorithm, which includes their use in a different metaheuristic algorithm for the same problem. Within the same metaheuristics, components can often be rearranged in the overall structure of the algorithm. For example, local search operators in a variable neighborhood search algorithm can be executed in a specific order; an evolutionary algorithm may use its selection operator before or after the crossover operator (or both), etc. In programming, this is called the *control flow*, a term we will adopt here. Clearly, not every control flow makes sense, but generally speaking, a sizeable number of possibilities exists. Determining the control flow of the algorithm, i.e., the order in which the components are executed is a task for the algorithm designer.

Each component may have one or more parameters that determine its functioning. Such parameters may be numerical, such as the tabu tenure of a tabu list, the number of iterations without improvement before a perturbation move is used, etc. Other parameters may be nonnumerical, like the choice of a discrete set of move strategies to use in a local search operator (steepest descent, mildest descent, random improving, ...). The algorithm designer usually defines a finite set of potential (or sensible) values, e.g., the restricted candidate list of a GRASP algorithm is defined to be an integer number between 5 and 20. In some situations, a parameter might also be a real number.

Using these concepts, we may now define the term *configuration*.

Definition 1. Given the set of components of a metaheuristic algorithm together with the set of all potential control flow alternatives, as well as its parameters and their potential values, the **configuration** of a metaheuristic algorithm defines the specific control flow and the specific set of parameter values it uses.

For example, a local search-based metaheuristic like the most basic tabu search method depicted in Algorithm 1 has one parameter, the tabu tenure (the length of the tabu list), and needs one type of neighborhood. It also needs several functions such as *initialize* or *update memory*. One specific tabu tenure, one type of neighborhood and the set of necessary functions will be the configuration of the current search method.

Algorithm 1: Basic tabu search

initialise: find an initial solution x
repeat
  neighbourhood search: find a solution $x' \in \mathcal{N}^*(x)$
  update memory: tabu list, frequency-based memory, aspiration level, ...
  move $x \leftarrow x'$
until *stopping criterion satisfied*

For a simple variable neighborhood search heuristic presented in Algorithm 2, the order in which neighborhoods are explored (or whether they are explored at all), the way in which the starting solution in the current neighborhood is generated and the local search method that is applied to improve the solution with the current neighborhood, as well as the total number of iterations k_{max}, define the configuration of the VNS.

Algorithm 2: Basic variable neighbourhood search

initialise: find an initial solution x, k $\leftarrow 1$
repeat
  shake: generate a point x' at random from the neighbourhood $\mathcal{N}_k(x)$
  local search: apply a local search procedure starting from the solution x' to find a solution x''
  if x'' *is better than* x **then**
    $x \leftarrow x''$ and $k \leftarrow 1$ (centre the search around x'' and search again with neighbourhood 1)
  else
    $k \leftarrow k + 1$ (enlarge the neighbourhood)
until $k = k_{max}$

We can now define the terms *adaptive* and *multilevel* in the context of metaheuristic algorithms.

Definition 2. A metaheuristic is **adaptive** when it includes a mechanism to modify its configuration *during* its execution.

In other words, an adaptive heuristic includes a mechanism to modify either the control flow or the parameter values (or both) of a heuristic, and, by doing so modify the behavior of the metaheuristic. As an example, consider the basic tabu search shown in Algorithm 1. A common adaptive mechanism might make some changes to the *update memory*: the length of the tabu list, the aspiration level, etc.

Even though the definition has been conceived to be as watertight as possible, there is always room for interpretation. For example, considering the VNS heuristic in Algorithm 2, the basic design of the heuristic is such that the order in which the neighborhoods are searched depends on the instance being solved. Since the heuristic will move to the next neighborhood once a local optimum has been reached (and local optima are different for every instance), the control flow of the algorithm will be different for every instance. Yet, most researchers would not call a simple VNS heuristic adaptive.

The mechanism that does the actual adaptation can range from very simple to complex. Essentially, the aim of the mechanism is to search for the best configuration of the metaheuristic algorithm. This search can itself be seen as a combinatorial optimization problem, and an optimization algorithm may be used to solve it. When the adaptation mechanism itself is a metaheuristic algorithm, we call the overall result a *multilevel metaheuristic*.

Definition 3. A **multilevel** metaheuristic algorithm is a metaheuristic algorithm for which the configuration is altered by another metaheuristic algorithm.

Note that, since the algorithm doing the adapting is itself a heuristic, with its own components and parameters, the road is paved for a recursive structure in which the configuration of the lowest-level heuristic is adapted by a higher-level metaheuristic; the configuration of which is adapted by an even higher-level heuristic, the configuration of which is adapted by ..., ad infinitum. However, the complexity added by implementing a higher-level metaheuristic algorithm to adapt the configuration of a lower-level metaheuristic is usually considerable, which precludes the design of a multilevel heuristic having many levels.

For reasons of clarity, a hybrid metaheuristic, i.e., a metaheuristic algorithm that combines ideas from several metaheuristic frameworks, for which the configuration remains unchanged throughout the search, is not considered a multilevel metaheuristic in this chapter. Also, the term multilevel is often used to denote optimization problems that can decompose into several (simpler) problems (e.g., a location–routing problem can often be decomposed into a location problem and a routing problem). Each of the problems may be separately solved by a metaheuristic algorithm. In this chapter, we do not use the term multilevel metaheuristics to describe such approaches.

Configuring a Metaheuristic

An optimization method and especially a metaheuristic has several (potentially hundreds) possible configurations. Among all of them, only a few will allow the search to reach the optimal solution or the best possible solution, but not for all instances and not at all time. The configuration may have a great influence on the quality of the final solution or the effectiveness of the search method. This is the reason why properly tuning the parameters or choosing the right configuration is

a very important task. Because of the "no free lunch theorem" [35], we know that there is no optimal configuration of a metaheuristic that will outperform all others on all problems and all instances.

Despite this fact, it is necessary to find an initial configuration suitable to solve the problem at hand. This initial part will be described in this section. If the configuration is not satisfying, instead of starting again the resolution with a new configuration, one can change the configuration during the search (see section "Adaptive Metaheuristics").

As noted by several researchers (e.g., [14]), the first step of setting up a metaheuristic has to go through the configuration phase during which the control flow is established and the parameters are tuned. This phase is sometimes called "offline parameter initialization." This is long and fastidious and usually done with trial-and-error methods. Moreover, even when this step is completed, its efficiency is often effective on a subset of instances (usually close to the instances on which the parameters have been calibrated). In addition, as already mentioned for metaheuristics, the parameters are not only numerical values but can be search components, updating function, etc. [32].

As a general observation, tuning these parameters is often so difficult that the designers change them one by one until they get the right configuration. And the value of these parameters is obtained empirically. Hence, the final combination of the parameters deduced from a sequential empirical adaptation of the parameters cannot guarantee that the final configuration is optimal. Furthermore, by changing the parameters one by one, it is impossible (or too difficult) to detect the possible interactions between these parameters. Moreover, the parameters and the control flow of a heuristic generally heavily influence each other, which makes the process of determining both even more difficult.

Eiben et al. [17] clearly define the parameter tuning for evolutionary algorithms: *By parameter tuning we mean the commonly practiced approach that amounts to finding good values for the parameters before the run of the algorithm and then running the algorithm using these values, which remain fixed during the run.*

Once this initial tuning phase is completed, most metaheuristic designers keep the configuration as it is to run their solution approach on the set of instances studied. This configuration remains the same (as cited above) until the designer believes that it is not adapted anymore and should be reconfigured with the same process.

Among the potential methods for tuning the parameters before solving, one can list:

- Manual tuning (usually from the experience of a metaheuristic designer),
- Parameter tuning on a subset of representative instances,
- Automatic parameter tuning by the use of an external method.

An experienced metaheuristic designer is often able to decide the value of a large number of parameters beforehand. The rules of thumb prevail on every other considerations. The reason for this is that after several years of practicing, one can

know that some parameters need to have certain values, and the values that are close to it will not make a big difference on the final effectiveness of the results.

The population of a genetic algorithm is a rather good example. A small population will have a premature convergence because too many individuals will be the same (clones), and to avoid this without a specific mechanism, it is important to have a population of a large size. Many papers on genetic algorithms have a population of 100 individuals, but none mentioned how they have obtained this value, or the motivations to set it to this value. No analysis is done to see if 95 or 105 will give better results.

Only a few researchers report the difficulty of finding the right parameter settings and the limitations of this kind of approach [34]. Moreover, the manual tuning, without post-analysis experiments, has the drawback of not being applicable to different instances than the one presented in the paper. This is even more the case for transferring the method to any similar industrial application.

Whatever the technique used for tuning these parameters or configurations beforehand, it is important to keep in mind that every metaheuristic should be well balanced between *intensification* and *diversification* [31]. Hence, the tuning of the parameters should take this into account for ensuring that the metaheuristic is not converging too fast (too much intensification or exploitation) or is wandering in the search space without converging (too much diversification or exploration). Of course, for being able to detect this, one has to set up some indicators showing the speed of convergence, the evolution of the solution quality, the evolution of the solutions themselves, etc.

The best practice is to report the results as Prins [26] has done in his paper on the vehicle routing problem. In that paper, the tables present at the same time the results obtained on a set of instances with some "standard parameter settings" and the results obtained with the "best parameter settings." This is a fair comparison to existing work. The only drawback is the missing information on the time needed to set up the standard parameter settings as well as the best parameter settings. This can be a long and fastidious task and the total computational time can be high.

Setting up the parameters of a metaheuristic based on preliminary experiments is probably the most common technique used in designing metaheuristics. A subset of representative instances is selected, and the parameters are tested on these instances until they converge to stable results. Hence, they are applied to the whole set of instances, and the results are reported. The subset of instances should be carefully selected to be representative of the future experiments.

Usually, the designer selects one parameter at a time, adjusts its value to the best one, and reiterates with the next parameter. Only a few reports that they practice a full factorial design as stated by Hooker [19]. With such a design, authors may try to understand the relative contribution of each parameter to the global effectiveness of the metaheuristic and the possible influence of the parameters between them. One of the best examples of the application of this type of parameter design is presented by Xu and Kelly [38] on a tabu search algorithm. In their paper, they have selected a small subset of seven instances to tune five components of the tabu search. A more

general approach is presented by Xu et al. [39] but still based on the same factorial analysis.

Another technique to find the right parameter settings is to let an external method to tune the parameters for the metaheuristic designers. Very few studies exist on that issue if they are not used intrinsically during the metaheuristic search. Dean and Voss [10] in their book present a technique known as the *response surface methodology* in statistics. This method has been effectively used in [1]. This technique consists in running a local search method in the space of the parameters. A specific metric measuring the distance between each pair of parameters is calculated by running the metaheuristic. For a fixed setting of parameters (or a point in the search space of the parameters), the neighbors are also explored. If no better neighbor can be found, the value for the parameters is fixed, and the search stops; otherwise, the search continues with the best neighbor and the new parameter settings.

Of course, one cannot guarantee the optimality for all the parameters at once and even at the end of the search. But usually, this technique is able to discover good parameter settings. One important drawback is the definition of the metric that is very sensitive, especially if an order cannot be defined on the variables.

For a more elaborate discussion on this topic, we refer the reader to [3] and to section "Hyper-heuristics for Metaheuristic Configuration" in this chapter.

Adaptive Metaheuristics

As stated in the previous section, once a designer has the best parameter settings for its metaheuristics, he is able to run it confidently on the set of targeted instances and produce results. The question is: "Can it go further?" And the answer is yes. Yes, there is always some space for improvement. A parameter setting that works very well on one instance might work poorly on another one.

To overcome this difficulty, it is always possible to analyze the behavior of the metaheuristic during the search and adapt it to obtain better results. This phase can be named "online parameter tuning." Based on indicators (e.g., convergence, solution quality, similarity of explored solutions), the configuration of the metaheuristic is changed. This technique is particularly appealing when one has to solve only one large instance, and the tuning of parameters cannot be done beforehand.

Simple Adaptive Mechanisms

Detecting why a metaheuristic is not giving satisfying results is not an easy task. It largely depends on the type of metaheuristic itself. For example, identical individuals (clones) in a genetic or memetic algorithm are one of the known consequences of premature convergence. In a local search method, cycling in the objective space or in the solution space is also a situation that needs to be avoided.

The online parameter tuning can be simple as in [30] where a tabu search procedure is having a *cycle detection mechanism* and increases the tabu tenure along the search when cycles are detected. To really improve the behavior of a metaheuristic algorithm, however, more elaborate methods can be used during the search and exploited for a better efficiency.

Such a mechanism includes that of Boutillon et al. [5] that can be activated during the search like *retroactive loops* where a simulated annealing temperature parameter is controlled during the search to follow a predefined probability acceptance decreasing scheme.

Reactive Search

Among all existing methods, the work of Battiti [2] had traced the path a long time ago. In the most simple reactive search, the past history of the search is intensely used for *feedback-based parameter tuning* and for *automated balance of diversification and intensification*. In the former, the tuning of parameters is automated, and decisions on the new values of parameters are made based on the past events of the search. In the latter, the concept of balancing exploration *vs.* exploitation is used to guide the search.

One of the simplest forms of reactive search is reactive tabu search. The main idea is to change the length of the tabu list (i.e., the tabu tenure) based on the search trajectory. Essentially, the tabu list is made longer if the search is not finding better solutions, and shorter if it does.

Greedy Randomized Adaptive Search Procedure

GRASP (greedy randomized adaptive search procedure) is a constructive metaheuristic, the main idea of which is to balance greediness and randomness. Many constructive heuristics are greedy, which means that, at every iteration, they pick the best element from the set of potential solution elements. An example is the nearest-neighbor heuristic for the TSP which starts from a given city and moves to the closest unvisited city at every iteration. The drawback of a fully greedy heuristic is that it only generates a single solution, which is most likely suboptimal. A wrong decision early on in the constructive procedure may lead to bad solutions in the end.

GRASP attempts to overcome this drawback by introducing randomness into the solution construction process. Instead of picking the best element at each iteration, GRASP creates a *restricted candidate list*, i.e., a list of the α best elements and picks one element from this list *at random* (α represents a number of elements). By doing this, GRASP generates a different solution at each iteration. After several iterations, some solutions will likely have been found that are better than the one found by a purely greedy heuristic.

In *reactive* GRASP, introduced by Prais and Ribeiro [25], the parameter α is randomly chosen from a set of discrete value. Initially, each possible α_i has

the same probability of being chosen. The search remembers the quality of the solutions found for each possible value of α_i. After some iterations, the probabilities of selecting α_i are updated to reflect the quality of the solutions it produced. The probabilities corresponding to α_i's that have resulted in good solutions are increased; the others are decreased.

The reactive GRASP by Delorme et al. [11], e.g., works as follows. A *value* λ_i is defined for each α_i. The probability of selecting α_i, p_i is calculated as follows:

$$p_i = \frac{\lambda_i}{\sum_{k=1}^{n} \lambda_k},$$

supposing that n different α_i's have been defined.

Whenever a good solution is found using a certain α_i, this solution is added to the *pool* P_i for this α_i. Periodically, the values of λ_i (and hence p_i) are updated according to the following formula:

$$\lambda_i = \left(\frac{\text{mean}_x \in P_i [f(x) - f(\underline{x})]}{f(\overline{x}) - f(\underline{x})} \right)^{\delta},$$

where $\underline{x}$ and $\overline{x}$ are the worst and best solutions found so far and δ is a parameter introduced to attenuate the update of the probabilities p_i.

Adaptive Large Neighborhood Search

Large neighborhood search (LNS) is a constructive metaheuristic that works by building solutions from their constituting elements. For this purpose, it relies on a set of simple constructive procedures to build solutions and on a set of destructive procedures to partially destroy these solutions so they can be rebuilt. For this reason, LNS has also been called ruin-and-recreate. At each iteration, LNS selects a pair consisting of one destructive heuristic and one constructive heuristic. Using this pair, a new solution is obtained. Most LNS implementations use a probabilistic mechanism to select both the destructive and the constructive heuristic at each iteration. A (nonadaptive) LNS algorithm could, e.g., assign equal probabilities to each constructive heuristic and to each destructive heuristic.

Adaptive large neighborhood search goes a step further by selecting the heuristic pair with a probability determined by the previous performance of both the constructive and the destructive heuristics. As a result, constructive and destructive heuristics that perform well will have a higher probability of being selected, whereas those that will not have a lower probability. Usually, however, the probabilities are bounded by some values that ensure all heuristics have at least a tiny chance of being selected.

An example of an ALNS adaptive constructive/destructive heuristic selection mechanism is the following. Suppose an ALNS heuristic has n constructive and m destructive heuristics. Initially, each constructive heuristic i (and each destructive

heuristic j) is assigned a *value* $\lambda_i^c = 1$ ($\lambda_j^d = 1$). Then, a constructive heuristic is selected from the set of constructive heuristics with a probability p proportional to its value:

$$p_i = \frac{\lambda_i}{\sum_{k=1}^{n} \lambda_k}$$

Similarly, a destructive heuristic is selected according to an equivalent rule.

Using the pair of destructive and constructive heuristic, a new solution is generated. The quality of the new solution is evaluated, and the values of the selected constructive and destructive heuristic are updated.

$$\alpha = \begin{cases} 0.5 & \text{if the new solution is worse than the current solution} \\ 1.5 & \text{if the new solution is better than the current solution} \\ 2 & \text{if the new solution is better than the global best solution} \end{cases}$$

$$\lambda_{i,\text{new}}^c = \gamma \lambda_i^c + (1 - \gamma)\alpha$$

$$\lambda_{j,\text{new}}^d = \gamma \lambda_j^d + (1 - \gamma)\alpha$$

where γ is a parameter between 0 and 1.

Using the formulas above, the probabilities of selection for each constructive and destructive heuristic will adapt to the problem at hand. Moreover, using the formulas above, the values can never increase above 2 and never drop below 0.5. In other words, all constructive and destructive heuristics will keep having a positive probability of being selected, even if they consistently fail to find improving solutions.

Multilevel Metaheuristics and Hyper-heuristics

In the previous section, we have examined different techniques for adapting the configuration of metaheuristics. In this section, we look at multilevel metaheuristics, i.e., metaheuristic algorithms for evolving the configuration of a metaheuristic. Much of the research in this area has used evolutionary algorithms to configure metaheuristics with one of the earliest studies being that conducted by Bölte and Thonemann [4] which uses genetic programming for generating annealing schedules, which were previously manually created in a simulated annealing algorithm to solve the quadratic assignment problem. Various evolutionary algorithms, namely, genetic algorithms, evolutionary strategies, and estimation of distribution algorithms, have been used for parameter tuning of evolutionary algorithms [16]. These are referred to as meta-EAs and operate on the design level, while the EA solving the problem at hand is considered to form the algorithm layer. Hyper-heuristics are proving to be effective for the automatic configuration of metaheuristics, and we provide an overview of the use of hyper-heuristics for this purpose.

Hyper-heuristics

Hyper-heuristics were initially introduced as "heuristics to choose heuristics" [6,28]. Hyper-heuristics aim at providing a more generalized solution for a problem domain rather than producing the best solution for certain problem instances. This is achieved by exploring a space of low-level heuristics rather than a solution space directly. The low-level heuristics can be constructive or perturbative. Constructive low-level heuristics are used to create an initial solution, while perturbative heuristics are used to improve an existing candidate solution. The first generation of hyper-heuristics was essentially selection hyper-heuristics which chose which constructive or perturbative heuristics to use at a particular point in constructing or improving a candidate solution, respectively.

Selection constructive hyper-heuristics are used to select the low-level construction heuristic at each stage in constructing a solution. Similarly, in the case of selection perturbative hyper-heuristics, the hyper-heuristic chooses a perturbative low-level heuristic at each stage of the improvement process. We use an application of hyper-heuristics to the domain of examination timetabling to illustrate these concepts. Low-level construction heuristics generally used to solve examination timetabling problems are the graph coloring heuristics, namely, largest degree, largest weighted degree, largest color degree, largest enrollment, and saturation degree [27]. Each of these heuristics assesses the difficulty associated with allocating an examination to the timetable. For example, the saturation degree heuristic is the number of timetable slots, given the current state of the timetable at the particular point in construction, an examination can be allocated to without causing hard constraint violations such as a student being scheduled to write more than one examination at the same time. A selection hyper-heuristic chooses which of the low-level heuristics to use to schedule each of the examinations. This has proven to be effective as different low-level heuristics work well for different problem instances, and more importantly, different low-level heuristics are more effective at different points of solution construction. Metaheuristics such as simulated annealing, tabu search, variable neighborhood search, and genetic algorithms have generally been used to search the heuristic space [6,9].

Examples of low-level perturbative heuristics for the examination timetabling domain include swapping examinations, swapping rows of the timetable, de-allocating examinations, and allocating examinations. Selection perturbative hyper-heuristics can perform a single point search or a multipoint search. In the case of the former, the hyper-heuristic is comprised of a heuristic selection and move acceptance component [9]. Different techniques are employed to for heuristic selection and move acceptance. These techniques can be as simple as randomly selecting a low-level heuristic and accepting on moves that results in improvement. Metaheuristics can also be employed for heuristic selection and move acceptance, e.g., simulated annealing and tabu search have previously been employed for this purpose. Multipoint search selection perturbation hyper-heuristics employ population-based methods such as evolutionary algorithms and particle swarm optimization to explore

the heuristic space, with the population-based approach, by its nature, performing both the heuristic selection and move acceptance.

As the field developed, the idea of creating new low-level heuristics emerged resulting in a second category of hyper-heuristics, namely, generation hyper-heuristics. Generation hyper-heuristics generate low-level constructive or perturbative heuristics. Genetic programming has primarily been employed by generation hyper-heuristics to create low-level heuristics [8, 9]. An example of a generation hyper-heuristic is that implemented by Burke et al. [7] for the one-dimensional bin-packing problem. Construction low-level heuristics, e.g., first-fit, best-first, and next-fit, are used to choose which bin to allocate an item to. In this study, genetic programming is used to evolve low-level heuristics to decide which bin to allocate an item to. In the study conducted by Sabar et al. [29] grammatical evolution, a variation of genetic programming is used to evolve new low-level perturbative heuristics by combining mechanisms for heuristic selection and move acceptance. Generated low-level heuristics can be reusable or disposable. In the case of reusable heuristics, the generated low-level heuristic created to solve the problem for one instance can be used to solve other problem instances without any regeneration. Disposable low-level heuristics are generated for the specific problem instance and cannot be reused.

Selection and generation hyper-heuristics have generally been used for the automatic configuration of metaheuristic algorithms. In this case, the low-level heuristics represent parameters or operators of the metaheuristic and are essentially perturbative. The hyper-heuristic employs a metaheuristic to explore the space of low-level heuristics. Selection hyper-heuristics are used to determine control flow and for parameter tuning. In this case, the hyper-heuristic selects a component at each point in the application of the algorithm to solve the problem. Furthermore, generation hyper-heuristics are used to create new low-level heuristics; in this context, these represent the components of the metaheuristic.

Hyper-heuristics for Metaheuristic Configuration

Control flow is achieved by producing a combination of low-level heuristics, each is a component of the metaheuristic algorithms, i.e., the hyper-heuristic selects which low-level heuristic to apply at each point in a metaheuristic algorithm. The low-level heuristics are components of the metaheuristic algorithm. Lourenço et al. [21] use grammatical evolution to evolve evolutionary algorithms for the royal road functions. The aim is for the evolutionary algorithm to adapt itself during the evolutionary process. Evaluating the evolutionary algorithm proved to be a computationally intensive task, and hence, a limited number of runs were performed. Grammatical evolution combines the different evolutionary algorithm components, namely, mutation, crossover, and selection components and parameter values for these components. The evolved evolutionary algorithms are applied to a seeded initial population. One instance was used for training, and the evolved

evolutionary algorithms were tested on the remaining four instances. Performance was found to be similar to that of the standard evolutionary algorithm. The evolved EA that performed better than the standard EA did not follow the standard structure and contained two types of crossover. There are a number of early initiatives that can be categorized as hyper-heuristics for inducing evolutionary algorithms, although this is not explicitly stated in the papers. Oltean and Groşan [24] use multigene expression programming to evolve an evolutionary algorithm to induce Griewank's function. Algorithms comprised of initialization, mutation, and crossover components are evolved. The number of crossover, initialization, and mutation components in the best individual was found to increase as the evolution progressed. Linear genetic programming has also been used for purposes of evolutionary algorithm induction [23]. Each algorithm evolved is a generational evolutionary algorithm composed of selection, crossover, and mutation components and is applied to an initial population of randomly generated elements. Evolutionary algorithms are evolved for function optimization, the traveling salesman problem, and quadratic assignment problem. For all three problems, the evolved algorithms are trained on a problem instance and are able to generalize and solve other problem instances. In later work, Dioşan and Oltean [13] use genetic algorithms to evolve evolutionary algorithms for function optimization. Each chromosome is comprised of a combination of selection, mutation, and crossover operations as well as population altering strategies to place the newly created offspring into the population. A different evolutionary algorithm was evolved to induce each of the ten functions. As in previous studies, the crossover operator was the most prevalent in evolved evolutionary algorithms producing the best results.

In some studies, the hyper-heuristic achieves both control flow and parameter tuning. In this case, the hyper-heuristic selects the component of the metaheuristic and the parameter value. Tavares and Pereira [33] employ grammatical evolution to automatically configure ant colonization algorithms for solving the traveling salesman problem. The architecture of the ant colonization algorithm including the components of the algorithm, e.g., method for evaporation, and parameter values are evolved. The evolved architecture was found to be effective when applied to problem instances different from those used for training during evolution. The evolved architectures producing the best results were found to be different from those of the standard ant colonization algorithms. Lourenço et al. [22] use grammatical evolution to design evolutionary algorithms to solve the knapsack problem. The evolved evolutionary algorithms are applied to unseen instances. Each evolved evolutionary algorithm specifies the type of selection, type of crossover, type of mutation, and parameter values. Evolved evolutionary algorithms using binary swap mutation and uniform crossover performed the best.

Generation hyper-heuristics go a step further, and instead of choosing a component of a metaheuristic to decide the control flow, these hyper-heuristics create a new component. In the study conducted by Hong et al. [18] a generative hyper-heuristic, employing genetic programming is used to evolve mutation operators

for evolutionary programming. The terminal set is comprised of random numbers produced by a Gaussian random number generator, and the function set is comprised of arithmetic operators. The approach was used to evolve ten function classes, seven unimodal and three multimodal, and was found to require less processor time than the man hours needed to design new mutation operators. A different mutation operator was evolved for each function class using one instance of the class. Woodward and Swan [37] use local search, namely, hill-climbing, to evolve mutation operators for genetic algorithms. These mutation operators were found to outperform human-created mutation operators. Register machines are used to simulate the behavior of mutation operators. Seven function classes were used to test the effectiveness of the evolved mutation operators. Different mutation operators were created for each function class. Woodward and Swan [36] evaluate a similar approach which uses random search to generate search heuristics for a genetic algorithm. As in the previous study, register machines are used to emulate the selection process. Selection is based on the fitness or rank of a bit string. The evolved selection heuristics were tested for the mimicry problem set and were found to perform better than the human-designed selection heuristics. In the study conducted by Dioşan and Oltean [12], genetic programming induces crossover operators for a genetic algorithm for function optimization. A different crossover operator is evolved for each of the 11 function classes. The performance of the evolved operators was found to be comparative to human-designed crossover operators. Drake et al. [15] also employ grammatical evolution to design the construction heuristic and move operators used by variable neighborhood search to solve the vehicle routing problem. The move operators generated are ruined, and insertion heuristics which are used to perform the shaking process in the variable neighborhood searched. The variable neighborhood search produced results close to the global optimum for all problem instances. Løkketangen and Olsson [20] use the ADATE system to generate the move selection, tabu tenure, and the aspiration criteria in a tabu search for solving the Boolean optimization problem (BOOP). The authors describe ADATE as a generation hyper-heuristic that performs offline learning. ADATE is an automatic programming system that produces functional programs in metalanguage ML. The generated components were found to perform better than manually designed components. One of the generated components producing good results was found to give good moves a longer tabu tenure which is not typical of human-designed tabu searches. This again emphasizes the advantage of automatic generation of metaheuristic components.

Discussion

Multilevel metaheuristics can be categorized as selection or generation perturbative hyper-heuristics. As previously outlined, designing a metaheuristic involves making decisions regarding what parameter values to use, what operators to use, and the control flow of the overall algorithm. From the survey presented above, it is evident

that hyper-heuristics have been fairly effective in making these design decisions. Selection perturbative hyper-heuristics are used to select numerical and discrete parameter values. Hence, the hyper-heuristic explores the space of parameter values. Evolutionary algorithms have chiefly been used for this purpose. The parameter values can either be coevolved while solving the problem at hand or optimized separately. In the latter case, the parameters can be determined offline during the training phase. Selection perturbative hyper-heuristics can also be used to make control flow decisions. This essentially involves selecting different operators at different stages of solving the problem. Hence, the decision of which operator to use can be made as part of the control flow decision. Evolutionary algorithms, including linear genetic programming, genetic algorithms, multigene expression programming, and grammatical evolution have been used for control flow design. The decision regarding which operator to use may not be a matter of selecting an existing operator but creating a new operator. Generation hyper-heuristics can be used to generate new operators. This research has focused to a large extent on selection, mutation, and crossover operators in evolutionary algorithms. Genetic programming has primarily been used to generate new operators. Grammatical evolution, local search, and random search have also been used for this purpose. A hyper-heuristic can be used to make all three design decisions simultaneously. This can be seen in the studies conducted by Lourenço et al. [21, 22] and Tavares and Pereira [33] where grammatical evolution is used to make this decision for the induction of an evolutionary and an ant colonization algorithm simultaneously.

In the case of all three design decisions, the parameter values, operators, and algorithms induced by the hyper-heuristic can be reusable or disposable. In the case of reusability, two phases are performed, a training phase and a testing phase. During the training phase, one or a subset of problem instances are used. The induced parameter values, operators, and algorithms are then used to solve unseen instances. Disposable parameter values, operators, and algorithms are induced for the particular problem instance, and hence, a training phase is not required. Reusability has the advantage of the time required for design being reduced as redesign is not needed for every new instance; however, there may be a limited number of instances for which the generated design is applicable.

One of the challenges associated with using hyper-heuristics for the design of metaheuristics is the processing time needed. Given the advances made in multicore architectures and the availability of multicore architectures on a standard desktop machine, distributed architectures can be designed for the implementation of these hyper-heuristics. Most of the research conducted this far has focused on the configuration of evolutionary algorithms. These ideas can be transferred to the design of other metaheuristics. Evolutionary algorithms have primarily been used for design purposes. These have ranged from genetic algorithms through to grammatical evolution. A comparative study into the performance of the different types of evolutionary algorithms and their contribution to the design process would be interesting.

Conclusion

A metaheuristic algorithm's configuration is the combination of its control flow and its parameter settings. Determining the best possible configuration for a metaheuristic is a difficult task that is commonly done by trial and error and based on the experience of the algorithm designer. For this reason, metaheuristics have been developed that are able to adapt their configuration during the search (adaptive metaheuristics), potentially using a higher-level metaheuristic (multilevel metaheuristics). In this chapter, we have surveyed the literature on this topic. The chapter has also highlighted the effectiveness of (see ▸ Chap. 17, "Hyper-heuristics") as multilevel metaheuristics. This serves as a starting point for researchers wanting to use hyper-heuristics for the automated design of metaheuristics. Hyper-heuristics have been fairly effective for the purpose of design, and in most cases, the generated designs have produced better results than the manually designed metaheuristic, in some experiments producing designs that have not previously been thought of. This overview has also brought to light certain research questions and hence areas for future research. Research thus far has highlighted the potential of hyper-heuristics in the automated configuration of metaheuristics. This has now set the foundation for wider application, including more complex problems.

Cross-References

▸ Hyper-heuristics

References

1. Adenso-Díaz B, Laguna M (2006) Fine-tuning of algorithms using fractional experimental designs and local search. Oper Res 54(1):99–114. https://doi.org/10.1287/opre.1050.0243
2. Battiti R (1996) Reactive search: toward self-tuning heuristics. In: Modern heuristic search methods. Wiley, Chichester, pp 61–83
3. Birattari M (2009) Tuning metaheuristics. Springer, Berlin/Heidelberg. https://doi.org/10.1007/978-3-642-00483-4
4. Bölte A, Thonemann UW (1996) Optimizing simulated annealing schedules with genetic programming. Eur J Oper Res 92(2):402–416. https://doi.org/10.1016/0377-2217(94)00350-5
5. Boutillon E, Roland C, Sevaux M (2008) Probability-driven simulated annealing for optimizing digital FIR filters. In: Studies in computational intelligence. Springer Science & Business Media, pp 77–93. https://doi.org/10.1007/978-3-540-79438-7_4
6. Burke EK, Kendall G, Newall J, Hart E, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. In: Glover F, Kochenberger GA (eds) Handbook of metaheuristics. International series in operations research & management science, vol 57. Springer, pp 457–474. https://doi.org/10.1007/0-306-48056-5_16
7. Burke EK, Hyde MR, Kendall G, Woodward J (2007) Automatic heuristic generation with genetic programming. In: Proceedings of the 9th annual conference on Genetic and evolutionary

computation – GECCO'07. Association for Computing Machinery (ACM). https://doi.org/10.1145/1276958.1277273
8. Burke EK, Hyde MR, Kendall G, Ochoa G, Ozcan E, Woodward JR (2009) Exploring hyper-heuristic methodologies with genetic programming. In: Intelligent systems reference library. Springer Science & Business Media, pp 177–201. https://doi.org/10.1007/978-3-642-01799-5_6.
9. Burke EK, Gendreau M, Hyde MR, Kendall G, Ochoa G, Özcan E, Qu R (2013) Hyper-heuristics: a survey of the state of the art. J Oper Res Soc 64(12):1695–1724. https://doi.org/10.1057/jors.2013.71
10. Dean A, Voss D (eds) (1999) Design and analysis of experiments. Springer, Berlin. https://doi.org/10.1007/b97673
11. Delorme X, Gandibleux X, Rodriguez J (2004) GRASP for set packing problems. Eur J Oper Res 153(3):564–580. https://doi.org/10.1016/s0377-2217(03)00263-7
12. Dioşan L, Oltean M (2006) Evolving crossover operators for function optimization. In: Genetic programming. Springer Science & Business Media, pp 97–108. https://doi.org/10.1007/11729976_9
13. Dioşan L, Oltean M (2009) Evolutionary design of evolutionary algorithms. Genet Program Evolvable Mach 10(3):263–306. https://doi.org/10.1007/s10710-009-9081-6
14. Dobslaw F (2010) A parameter tuning framework for metaheuristics based on design of experiments and artificial neural networks. In: Proceedings of the international conference on computer mathematics and natural computing 2010. WASET
15. Drake JH, Kililis N, Ozcan E (2013) Generation of VNS components with grammatical evolution for vehicle routing. In: Genetic programming. Springer Science & Business Media, pp 25–36. https://doi.org/10.1007/978-3-642-37207-0_3
16. Eiben AE, Smit SK (2011) Evolutionary algorithm parameters and methods to tune them. In: Autonomous search. Springer, Berlin/Heidelberg, pp 15–36. https://doi.org/10.1007/978-3-642-21434-9_2
17. Eiben AE, Hinterding R, Michalewicz Z (1999) Parameter control in evolutionary algorithms. IEEE Trans Evol Comput 3(2):124–141. https://doi.org/10.1109/4235.771166
18. Hong L, Woodward J, Li J, Ozcan E (2013) Automated design of probability distributions as mutation operators for evolutionary programming using genetic programming. In: Proceedings of the 16th European conference on genetic programming – EuroGP 2013, vol 7831, pp 85–96
19. Hooker JN (1995) Testing heuristics: we have it all wrong. J Heuristics 1(1):33–42. https://doi.org/10.1007/bf02430364
20. Løkketangen A, Olsson R (2009) Generating meta-heuristic optimization code using ADATE. J Heuristics 16(6):911–930. https://doi.org/10.1007/s10732-009-9119-1
21. Lourenço N, Pereira FB, Costa E (2012) Evolving evolutionary algorithms. In: Proceedings of the fourteenth international conference on genetic and evolutionary computation conference companion – GECCO 2012. ACM Press. https://doi.org/10.1145/2330784.2330794
22. Lourenço N, Pereira FB, Costa E (2013) The importance of the learning conditions in hyper-heuristics. In: Proceedings of the fifteenth annual conference on genetic and evolutionary computation conference – GECCO 2013. ACM Press. https://doi.org/10.1145/2463372.2463558
23. Oltean M (2005) Evolving evolutionary algorithms using linear genetic programming. Evol Comput 13(3):387–410. https://doi.org/10.1162/1063656054794815
24. Oltean M, Groşan C (2003) Evolving evolutionary algorithms using multi expression programming. In: Advances in artificial life. Springer Science & Business Media, pp 651–658. https://doi.org/10.1007/978-3-540-39432-7_70
25. Prais M, Ribeiro CC (2000) Reactive GRASP: an application to a matrix decomposition problem in TDMA traffic assignment. INFORMS J Comput 12(3):164–176. https://doi.org/10.1287/ijoc.12.3.164.12639
26. Prins C (2004) A simple and effective evolutionary algorithm for the vehicle routing problem. Comput Oper Res 31(12):1985–2002. https://doi.org/10.1016/s0305-0548(03)00158-8

27. Qu R, Burke EK, McCollum B, Merlot LTG, Lee SY (2008) A survey of search methodologies and automated system development for examination timetabling. J Sched 12(1):55–89. https://doi.org/10.1007/s10951-008-0077-5
28. Ross P (2005) Hyper-heuristics. In: Search methodologies. Springer Science & Business Media, pp 529–556. https://doi.org/10.1007/0-387-28356-0_17
29. Sabar NR, Ayob M, Kendall G, Qu R (2013) Grammatical evolution hyper-heuristic for combinatorial optimization problems. IEEE Trans Evol Comput 17(6):840–861. https://doi.org/10.1109/tevc.2013.2281527
30. Sevaux M, Thomin P (2001) Heuristics and metaheuristics for parallel machine scheduling: a computational evaluation. In: Proceedings of 4th metaheuristics international conference, MIC 2001, Porto, pp 411–415
31. Sörensen K, Sevaux M (2006) MA|PM: memetic algorithms with population management. Comput Oper Res 33(5):1214–1225. https://doi.org/10.1016/j.cor.2004.09.011
32. Talbi E-G (2009) Metaheuristics: from design to implementation. Wiley & Sons, Hoboken. ISBN:978-0-470-27858-1
33. Tavares J, Pereira FB (2012) Automatic design of ant algorithms with grammatical evolution. In: Genetic programming. Springer Science & Business Media, pp 206–217. https://doi.org/10.1007/978-3-642-29139-5_18
34. Van Breedam A (1995) Improvement heuristics for the vehicle routing problem based on simulated annealing. Eur J Oper Res 86(3):480–490. https://doi.org/10.1016/0377-2217(94)00064-J
35. Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 1(1):67–82. https://doi.org/10.1109/4235.585893
36. Woodward JR, Swan J (2011) Automatically designing selection heuristics. In: Proceedings of the 13th annual conference companion on genetic and evolutionary computation – GECCO 2011. ACM Press. https://doi.org/10.1145/2001858.2002052
37. Woodward JR, Swan J (2012) The automatic generation of mutation operators for genetic algorithms. In: Proceedings of the fourteenth international conference on genetic and evolutionary computation conference companion – GECCO 2012. ACM Press. https://doi.org/10.1145/2330784.2330796
38. Xu J, Kelly JP (1996) A network flow-based tabu search heuristic for the vehicle routing problem. Transp Sci 30(4):379–393. https://doi.org/10.1287/trsc.30.4.379
39. Xu J, Chiu SY, Glover F (1998) Fine-tuning a tabu search algorithm with statistical tests. Int Trans Oper Res 5(3):233–244. https://doi.org/10.1111/j.1475-3995.1998.tb00117.x