Genet Program Evolvable Mach
DOI 10.1007/s10710-013-9186-9

Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms

Gisele L. Pappa · Gabriela Ochoa · Matthew R. Hyde · Alex A. Freitas · John Woodward · Jerry Swan

Received: 10 December 2012 / Revised: 28 March 2013

Abstract The fields of machine meta-learning and hyper-heuristic optimisation have developed mostly independently of each other, although evolutionary algorithms (particularly genetic programming) have recently played an important role in the development of both fields. Recent work in both fields shares a common goal, that of automating as much of the algorithm design process as possible. In this paper we first provide a historical perspective on automated algorithm design, and then we discuss similarities and differences between meta-learning in the field of supervised machine learning (classification) and hyper-heuristics in the field of optimisation. This discussion focuses on the dimensions of the problem space, the algorithm space and the performance measure, as well as clarifying important issues related to different levels of automation and generality in both fields. We also discuss important research directions, challenges and foundational issues in meta-learning and hyper-heuristic research. It is important to emphasize that this paper is not a survey, as several surveys on the areas of meta-learning and hyper-heuristics (separately) have been previously published. The main contribution of the paper is to contrast meta-learning and hyper-heuristics methods and concepts, in order to

G. L. Pappa (✉)
Computer Science Department, Universidade Federal de Minas Gerais,
Av. Antonio Carlos, 6627, Pampulha, Belo Horizonte 31270-010, Brazil
e-mail: glpappa@dcc.ufmg.br

G. Ochoa · J. Woodward · J. Swan
Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA,
Scotland, UK

M. R. Hyde
School of Environmental Sciences, University of East Anglia, Norwich Research Park,
Norwich NR4 7TJ, UK

A. A. Freitas
School of Computing, University of Kent, Canterbury, Kent CT2 7NF, UK

promote awareness and cross-fertilisation of ideas across the (by and large, non-overlapping) different communities of meta-learning and hyper-heuristic researchers. We hope that this cross-fertilisation of ideas can inspire interesting new research in both fields and in the new emerging research area which consists of integrating those fields.

Keywords Hyper-heuristics · Meta-learning · Genetic programming · Automated algorithm design

1 Introduction

Despite the success of heuristic optimisation and machine learning algorithms in solving real-world computational problems, their application to newly encountered problems, or even new instances of known problems, remains difficult; not only for practitioners or scientists and engineers in other areas, but also for experienced researchers in the field. The difficulties arise mainly from the significant range of algorithm design choices involved, and the lack of guidance as to how to proceed when choosing or combining them. This motivates the renewed and growing research interest in techniques for *automating the design of algorithms* in optimisation, machine learning and other areas of computer science, in order to remove or reduce the role of the human expert in the design process.[1]

Automating the design of algorithms is not a new idea, and what has changed is the *algorithmic level* at which the automation is performed. Consider the area of evolutionary computation, for example. Initially, researchers concentrated on optimizing algorithm *parameters* automatically, which gives rise to adaptive and self-adaptive parameter control methods [7]. With time, the definition of parameters was broadened to include not only continuous variables, such as crossover and mutation rates, but also include 'categorical' parameters, i.e. evolutionary algorithms *components*, such as the selection mechanism and crossover and mutation operators [65]. After that, evolutionary algorithms were first used in the meta-level, i.e. to generated a complete evolutionary algorithm, as showed in the works of Oltean [83].

In the area of machine learning, automated algorithm design appeared as a natural extension of the first works focusing on automated algorithm selection. As discussed in [106], the algorithm selection problem was formally defined by John Rice in 1976 [94], and the big question posed was: *Which algorithm is likely to perform best for my problem?* The area of *meta-learning* [11] took the challenge and started to take shape in the late eighties, but was formally introduced in 1992 with the MLT project [63]. The MLT project created a specialist system (named Consultant-2) to help in selecting or recommending the best algorithm for a given problem. This first project was followed by two others, namely Statlog [73] and

[1] Note that when we talk about algorithms, we mean any sequence of steps that is followed to solve a particular problem, regardless of whether these steps describe a heuristic, a neural network or a genetic algorithm.

METAL [12]. In all three projects, the main difference between meta-learning and the traditional base-learning approach was in their level of adaptation. While learning at the base level focused on accumulating experience on a specific learning task, learning at the meta level accumulated experience in the performance of multiple applications of a learning system [11]. Later, meta-learning developed other research branches, such as model combination and, more recently, *automated algorithm generation* [87].

Within combinatorial optimisation, the term *hyper-heuristics* was first used in 2000 [30] to describe *heuristics to choose heuristics*. In this case, a hyper-heuristic was defined as a high-level approach that, given a particular problem instance and a number of atomic heuristics, selects and applies an appropriate heuristic at each decision point [14, 96]. This definition of hyper-heuristics was also expanded later to refer to an automated methodology for *selecting or generating* heuristics to solve hard computational search problems [16].

Note that the original definitions of both meta-learning and hyper-heuristics were expanded to move from heuristic/algorithm selection to heuristic/algorithm generation. In both areas, the turning point from selecting to generating heuristics/algorithms had the same cause: the expressive power of genetic programming as a tool for algorithm design (see [15, 87] for an overview). In this new context, hyper-heuristics aimed to generate new heuristics from a set of known heuristic components given to a framework. Similarly, in meta-learning, the idea was to generate new learning algorithms by combining algorithm primitives (such as loops and conditionals) with components of well-established learning methods to generate new learning algorithms. In both cases, the distinguishing feature of search methods in the meta/hyper level is that they operate on a search space of algorithm components rather than on a search space of solutions of the underlying problem. Therefore, they search for a good *method* to solve the problem rather than for a good solution [27].

One issue when migrating from heuristic/algorithm selection to generation was the need for generalization. In typical applications of heuristic optimisation methods, the quality of a candidate solution returned by the fitness function is evaluated with respect to a single instance of the target problem. In such applications, the generality of the method applied does not matter (for a further discussion of this point, the reader is referred to [87] pp. 97–100)). By contrast, in machine learning tasks such as classification, the algorithm learns a classification model from the training set, which is later applied to classify instances in the test set. The goal of a classification algorithm is to discover a classification model with high generalisation ability, i.e. a model that has a high predictive accuracy on the test set, containing data instances not observed during the training of the algorithm. Hence, the work previously done in this area can be of great help for the hyper-heuristic community.

Given the historical resemblance between the evolution of automated design in both the learning and optimisation communities, the main objective of this paper is to bring together the supervised machine learning (classification) and heuristic optimisation communities to contrast their work, which both seek to: (i) automate the process of designing or selecting computational problem solving methodologies; and (ii) raise the level of generality of these methodologies.

It is important to emphasize that this paper does not intend to be a survey, as several surveys in the areas of meta-learning and hyper-heuristics (separately) have being previously published [16, 85, 115]. The main contribution of the paper is rather to contrast meta-learning and hyper-heuristics methods and concepts, in order to promote awareness and cross-fertilisation of ideas across the (by and large, non-overlapping) different research communities of meta-learning and hyper-heuristics. We hope that this cross-fertilisation can inspire interesting new research in both fields and in the new emerging research area which consists of integrating those fields.

The remainder of this paper is organized as follows. Section 2 brings a historical perspective of the idea of automatic algorithm design. Section 3 introduces the areas of meta-learning and hyper-heuristics, and contrasts them according to three points: (i) the problem space; (ii) the algorithm space, and (iii) the performance measure. Section 4 presents various examples of automatic algorithm design in different levels of generalization, emphasizing the differences between algorithm selection and generation. Finally, Sect. 5 discusses the differences between current machine learning and optimisation approaches for automatic algorithm design, and presents directions for future research mainly in foundation studies, generalization of the algorithms generated and the evaluation process.

2 A historical perspective on automated algorithm design

As previously discussed, the idea of automatic algorithm design is not new: it has been investigated by different areas for the past 50 years, from different perspectives. The desire to automatically create computer programs for *machine learning* tasks, for example, dates back to the pioneering work of Samuel in the 1950s, when the term machine learning was first coined meaning “computers programming themselves” [75]. Given its original difficulty, this definition was revised with time, being redefined as the system’s capability of learning from experience. Later, based on this same idea, the area of meta-learning was the first to deal with selecting/building algorithms tailored to the problem at hand, as detailed in Sect. 3.1.

In evolutionary computation, this problem was studied in different algorithms and at various abstraction levels. For instance, the popularity of genetic programming in the early 1990s for the automatic evolution of computer programs [64] was the first step towards current efforts to evolve programs using knowledge from the user (such as those based on grammars [86]) or evolving code in a particular language, such as C or Java [53, 77, 84]. In parallel, the first studies on adaptive and self-adaptive evolutionary algorithms appeared, in which the algorithms were dynamically adapted to the problem being solved.

Initially, Angeline [6] grouped the latter methods according to the level of adaptation they employed. Three different levels were defined: population-level (global parameters), individual-level (parameters to particular individuals) and component-level (different components of a single individual). Different strategies were proposed according to the desired adaptation level, and the same is true for automatic algorithm design. Later, Back [7] summarized the strategies of parameter control in three groups: (i) dynamic, where the parameters were modified according to a deterministic strategy defined by the user; (ii) adaptive, where a feedback mechanism monitors evolution and changes parameters according to their impact on the fitness function; and (iii) self-adaptive, where the parameter values are inserted into the individual representation or in a new cooperative population, and suffer the same types of variations as the solutions themselves.

Summarizing the previous works in this area, Eiben et al. [40] proposed a taxonomy for parameter setting methods, which were divided into two main types: parameter tuning and parameter control. The main difference between the two is that parameters are tuned before the evolution starts, and controlled during evolution. While Eiben et al. focused on reviewing parameter control, Kramer [65] extended the taxonomy to include different types of parameter tuning, including meta-evolution. Meta-evolutionary algorithms (such as meta-genetic algorithms and meta-genetic programming) use evolutionary algorithms in a nested fashion, which occurs on two levels. The outer level evolutionary algorithm is used to tune the parameters of the inner level evolutionary algorithm (i.e. the one solving the problem).

The initial works in meta-evolutionary algorithms also focused on optimizing continuous parameters, although in 1986 Grefenstette took six parameters into account, including population size, crossover and mutation rates and the type of selection to be followed by the algorithm (which was a discrete parameter) [58]. However, works in this area were criticized for bringing a problem similar to the one being solved: optimising the parameters of the outer loop evolutionary algorithms, which made the problems of parameter optimisation recursive. Other works chose to use two co-evolving populations to solve this same type of problem, where the main population used crossover and mutation operators evolved by the second population during evolution [38]. Methods based on meta-approaches also evolved with time, and the parameters were replaced by high levels of components that, in the ultimate case, can generate complete evolutionary algorithms [83]. In [83], the authors proposed to evolve an evolutionary algorithm for families of problems using a steady-state linear genetic programming (LGP), as detailed in Sect. 4.4.1. Another type of system worthy of note is the autoconstructive evolution system proposed by [107], named Pushpop, where the methods for reproduction and diversification are encoded in the individual programs themselves, and are subject to variation and evolution.

In the area of artificial life, in contrast with the efforts for automatic program generation, computer simulations such as Tierra and Avida appeared in the early 90s to create digital creatures or organisms. In Tierra, computer programs (digital organisms) competed for CPU time and memory, and could self-replicate, mutate and recombine. The main purpose of the system was to understand the process of evolution, rather than solve computational problems. Avida, in turn, was an extension of Tierra that guided evolution to solve simple problems [82]. Avida has been largely used to simulate biological and chemical systems [1], but was also extended to other interesting problems that need robust and adaptive solutions. Georgiou and Teahan [56], for example, developed Avida-MDE, which generates behavior models for software (represented by a set of finite state machines) that capture autonomic system behavior that is potentially resilient to a variety of environmental conditions.

3 Contrasting meta-learning and hyper-heuristic optimisation methods

One of the objectives of this paper is to contrast automated methods for selecting/generating new heuristics/algorithms for a given problem. This section starts by summarizing how the meta-learning and hyper-heuristics fields developed to automatically select or build algorithms/heuristics, and then contrasts the two approaches.

3.1 Meta-learning

Meta-learners were first developed to help users choose which algorithm to apply to new application domains. The area does that by benefitting from previous runs of each algorithm on different datasets. Hence, while a traditional (or base) learner accumulates experience over a specific problem, meta-learners accumulate experience from the performance of the learner in different applications [11].

The most common types of meta-learning are algorithm selection/recommendation and model combination. Algorithm selection/recommendation is based on the use of meta-features, which can be expressed using: (i) dataset statistics, such as number of features, class entropy, attributes and classes correlations, etc; (ii) properties of an induced hypothesis (e.g. for a rule induction model, features such as the number of rules, number of conditions per rule using numerical and categorial attributes, etc); and (iii) performance information of learning systems with different learning biases and processes. Hence, algorithm selection/recommendation helps the user to choose the learner by generating a ranking of algorithms or indicating a single algorithm according to their predictive performance. The ranking/selection is created using the meta-data aforementioned, which combines dataset characteristics with algorithms performance.

Model combination, in turn, combines the results of different inductive models, based on the idea that sets of learners may present better generalization than the learners by themselves. The models are generated based on two main approaches. In the first case, different dataset samples are used to train the same learner. In the second case, different learners are used to learn from the same dataset. In both approaches, the final results are given by the combination of the outputs of the single learners. Two well-known algorithms that follow the first approach are bagging and boosting.

Bagging extracts n samples from the dataset with replacement, and learns a model from each of them [13]. Given a new instance to be classified, the method performs a majority voting, assigning the class given by the majority of the n learners to the example. Boosting also works with a single learner, but is based on the assumption that combining weak learners (i.e. those with predictive performance just above random) is easier than finding a single learner with high predictive accuracy [102]. Based on this idea, boosting runs a learner repeatedly over different data distributions, and combines their final results. However, boosting algorithms change the data distribution according to the errors made by the first learners. Initially, each example is assigned a constant weight *w*. On each iteration, the weights of misclassified instances are increased, so that the next model built gives more importance to them.

Stacking [117] is an example of an algorithm that works with different learners instead of different data samples. It runs a set of base-learners on the dataset being considered, and generates a new dataset *M* replacing (or appending) the instance features with the learner results. A meta-learner is then used to associate predictions of the base-learners with the real class of the examples. When new examples arrive, they are first run in the set of base-learners, and the results given to the meta-learner. Examples of other algorithms combining different datasets and multiple base-learners are cascading and delegating.

In addition to the two approaches of meta-learning just described, a third type, called algorithm generation, is discussed in this paper. Instead of selecting the best learner, this approach builds a learner based on its low-level components, such as search mechanism, evaluation function, pruning method, etc. In this case, the method working at the meta-level is usually an evolutionary algorithm. In particular, genetic programming, which is intrinsically a machine learning approach [8], is the most explored method.

The first approaches based on algorithm generation focused on evolving neural networks [119]. There are still many efforts in this direction, as described in Sect. 4.3. In the case of evolutionary artificial neural networks, researchers have gone one step further, and are using evolutionary algorithms to create ensembles of these networks [28], which is categorized as a combination method in meta-learning. After neural networks, approaches for building rule induction algorithms [87] and decision trees [9] were also proposed. These approaches resemble work in meta-evolutionary algorithms, and in this case the outer loop algorithm is an evolutionary approach and the inner loop is essentially a learning algorithm, which can be, again, an evolutionary one.

To summarize this section, Fig. 1 presents a classification of the most common approaches for meta-learning. This paper covers the approaches for selection and

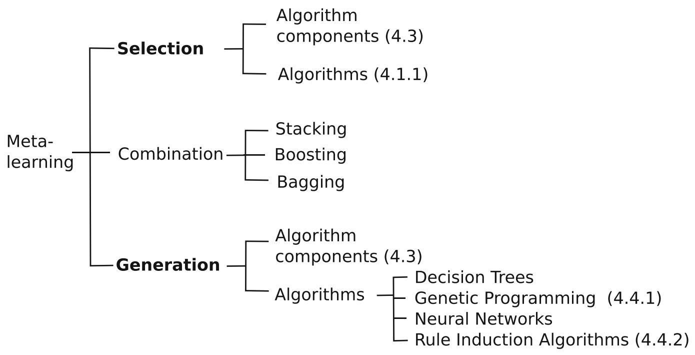


Fig. 1 Classification of meta-learning methods

generation of heuristics, although their combination is out of the scope of the paper. For more details on this, the reader is referred to [95]. The numbers adjacent to some categories indicate the section of this paper in which an example of that approach can be found. Note that in both selection and generation methods, one can work at the level of algorithmic components or algorithms. Although the idea of meta-learning refers to the choice of algorithms (see Sect. 4.1), methods for component selection have also been proposed (see Sect. 4.3 for an example of decision-tree split functions). Additionally, the generation of algorithms and algorithm components is a more recent development, as shown in the examples in Sect. 4.4.

3.2 Hyper-heuristics

The term hyper-heuristic is relatively new—it first appeared in a peer-reviewed conference paper in 2000 [30], to describe *heuristics to choose heuristics* in the context of combinatorial optimisation, and the first journal paper to use the term in this sense was [22]. However, the idea of automating the design of heuristic methods is not new; it can be traced back to the 1960s, and can be found across Operational Research, Computer Science and Artificial Intelligence. Fisher and Thompson [45], showed that combining scheduling rules (also known as priority or dispatching rules) in production scheduling was superior to using any of these rules separately. This pioneering work should be credited with laying the foundations of the current body of research into hyper-heuristic methods. Another body of work that inspired the concept of hyper-heuristics came from the Artificial Intelligence community. In particular, from work on automated planning systems and the problem of learning control knowledge [57]. The early approaches to automatically set the parameters of evolutionary algorithms can also be considered as antecedents of hyper-heuristics. The notion of 'self-adaptation', first introduced within evolution strategies for varying the mutation parameters [93, 103], is an example of an algorithm that is able to tune itself to a given problem whilst solving it. Another idea is to use two evolutionary algorithms: one for problem solving and another one (a so-called meta-evolutionary algorithm) to tune the first one [59]. Finally, a pioneering approach to the automated generation of heuristics can be found in the domain of constraint satisfaction [74]; where a system for generating reusable heuristics is presented.

Hyper-heuristics are related to metaheuristics [55, 110] but there is a key distinction between them. Hyper-heuristics are search methods that operate on a search space of heuristics (or algorithms or their components), whereas most implementations of metaheuristics search on a space of solutions to a given problem. However, metaheuristics are often used as the search methodology in a hyper-heuristic approach (i.e. a metaheuristics is used to search a space of heuristics). Other approaches, not considered as metaheuristics, can and have been used as the high-level strategy in hyper-heuristics such as reinforcement learning [31, 37, 81, 88], case-based reasoning [24] and learning classifier systems [98, 112].

In a recent book chapter [16], the authors extended the definition of hyper-heuristics and provided a unified classification which captures the work that is being undertaken in this field. A hyper-heuristic is defined as *a search method or learning* *mechanism for selecting or generating heuristics to solve computational search problems*. The classification of approaches considers two dimensions: (i) the nature of the heuristics' search space, and (ii) the different sources of feedback information from the search space. According to the nature of the search space, we have;

- *Heuristic selection*: methodologies for choosing or selecting existing heuristics
- *Heuristic generation*: methodologies for generating new heuristics from the components of existing heuristics.

A second level in this first dimension (the nature of the search space) corresponds to the distinction between constructive and perturbative (or improvement) heuristic search paradigms. Constructive hyper-heuristic approaches build a solution incrementally: starting with an empty solution, the goal is to intelligently select and use constructive heuristics to gradually build a complete solution. In selection hyper-heuristics, the framework is provided with a set of pre-existing (generally problem-specific) constructive heuristics, and the challenge is to select the heuristic that is somehow the most suitable for the current problem state. This type of approach has been successfully applied to hard combinatorial optimisation problems such as cutting and packing [98, 112], educational timetabling [23, 24, 97] and production scheduling [25, 43]. In the case of generation hyper-heuristics, the idea is to combine sub-components of previously existing constructive heuristics to produce new constructive heuristic. Examples can be found in bin packing [19, 20] and production scheduling [35, 111]. This classification is illustrated in Fig. 2.

In contrast, improvement hyper-heuristic methods start with a complete solution, generated either randomly or using simple constructive heuristics, and thereafter try to iteratively improve the current solution. In selection hyper-heuristics, the framework is provided with a set of neighborhood structures and/or simple local searchers, and the goal is to iteratively select and apply them to improve the current complete solution. This type of approach has been applied to problems such as personnel scheduling [22, 31], timetabling [22], packing [37] and vehicle routing [51, 88]. Improvement heuristics can also be automatically generated. Examples can be found for both producing a complete improvement search method or some of its algorithmic components; in domains such as boolean satisfiability, [49], bin packing [18], and traveling salesman problem [83].

The second dimension in the classification considers the source of the feedback during learning: we can distinguish between *online* and *offline* learning. In online learning hyper-heuristics, the learning takes place while the algorithm is solving an instance of a problem. Therefore, task-dependent properties can be used by the

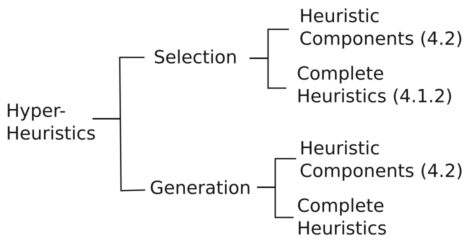

Hyper-Heuristics
Selection
Heuristic Components (4.2)
Complete Heuristics (4.1.2)
Generation
Heuristic Components (4.2)
Complete Heuristics

Fig. 2 Classification of hyper-heuristics methods

high-level strategy to determine the appropriate low-level heuristic to apply. Examples of online learning approaches within hyper-heuristics are: the use of reinforcement learning for heuristic selection [31, 37, 81, 88] and generally, the use of metaheuristics as high-level search strategies across a search space of heuristics [22, 23, 25, 43, 97]. In offline learning hyper-heuristics, the idea is to gather knowledge in the form of rules or programs, from a set of training instances, that will hopefully generalize to the process of solving unseen instances. Examples of offline learning approaches within hyper-heuristics are: learning classifier systems [98], case-based reasoning [24] and genetic programming [18–20, 35, 49, 111].

These categories reflect current research trends. However, there are methodologies that can cut across categories. For example, we can see hybrid methodologies that combine constructive with perturbation heuristics [51], or heuristic selection with heuristic generation [42, 60, 66, 71].

3.3 A framework to contrast meta-learning and hyper-heuristics

A first attempt to contrast meta-learning and hyper-heuristics was done by [32], which focused on the main applications of algorithm selection for real-world problems. The idea of this paper, however, is to review methods for selection and generation of algorithms in both areas and contrast them. In order to make this comparison easier, we will borrow Rice's framework for algorithm selection and (without loss of generality) extend it to a framework that encompasses algorithm generation. According to Rice [94], there are three important dimensions to be taken into account when tackling algorithm selection problems: (i) the problem space; (ii) the algorithm space, and (iii) the performance measure. These three dimensions are essential for both hyper-heuristics and meta-learning.

The *problem space* defines the set of all possible instances of the problem. In both hyper-heuristics and meta-learning, the problem space can be seen from three different perspectives. In the first case, one might want to create a general heuristic/algorithm, which has a robust, good performance across a very wide range of instances of the target problem. In the second case, one can generate heuristics/algorithms for datasets/problem instances with similar characteristics (or families of problems). In the third case, the heuristic/algorithm can be developed to solve a specific instance of a problem, without the need to generalize to different problem instances.

The *algorithm space* can be explored at different levels, considering either all available algorithms/heuristics to solve the problem or components of the former at different abstraction levels. The variety of each of these algorithms/components is defined by the user, and ultimately determines the size of the search space. Finally, the *performance measure* defines which criteria will be used to evaluate an algorithm in a specific problem. Different algorithms might be appropriate to different problems according to this measure.

3.3.1 Problem space

This section discusses the three aforementioned levels at which heuristics or algorithms can be generated: creating a general heuristic/algorithm, generating heuristics/algorithms for datasets/problem instances with similar characteristics, and solving a specific instance of a problem.

Research in meta-learning has already shown that one can customize rule induction algorithms using both selection and generation approaches for a *single dataset* [86], and obtaining success in this case is more likely than when competing with more general algorithms, fine-tuned to generalize well in the great majority of problems. In the optimisation field, genetic programming hyper-heuristics have also been shown to operate on *single problem domains*. Examples of single domains where generative hyper-heuristics have been applied are 2D bin packing [19], and job shop scheduling [52]. Note that here we refer to the term hyper-heuristics in the context of heuristic generation rather than heuristic selection. Heuristic selection has been shown to be able to operate over multiple problem domains, but the domains must have pre-existing human-generated heuristics. Therefore, the operation of the selection hyper-heuristic over multiple problem domains is not completely automated.

Regarding the second case (building algorithms/heuristics for problem instances with similar characteristics), it is demonstrably possible in the field of optimisation to automatically design heuristics for certain families of problem instances having shared characteristics. In particular, [17] study the trade-off between generalization and performance which is associated with evolving generic heuristics (which work reasonably well across a very broad class of problem instances) or evolving specialised heuristics (which work very well in a sub-class of problem instances, but not well in a very different sub-class of problem instances). Their results are consistent with the 'no free lunch' theorem [118], which suggests that a heuristic that performs well over a large set of problem instances can be beaten on any particular subset by a specialised heuristic. A computer system which can automatically design specialised heuristics can be much more successful over a targeted subset of problem instances than a fixed, hand-crafted, general heuristic.

For meta-learning, this second approach was tried with algorithm selection, but not generation. One of the main problems here is how to characterise datasets as similar. Dataset characterisation is still an open research question [11], but simple attributes such as number of attributes, classes by distribution of attribute values, number of classes, among others, have already been tried for algorithm selection.

The generation of algorithms which can generalise to any other instances (the first of the three problem spaces mentioned above) has already been tackled in meta-learning, but is probably the case where one can expect fewer advantages of the built method regarding the other algorithms in the literature. This is because it is easier to generate a better algorithm for a target domain than one that performs well in a wide range of datasets. In supervised machine learning (in particular, for the classification task addressed in this paper), even the simplest models generated from data must be generalisable, and methods to prevent data overfitting have been studied for quite some time [76]. The automatic design of algorithms gives a new context to the problem, considering now the algorithms should generalise well to many new datasets. So far, this problem was tackled by training an algorithm with many datasets, and then testing its performance in a different set of non-overlapping datasets (from different application domains than the datasets used to train the system). According to previous results, this approach is capable of generating algorithms that generalise well, and have comparative accuracy with state of the art algorithms [87]. However, a motivation for automatic algorithm design is to generate an algorithm which is better than others already proposed in the literature, given the specificity and computational cost of the task.

In this direction, and taking into account the Law of Conservation of Generalisation Performance [92, 101], which states that two different algorithms have on average exactly the same performance over all possible classification problems, maybe the best approach is to focus first on automatically designing algorithms for ‘families’ of datasets or single datasets. After this problem is well-understood, we may move on to more complicated and general domains. The principle here is the same as discussed above for the case of hyper-heuristics in optimisation, namely the trade-off between generalization and performance. This trade-off suggests that, all other things being equal, a classification algorithm automatically designed for a specific class of classification problems is expected to be more effective in problems of that class than another classification algorithm automatically designed to be robust across a much larger class of problems.

In summary, a problem domain in classification is a set of data points having different features, such as cancer patient medical data, large scale bioinformatics data, or financial data. It has been shown that algorithms can be automatically generated for different classification domains, but it remains to be seen if the same can be done for different optimisation domains. The challenge is different in optimisation, since optimisation problems are represented with different data structures, rather than a different set of feature-value pairs. The higher levels of generality recently reached by machine learning research may not be possible in optimisation systems, given the different problem formulations and (most importantly) modeling required in optimisation problems. This is an important point for future research.

Still related to the point of generalisation is the idea of problem class hierarchies, which can be defined based on certain characteristics of the problems, or how they have been generated. Algorithms do not have to be evolved for **any** possible future instance, and in real world problems we do not expect them to. For example, an organisation’s algorithm may be trained on real instances from the past year. Such an algorithm would not need to operate well on instances from another organisation, and has been implicitly trained on the ‘class’ of problems that the organisation expects to see in the future. It has been shown in the one-dimensional bin packing domain that optimisation heuristics can be specialised to progressively more narrow classes of instances in a hierarchy [17]. Further studies on how problem class hierarchies relate to the generalisation of hyper-heuristics are also an interesting direction of future work.

3.3.2 Algorithm space

As already discussed in Sects. 3.1 and 3.2, both hyper-heuristics and meta-learning can explore the algorithm space using a selection or a generation approach. In the case of meta-learning, there is also a third scenario, in which one can combine the results of different machine learning approaches. Both selection and generation approaches are well defined, as illustrated in Figs. 1 and 2. Regardless of the approach followed, it can work at the component or algorithm level.

One of the aspects we want to emphasize in this paper is the use of GP to explore this algorithm space, possibly due mainly to the ease with which heuristics/algorithms can be represented. GP is able to evolve heuristics/algorithms by virtue of its expressive power (defined by a function and terminal set), rather than because of its search operators (e.g. crossover/mutation). Hence, the search in the space of heuristics or algorithms can be performed by any other type of search method, given a suitable equivalent of GP's function set with sufficient expressive power. Some interesting alternatives to GP to construct programs include the use of a variation of Estimation of Distribution Algorithms called Estimation of Distribution Programming [104] and a variation of Ant Colony Optimisation called Ant Programming [99, 100]. Although ant algorithms have been used as a hyper-heuristic to *select* heuristics [21, 29, 33, 62], to the best of our knowledge neither technique has been used as a hyper-heuristic to *generate* heuristics, and this could be an interesting research direction.

The best method of searching the space of heuristics or algorithms is still an open research question, with very little research done on this topic so far. One such example is a comparison between grammar-based GP and grammar-based greedy hill-climbing hyper-heuristics to generate a full data mining (rule induction) algorithm, using the same grammar (defining the same data mining algorithm space) in both methods. Pappa and Freitas [86, 87] reported that GP was more effective than hill-climbing. In contrast, on a different set of problem domains, [61] suggests that a grammar based local search methodology can outperform GP in the task of automatically generating heuristics. In any case, given the large diversity of search methods available in the literature, we would argue that the current popularity of population-based methods does not necessarily mean that they are the best choice to automatically design algorithms, compared to single point stochastic local search methods.

At present, the majority of the computational methods produced by GP are not algorithms in the sense of the term considered in this paper. They do not have loops nor nested If-Then-Else statements, and they do not consist of multiple heuristic procedures. In most GP applications, the entity being evolved is better described as a mathematical expression, consisting of mathematical operators applied to variables and randomly generated constants. This is highly applicable in many cases, especially in optimisation, where mathematical expressions can be used as heuristics to choose between different options. However, GP does have the potential to evolve full algorithms, as long as the function set and the terminal set are carefully defined to allow the representation of loops, nested If-Then-Else statements, instructions for integrating the results of multiple heuristic procedures, etc. In the following section on case studies, we discuss some examples of GP systems that evolve full algorithms for optimisation and classification problems.

One of the goals of this paper is precisely to draw the attention of the GP research community to this more challenging and arguably more interesting usage of GP, closer to the original spirit of GP as an automatic programming tool, although our interest is mainly on automatic algorithm design rather than on the details of any particular programming language.

3.3.3 Evaluation measure

The evaluation measure is a dimension in which, at first glance, there is a large difference between meta-learning for classification and hyper-heuristics for optimisation. The reason for this apparently great difference is the differing nature of optimisation and classification problems, as follows. First, note that there are many different types of optimisation problems (e.g. Travelling Salesman Problem and Bin Packing Problem), each with its specific evaluation measure (e.g. tour length or some function of the bins required to pack all items, respectively). Note also that, in a conventional optimisation framework, a candidate solution is evaluated with respect to a single instance of the target problem, e.g. the quality of a candidate solution for the Travelling Salesman Problem typically refers to the length of a tour for one predefined set of cities and corresponding pairwise distances. That problem instance is just one instance out of the infinite number of possible problem instances that can be obtained by varying the number of cities and the real values of the distances between cities.

In contrast, in a conventional classification framework, a candidate solution must be evaluated with respected to its generalization ability across many different data instances of the same application domain (e.g. different customers of the same credit-scoring domain, or different patients of the same medical-diagnosis domain). Furthermore, the training and testing instances must be drawn independently from the same distribution. The need for this generalization is not present in a conventional optimisation framework.

However, the interesting point is that the use of hyper-heuristics in optimisation blurs the aforementioned distinction between evaluation measures for classification and optimisation. When using hyper-heuristics to select or construct a heuristic for optimisation, a candidate heuristic is typically evaluated in terms of how well it performs on a set of problem instances, not just a single problem instance. It is also possible and desirable to use the notion of training and testing sets in this context, i.e. once a hyper-heuristic has selected or constructed a heuristic by using a training set of problem instances, it is interesting to measure the performance of that heuristic on a different set of testing problem instances, unused during the training of the hyper-heuristic.

In the case of meta-learning for classification, the same basic principle of generalization still applies, but now the generalization issue is further extended to an even higher level of abstraction. More precisely, when a meta-learning system selects or constructs a classification algorithm, we can measure generalization performance at two levels. At the base level, we measure the generalization ability (predictive accuracy) of the classification model built by the (automatically selected or constructed) classification algorithm in a set of testing data instances from a specific dataset (from a specific application domain, like medicine or finance). In addition, at the meta-level, we can measure the generalization ability of the meta-learning system itself across different types of datasets or application domains. In other words, we can measure a kind of average predictive accuracy associated with application of the classification algorithm produced by the meta-learning system to different types of datasets or application domains.

4 Automating the design of algorithms: different levels, different approaches

This section focuses on automatic algorithm design at different levels. We start with methods that follow meta-learning approaches in classification and optimisation to solve the algorithm selection problem. We then consider the selection and generation of algorithm components, contrasting the similarities and differences between these approaches in the same problem domain. We next give two examples of complete heuristic/algorithm design, where sets of components were combined, both with and without algorithm primitives. Finally, we discuss and contrast the approaches followed in meta-learning and hyper-heuristics.

4.1 Algorithm selection: what researchers did before automatic algorithm generation

As previously explained, the areas of meta-learning and hyper-heuristics, in their early days, focused on algorithm selection. Meta-learning ideas have traditionally been applied to learning algorithms to solve classification problems, where the goal is to relate performance of algorithms to characteristics or measures of classification datasets. Smith-Miles [106] presented a framework for the generalisation of algorithm selection and meta-learning ideas to algorithms focused on other tasks such as sorting, forecasting, constraint satisfaction and optimisation. However, classification is still the most studied task in meta-learning. Hence, the next sections discuss examples of algorithm selection in classification, and then show how an optimisation task is solved using the same type of approach.

4.1.1 Algorithm selection for classification problems

For quite some time, the machine learning community has been interested in meta-learning for selecting the best learning algorithm to solve a classification problem using performance measures related to classification accuracy. Different studies over the years have increased the sophistication of both the features used to characterise the datasets, and the learning algorithms to learn the mapping from the features to the algorithms. We discuss here one of the earliest attempts to characterise a classification problem and examine its impact on algorithm behavior, using features related to the size and concentration of the classes [2]. The approach used rule based learning algorithms to develop rules like:

```
If the given dataset has characteristics C1, C2, ..., Cn
then use algorithm A1
else use algorithm A2
```

Fig. 3 Example of the rules produced by the algorithm selection approach in [2]

```
IF (# training instances < 737) AND
    (# prototypes per class > 5.5) AND
    (# relevants > 8.5) AND
    (# irrelevants < 5.5)
THEN IB1 will be better than CN2
```

In this particular problem, the three algorithms used to learn the association between the attributes describing the classification problems and the learning algorithms were the following: (i) IB1: nearest neighbour classifier, (ii) CN2: set covering rule learner, and (iii) C4: decision tree learner.

The classification problems being addressed considered datasets from the Frey and Slate letter recognition problem [48]. The features which described the classification problems included the number of instances, number of classes, number of prototypes per class, number of relevant and irrelevant attributes, and the distribution range of the instances and prototypes. Note that the number of relevant and irrelevant attributes is not usually known a priori, but in this case the datasets were artificially generated to study the behavior of the learning algorithms. Figure 3 illustrates an example of the kind of rules produced by this algorithm selection approach.

The study found that despite some constraints, the rules derived from the proposed method yielded valuable characterisations describing when to prefer using specific learning algorithms over others.

4.1.2 Algorithm selection for optimisation problems

There has been surprisingly few attempts to generalize the relevant meta-learning ideas to optimisation, although several approaches can be found in the related area of constraint satisfaction [70]. We discuss here the approach proposed in [105] to use meta-learning ideas for modeling the relationship between instance characteristics and algorithm performance for the Quadratic Assignment Problem (QAP).

The study considered a set of 28 problem instances taken from [109], and three metaheuristic algorithms were considered for selection: (i) robust tabu search, (ii) iterated local search, and (iii) min-max ant system. The performance of each algorithm for each dataset was measured by the percentage difference between the objective function value obtained by the algorithm and the known optimal solution.

Each problem instance was characterized using 9 meta-features, which included four measures of problem size (dimensionality, dominance of distance, flow matrices, and sparsity of matrices) and five measures based on iterated local search runs, namely: the number of pseudo-optimal solutions, average distance of local optima to closest global optima using two different local search procedures, the empirical fitness distance correlation coefficient based on the local search experiments; and the empirical fitness distance correlation coefficient based on the iterated local search experiments.

Both unsupervised and supervised neural network models were used to learn the relationships in the meta-dataset and automate the algorithm selection process. Two supervised neural network architectures were tested. The first model used as input the 9 features discussed above, and 3 outputs corresponding to each of the meta-heuristic performances. The neural network was able to successfully predict meta-heuristic performance, measured as the percentage deviation from the optimal solution.

The second supervised neural network explored another form of performance prediction: the goal was to predict which algorithm would perform best. This was modeled as a classification problem: given the 9 inputs, learn to classify each example according to the known classification (the best performing of the 3 available meta-heuristics). After training, the model was used on unclassified data (i.e. QAP instances where it was not known which algorithm was best). Again, the neural network was found to successfully predict the best algorithm with 94% accuracy. The study also considered an unsupervised model, self-organizing maps, to select the best algorithm by creating visual explorations of the performance of different algorithms under various conditions describing the complexity of the problem instances. Given the limited size of the data, this is a preliminary study, but it demonstrates the relevance of meta-learning ideas to the optimisation field.

4.2 Hyper-heuristics for 2D packing: selecting versus generating components

This section presents two hyper-heuristics for the 2D strip packing problem, one based on the selection approach and the other in the generation approach, and its main objective is to make clear the differences between the two approaches. While the selection approach chooses two components according to the item at hand, the generation approach evolves a single algorithm component which scores the best position to place an item.

There are many types of cutting and packing problems in one, two and three dimensions. A typology of these problems is presented by Wascher et al. [116], which explains the two dimensional strip packing problem in the context of other such problems. In the 2D packing problem, a set of items of various shapes must be placed onto a sheet with the objective of minimising the length of sheet that is required to accommodate the items. The sheet has a fixed width, and the required length of the sheet is measured as the distance from the base of the sheet to the item edge furthest from the base. The items may be rectangular or non-rectangular, and this characteristic classifies the problem as a regular or irregular 2D packing problem. This problem is known to be NP hard [50], and has many industrial applications as there are many situations where a set of items of different sizes must be cut from a sheet of material (for example, glass or metal) while minimising waste.

We first describe the work of [113], where the hyper-heuristic *selects* heuristics for the regular and irregular two dimensional strip packing problems. In this study, the solutions are constructed by packing one item at a time. A packing heuristic decides which item to pack next, and where to place it in the partial solution. Ten item selection heuristics and four placement heuristics are combined to produce 40 heuristics in total.

The main motivation to model that problem that way is that different heuristics perform well in different situations. For example, one heuristic may be good at packing small items, one heuristic may be good at the start of the packing process, and another heuristic may perform well when there are a variety of items to pack. The hyper-heuristic aims to select the best heuristic from those available, at each decision point.

After each item has been packed, the state of the remaining item list is determined, by calculating eight values, such as the fraction of large, medium and small items remaining. The hyper-heuristic represents a set of mappings from states to one of the 40 heuristics. This mapping is evolved with a genetic algorithm [113]. An interesting point here is the way the fitness of the GA is calculated. Each individual receives a set of five different instances to evaluate, and their fitness takes into account the difference between the hyper-heuristic being evolved and the results given by the single best heuristic (i.e. the percentage of usage for each object) and how many instances the individual has seen so far.

In contrast with the work just described, [19] uses a hyper-heuristic to generate heuristics for the regular 2D strip packing problem. Here, instead of pre-selecting item selection and placement heuristics, a single heuristic is generated and evolved with genetic programming. This heuristic is represented by a function, used to assign a score to all of the candidate positions of placement at each decision point, as shown in Fig. 4. This is useful as, when faced with multiple options of which item to pack next and where to put it in the partial solution, the optimal function to score those options is not obvious. Moreover, 2D packing problem instances can be separated into classes with different characteristics, and different scoring function heuristics will perform well on different instances. Given a problem class, it is not easy to manually generate a specialised heuristic for that class.

The terminals and functions used to generate these heuristics include the width and length of the items, the bin dimensions, and various metrics that calculate the

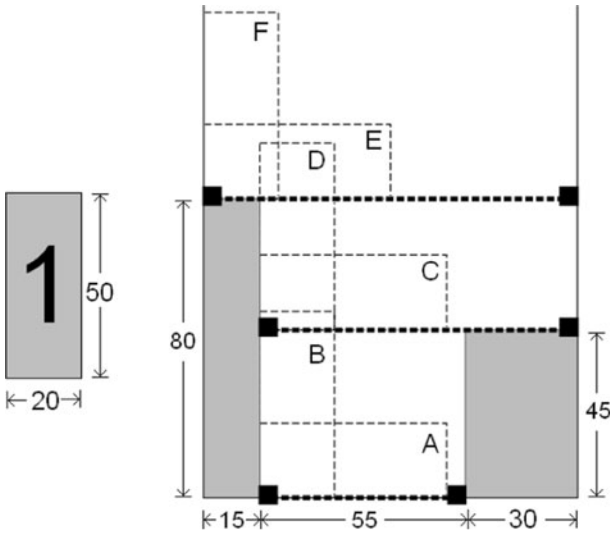


Fig. 4 The locations where item "1" can be placed. The heuristic must decide where to place the piece by assigning a score to each

differences between the bin and item sizes. Note that only the score function is evolved. The other basic algorithm elements, such as the loop which specifies that all of the items must be packed, are kept constant to ensure a feasible solution.

The literature on human-created heuristics for this problem focusses on scoring functions which are good over many different problem instances, with many different characteristics. The “best-fit” heuristic is shown to perform well over all benchmark instances. To manually create a heuristic for each different instance class would require a prohibitive amount of effort. Automating this creative process makes generating specialised heuristics viable, as no additional human intervention is necessary. All that is needed is a large enough set of representative training instances of the problem class and rich enough function and terminal sets.

4.3 Selecting/generating algorithm components for classification

While the previous section showed an example of hyper-heuristics using selection and generation of algorithm components, here we give an example of evolving an algorithm component through selection in the learning context. Two works are discussed: the first shows the use of an evolutionary algorithm to evolve a component of a decision tree induction algorithm: the heuristic for selecting the best attribute to split the tree [114]. The second discusses the evolution of (sets of) components of neural networks [119].

Decision tree algorithms build models by adding attributes to the tree, one at a time, according to their capabilities of discriminating examples belonging to different classes. Each value (in the case of categorical attributes) or range (in the case of numerical attributes) of the selected attribute is used to generate a new branch (split) in the tree. The discrimination power of each attribute is measured by a heuristic, and there are many in the literature that have already been explored, such as information gain and gain ratio. The method described here selects this heuristic.

The problem solving strategies (or hyper-heuristics) are represented by rules. These rules select the most appropriate splitting heuristic according to the degree of entropy of data attributes. An example of a rule is: *IF* ($x\% >$ *high*) *and* ($y\% <$ *low*) *THEN use heuristic H*, where x and y are both percentage values ranging from 0 to 100, and high and low are thresholds for the entropy value H. The meaning of the previous rule is: if x% of the attributes have entropy values above high, and if y% of attributes have entropy values below low, then use heuristic *H* to choose the splitting attribute at the current node.

Individuals represent rule sets of size *n*, and a set of 12 heuristics can be recommended. In order to ensure valid solutions, each individual is associated with a default heuristic. When a new split needs to be created, the conditions of the *n* rules are checked, and zero or more rules can hold. If no rule is selected, the default heuristic is applied. If one rule is selected, the heuristic that appears in its consequent is applied. Finally, if more than one rule holds, a majority voting is performed.

A few years after this first work, [9] proposed to generate a complete algorithm to create a decision tree induction algorithm from sub-components of well-known algorithms. This was achieved using a genetic algorithm with a linear individual representation where each gene value represents a specific choice in the design of one component of a decision tree algorithm. Different genes refer to different design choices for decision tree components like the criterion used to select attributes, parameters related to pruning, the stopping criteria for the tree-construction process, etc. Hence, the combination of values for all genes of an individual specify a unique complete decision tree induction algorithm.

Considering other classification models apart from decision trees, research on neural networks has invested a lot of effort on algorithm component selection and generation. Here we describe the main aspects of *neuroevolution* [46] (i.e. evolution of artificial neural networks). As pointed out by Yao [119], neural networks can be evolved at three different levels: synaptic weights choice, topology design and learning rule selection. Initially, neuroevolution followed mainly an algorithm selection approach but, nowadays, it is moving in the direction of algorithm generation [26]. The first use of the term neuroevolution was to evolve the connection weights of a population of neural networks. This basic idea grew and researchers started to evolve, together with the weights, the topology of the networks. In the past decade, research on learning algorithms has advanced significantly.

Here we focus on a specific example of Neuroevolution: NEAT (Neuroevolution of Augmenting Topologies) [108]. This system is well suited to reinforcement learning tasks, and evolves network topologies along with weights. As in all evolutionary systems, choosing how to represent the networks in the evolutionary process is one of the key design issues. In NEAT, each individual (neural network) is a linear representation of connecting genes, which specify the input node, output node, weight of the connection, whether or not the gene is expressed and an innovation number, used to mark genes belonging to the same 'evolution line'. Crossover and mutation operations may add perturbations to weights or add new connections or nodes, but they use the innovation numbers for historical tracking. During crossover, only genes with the same innovation number can be exchanged. The innovation number is assigned to a gene the first time it is created, and is kept unchanged during evolution. Hence, genes with the same innovation number must represent the same structure, maybe with different weights. The system also uses speciation to evolve different network structures simultaneously.

4.4 Generating complete heuristics/algorithms

This section presents two case studies of the *generation of algorithms*. In the same fashion as when generating heuristics, this approach is appropriate where there is a basic sequence of steps to generate an algorithm; a range of components that can be used to implement each step, and a set of variations of this sequence that can be tested, modified or extended (automatically or by hand). Here we describe case studies regarding the automatic creation of evolutionary algorithms and rule induction algorithms, both using genetic programming.

4.4.1 Evolving evolutionary algorithms with genetic programming

There have been many attempts to make evolutionary algorithms more self-adaptive, avoiding necessity for choosing from many types of operators and parameters. Initially, many self-adaptive systems focused on parameters [40], and others were created to evolve the operators of the evolutionary algorithm [39]. Going one step further, Spector [107] and Oltean [83] employ genetic programming to evolve the algorithms themselves.

This section discusses in detail the LGP proposed by Oltean [83], as it also has one particularly interesting characteristic: it generalizes for families of problems. Each individual in the LGP (the macro-level algorithm) corresponds to an evolutionary algorithm (the micro-level algorithm), where individuals differ from each other according to the order that the selection, crossover and mutation operations are performed. Figure 5 shows an example of an individual. Only the commands in the body of the *for* loop are evolved. Pop [8] represents a population with 8 individuals, and the first command of the body of the *for* loop mutates the individual in position 5 of the population and saves the result in position 0. In the second *for* loop line, the select acts as a binary tournament selection, choosing the best individual among those in positions 3 and 6 and storing it in position 7. The third command crosses over the individuals in positions 0 and 2, saving the result in position 2.

The fitness of LGP is calculated by running the micro-level evolutionary algorithm. However, given the stochastic nature of the method being evolved, the micro-level algorithm was run a predetermined number of times on a set of training problems. The average fitness of the micro-level algorithm was used to set its fitness in the macro-level algorithm.

This work was later extended [36], with the LGP replaced by a genetic algorithm. The main modification in this system is that now the algorithm evolves not only the evolutionary algorithm, but also their parameters, such as crossover and mutation probabilities.

```
void LGP Program(Chromosome  Pop[8])
{
    Randomly initialize the population();
    // repeat for a number of generations
    for (int k = 0; k < MaxGenerations; k++) {
        Pop[0] = Mutate(Pop[5]);
        Pop[7] = Select(Pop[3], Pop[6]);
        Pop[2] = Crossover(Pop[0], Pop[2]);
        Pop[4] = Mutate(Pop[2]);
        Pop[6] = Mutate(Pop[1]);
        Pop[2] = Select(Pop[4], Pop[3]);
        Pop[1] = Mutate(Pop[6]);
        Pop[3] = Crossover(Pop[5], Pop[1]);
    }
}
```

Fig. 5 LGP individual representing an Evolutionary Algorithm, adapted from [83]

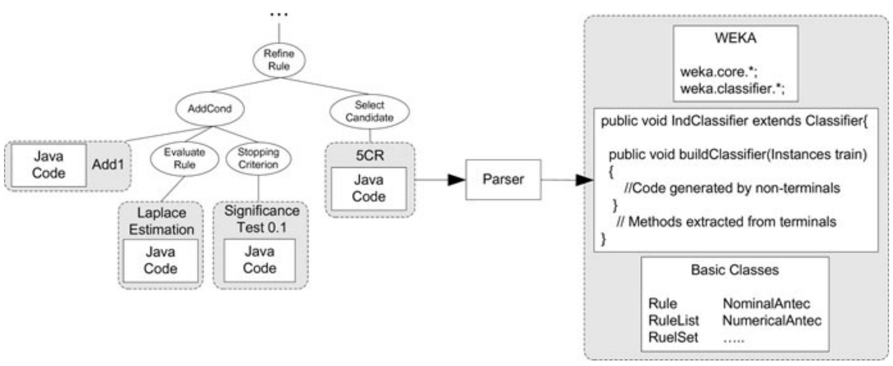
4.4.2 Automatically evolving rule induction algorithms

In [86, 87], the authors proposed a grammar-based genetic programming algorithm to evolve rule induction algorithms. The process of automatically evolving algorithms first requires a study of manually-designed algorithms. In the case of [87], the literature concerning rule induction algorithms was surveyed, and a set of algorithm components, such as rule search, evaluation and pruning, were identified. Different implementations of these components were found and added to the grammar, including methods which had not previously appeared in the literature. Loop and conditional statements were also added when appropriate, resulting in a grammar with 26 production rules.

Based on this grammar, a grammar-based genetic programming algorithm was used to generate, evaluate and evolve the rule induction algorithms (individuals). As the algorithm was being designed to work with any given dataset, the research challenge was to implement a fitness function that would facilitate the generalization of the produced algorithms. This problem was tackled using a set of datasets, termed the *meta-training set* to calculate the fitness of the individual. Hence, for each individual of the population, its corresponding rule induction algorithm was translated into Java code, and then run on the meta-training set. For each dataset in the meta-training set, a classification model was generated and its respective accuracy on the test set was calculated. The average accuracy in all datasets in the meta-training set was used as the fitness of the GP individual.

Figure 6 shows an example of the conversion from individual to Java code. Note that the individuals are trees, where each leaf node is associated with a portion of code that implements the respective function. These functions are combined with a set of core classes that implement the basics of any classifier.

The results showed that GP could generate rule induction algorithms different from those already proposed in the literature, and with competitive accuracy. Following another approach, the authors also proposed the use of the algorithm to generate algorithms targeted to a specific dataset [86] or datasets with similar characteristics.



```
public void IndClassifier extends Classifier{

  public void buildClassifier(Instances train)
  {
     //Code generated by non-terminals
  }
   // Methods extracted from terminals
}
```

Fig. 6 Example of the method used to convert GGP individuals in Java code [87]

4.5 Summary

This section summarizes and compares the methods previously described according to the three dimensions introduced in Sect. 3.3, namely problem space, algorithm space and evaluation metrics. To facilitate this comparison, Table 1 lists the methods according to the problem being tackled, the approach followed to generate the hyper-heuristic (selection or generation, components or algorithm/heuristic) and the search method used.

Recall that the problem space is defined according to the problem being solved and the level of abstraction at which one is working: components or algorithms/heuristics. Furtheremore, it considers if a hyper-heuristic is being developed for a set of datasets or for a specific dataset. Table 1 shows a sample of learning and optimisation problems solved so far considering both components and algorithms/heuristics. Note that all types of approaches have been previously explored. In terms of developing heuristics/algorithms for specific problems or sets of problems, all the work referenced in Table 1 uses multiple problem instances, divided into training and test sets.

The algorithm space, in turn, refers to whether we use a selection or a generation approach. This space can be explored by different search methods. In Table 1, we observe that initially machine learning methods, such as rule induction algorithms and neural networks were explored for selection. However, they do not offer enough expressive power to generate algorithms. Most of the methods created for generation are evolutionary algorithms, with a special emphasis on different types of genetic programming.

Finally, regarding the evaluation metric, the quality of the produced algorithms is measured with the average of a well-known problem-specific metric over different test instances. In optimisation, this is often translated as the error between the optimal hand-designed solution (when one is available) and the one found by the search method, or using some other measure of optimisation performance. In classification (supervised machine learning), any metric estimating the predictive accuracy of the results in the test set (containing only problem instances not present

Table 1 Comparison of the methods described in Sect. 4 considering the problem being tackled, the approach followed and the search method

Ref.	Problem	Approach followed	Search Method
Aha [2]	Data classification	Alg. selection	Rule induction alg.
Smith-Miles [105]	QAP problem	Heuristic selection	Neural networks
Terashima-Marin et al.[113]	2D packing	Heuristic selection	Genetic algorithm
Vella et al.[114]	Decision tree split	Component selection	Rule induction alg.
Stanley and Miikkulainen [108]	Neural networks	Component selection	Evolutionary algorithm
Burke et al.[19]	2D packing	Component generation	Genetic programming
Barros et al.[9]	Decision tree alg.	Alg. generation	Genetic algorithm
Oltean [83]	EA for 3 problems	Alg. generation	Linear GP
Pappa and Freitas [87]	Rule induction alg.	Alg. generation	Grammar-based GP

in the training set) can be used. Hence, generalization ability is always measured when evaluating a classification model's predictive performance. The only approach which differs from this, and an interesting research direction to be further investigated, is the one presented in [113], where different individuals evaluate different instances, and the fitness considers how many instances the individuals have seen so far, trying to improve generalization and prevent overfitting.

5 Discussion and conclusions

The previous sections reviewed the work done in optimisation and machine learning for automatic algorithm design, and identified that both areas work with two main approaches: those for selecting and those for generating heuristics and algorithms. Examples of systems following the two approaches in both domains were presented and compared.

Despite similarities, the issue of generalisation is the most different between automated approaches to machine learning and optimisation. This section emphasizes and discusses the differences between the current approaches for generalisation and asks how much further we can go. It also discusses some theoretical foundations and concludes with some final remarks.

5.1 Differences between current machine learning and optimisation approaches

In this paper we have explored the similarities and differences between recent research undertaken in supervised machine learning (classification) and optimisation. The first significant observation is regarding the levels of generalisation in the algorithms in the respective communities. The ability of algorithms to generalise to new datasets has long been a concern of the classification community. Within the context of optimization, starting in the 1970s, the application of machine-learning to planning has seen the emergence of increasingly sophisticated domain-independent planners [3–5, 10, 68, 69]. However, the wider optimisation community has only recently begun to seriously focus on automatically designing heuristic systems (hyper-heuristic research) which can adapt to new problem instance data without further human intervention.

In this section, we want to draw attention to these differences in levels of generalisation. Figure 7 shows the different levels of generality at which machine learning and optimisation systems can operate. Level A is the least general, and level C is the most general. In level A, also known as the *executable level*, the optimisation heuristic produced by the heuristic generator and the classifier (classification model) produced by the classification algorithm are directly executed on the training instance, and their performance is evaluated on the test instances. In level B, also called *generator level*, the methods generate a heuristic for optimisation or a classifier for supervised machine learning. Finally, when operating in level C, also named *meta-generator level*, machine learning methods act as meta-learning systems, generating a classifier generator. The *meta-generator level* is a recent development, and we are aware of only two *meta-generator level* systems [9, 86, 87].

In optimisation, a level A heuristic operates on problem instances to produce solutions. This is analogous to the case in supervised machine learning, where a classification model operates on a dataset specific to a data domain. Note that, although level A is the least general and involves both optimisation and classification, the latter requires that the model works well on new data coming from the same data domain (i.e. represented by the same set of features), which is not always required in optimisation. In level B, hyper-heuristics for optimisation automatically design heuristics, which can then operate on similar problem instances. The hyper-heuristic operates at a higher level of generality—it is a system which can automatically design specialised heuristics.

Most current classification algorithms are expected to work well over a variety of domains, and one would not generally hand-craft a classifier for a given dataset. For example, a neural network classifier would typically be automatically trained using an algorithm such as back-propagation on the target dataset, since hand-crafting a neural network is well-understood to be inefficient. This automatic training of 'level A' entities (using a classifier generator at the 'level B') has been a feature of machine learning approaches for decades. The same is not true in the optimisation community, where hyper-heuristic approaches which generate heuristics have only recently become successful, mainly due to the application of genetic programming.

Recent application of genetic programming in classification has resulted in a meta-learning system which can automatically generate new classification algorithms [9, 87], which are in turn used to generate classifiers (classification models for the input data). This system operates at level C in Fig. 7. For example, the system can produce an algorithm for one dataset with a certain set of features, and then also produce a different algorithm for a second dataset with a different set of features. This process of algorithm generation is automated, and allows one system to operate over different problem domains.

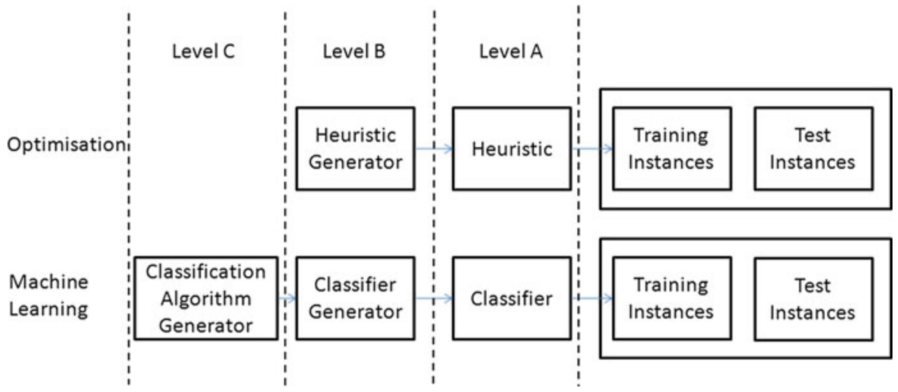


Fig. 7 The levels of generality at which methods for automating the design of optimisation and machine learning algorithms operate

5.2 Generalisation: future research directions

The fact that the latest machine learning research can operate at level C (in Fig. 7) means that it can operate over different datasets, from different problem domains, and even with different features. Genetic programming-based hyper-heuristics for optimisation have so far only been shown to operate at level B, meaning that one system can generate heuristics for one problem domain. We can generate a human-competitive heuristic for 2D packing, but the same system cannot generate a heuristic for the traveling salesman problem, or for vehicle routing. To do so would require a fundamentally different set of functions and terminals, or (in the case of grammar-based genetic programming) a different grammar. Recognising this current level of generality is a key contribution of this paper, and we argue that removing this limitation on optimisation systems should be a focus of current and future research. One possible way forward is adopt one or more of the domain description languages (e.g. [54, 72]) (many of which have evolved from the well-known STRIPS solver [44]) that have enjoyed success in the wider Artificial Intelligence community. This would also have the added advantage of making optimization problems amenable to processing by a wider variety of problem-solving architectures than is currently the case.

If we instead consider heuristic selection (as opposed to heuristic generation), the winning algorithm of the Cross-domain Heuristic Search Challenge (CHeSC) did in fact show high generality across problem domains. It was shown that it was possible to design one hyper-heuristic that could provide good solutions to instances of six different problem domains, including two that were not seen by the hyper-heuristic designers (travelling salesman and vehicle routing). For each problem domain, a problem representation, fitness function and a set of low-level heuristics, encapsulating the domain specific components, was provided. The task of the hyper-heuristic was to intelligently select the sequence of heuristics to apply, based on their performance and characteristics. This is very different to the related goal of automatically generating the heuristics themselves, but the results of CHeSC represent the state of the art in automatic selection of algorithms for optimisation problems. Details of the challenge and the results can be found at http://www.asap.cs.nott.ac.uk/external/chesc2011/, and the challenge was made possible by the use of the HyFlex framework [78] through which the participants developed their hyper-heuristics. HyFlex is available for future research at the CHeSC website, allowing comparison with the competition results.

Another point that needs to be explored is how to make the algorithm even less dependent on the human designer, making him/her less responsible for the definition of a set of pre-defined functions, increasing even more the level of generality.

5.3 Challenges relating to datasets and training

When we talk about using various datasets for training and testing in the context of the automatic design of classification algorithms, there are a lot of public repositories to be explored, such as the one maintained by the University of California Irvine (UCI) [47], or the huge amounts of bioinformatics data freely available on the Web. The same is not always true in optimisation tasks. In order to obtain a number of instances large enough for training and test purposes, the use of problem-instance generators might be necessary.

The use of many datasets as a training set also raises problems of performance, as each individual in the population must be tested on the training data. This problem is related to both the number and size of the training data. In the case of machine learning, research on intelligent sampling methods [34, 67] might be necessary to enable the use of the methods. This problem has been already explored in other contexts, such as selecting a subset of instances for the user to label or reducing data sets for evolutionary algorithms fitness computation, and some such well-understood methods could certainly be applied.

Alternative strategies can be used to reduce the time spent on fitness computation while improving the generalization of the solutions. In [42], for example, the available instances are divided into groups according to their level of difficulty. Initially, solutions are evaluated using easy instances and, as the solutions get better, the initial instances are replaced by harder ones. Terashima-Marin et al. [113], in contrast, evaluates different solutions using different subsets of instances, and the fitness of individuals take into account how many instances they have seen so far.

However, even if algorithm generators take a considerable amount of time to run, it is important to remember that this time would probably still be a small fraction of the time taken by the human-designer of a new algorithm. Furthermore, when the focus is on automatically designing heuristics/algorithms that are robust across many different types of datasets, constructing new heuristics or algorithms would not be a task frequently performed, as the intention is that the generated algorithms will be reused.

5.4 Foundational studies

Thus far, little progress has been made in enhancing our theoretical understanding of hyper-heuristics and the automated design of algorithms. The theoretical study of the run-time and convergence properties of algorithms that explore the complex search spaces of algorithms/heuristics seems far from feasible. One direction that does seem promising is to study the structure of the search space of heuristics/algorithms. The approaches explored in this article generally involve searching in multiple spaces simultaneously. An additional question is then how the different search spaces interact.

Analysis of the heuristic search spaces in hyper-heuristics for educational timetabling and production scheduling [79, 80] revealed common features, viz. (i) a 'big-valley' structure in which the cost of local optima and their distances to the global optimum (best-known solution) are correlated; (ii) the presence of a large number of distinct local optima, many of them of low quality; and (iii) the existence of plateaus (neutrality): many different local optima are located at the same level in the search (i.e. have the same value). It remains to be seen whether such features occur in other heuristic search spaces. In these studies, the heuristic search space is generally smaller in size (when compared to the solution space). With respect to the mapping between the two spaces; the heuristic search space seems to cover only a subset of the solution search space (but well distributed areas) [91].

Finally, since genetic programming is an important technique in the approaches explored in this article, the theoretical studies in this area are of relevance [89].

5.5 Final remarks

This paper focused on the automatic design of optimisation and supervised machine learning (classification) methods. Many real-world problems can be modeled as optimisation or classification problems, so the issues discussed here are widely relevant. However, other domains, such as bioinformatics, control, constraint programming and games have already investigated forms of both automated algorithm/heuristic selection and generation [5, 41, 74, 90]. We argue that algorithm/heuristics selection and generation are crucial for all types of domains in which many methods and/or parameters are available, but no clear methodology or criteria for choosing them are available.

Our discussion concentrated on the role of genetic programming and evolutionary algorithms as a methodology for designing algorithms/heuristics, although some of the case studies discussed involved other methods such as neural networks and other metaheuristics. We believe that the representation power offered by different types of genetic programming system makes them a suitable methodology. However, the exploration of other techniques is also a direction worth pursuing.

Optimisation and classification are generally considered as to be distinct. However, while the problem types and methods are different, the meta-learning and hyper-heuristic methodologies discussed in this paper share a focus on automatic design of heuristic methods. Comparing and contrasting the various approaches in this paper will hopefully lead to closer collaboration and significant research progress.

Acknowledgments G. L. Pappa, A. A. Freitas, J. Woodward and J. Swan would like to thank the 2011 GECCO's workshop organizers for putting them in contact. G. L. Pappa was partially supported by CNPq, CAPES and FAPEMIG, all Brazilian Research Agencies. G. Ochoa and J. Swan were supported by UK Research Council EPSRC, grants EP/F033214/1 and EP/D061571/1. J. Wooward was supported by UK Research Council EPSRC, grant EP/J017515/1. A.A. Freitas was partially supported by UK Research Council EPSRC, grant EP/H020217/1.

References

1. C. Adami, T.C. Brown, Evolutionary learning in the 2d artificial life system avida. in *Artificial Life IV*, ed. by R.A. Brooks, P. Maes (MIT Press, Cambridge, 1994), pp. 377–381
2. D.W. Aha, Generalizing from case studies: A case study. in *Proceedings of the Ninth International Conference on Machine Learning*. (Morgan Kaufmann, Burlington, 1992), pp. 1–10
3. R. Aler, D. Borrajo, P. Isasi, Evolving heuristics for planning. in *Lecture Notes in Computer Science*. (1998)
4. R. Aler, D. Borrajo, P. Isasi, Learning to solve planning problems efficiently by means of genetic programming. Evol. Comput. **9**(4), 387–420 (2001)
5. R. Aler, D. Borrajo, P. Isasi, Using genetic programming to learn and improve control knowledge. Artif. Intell. **141**(1-2), 2956 (2002)

6. P.J. Angeline, Adaptive and self-adaptive evolutionary computations. in *Computational Intelligence: A Dynamic Systems Perspective*. (IEEE Press, New York, 1995), pp. 152–163
7. T. Bäck, An overview of parameter control methods by self-adaption in evolutionary algorithms. Fundam. Inf. **35**(1-4), 51–66 (1998)
8. W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, *Genetic Programming: An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. (Morgan Kaufmann, San Francisco, 1998)
9. R.C. Barros, M.P. Basgalupp, A.C. de Carvalho, A.A. Freitas, A hyper-heuristic evolutionary algorithm for automatically designing decision-tree algorithms. in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference, GECCO '12*. (2012), pp. 1237–1244
10. D. Borrajo, M. Veloso, Lazy incremental learning of control knowledge for efficiently obtaining quality plans. AI Rev. J. Spec. Issue Lazy Learn. **11**, 371–405 (1996)
11. P. Brazdil, C. Giraud-Carrier, C. Soares, R. Vilalta, *Metalearning: Applications to Data Mining*. (Springer, Berlin, 2008)
12. P.B. Brazdil, C. Soares, J.P. Da Costa, Ranking learning algorithms: using ibl and meta-learning on accuracy and time results. Mach. Learn. **50**(3), 251–277 (2003)
13. L. Breiman, Bagging predictors. Mach. Learn. **24**, 123–140 (1996)
14. E.K. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, S. Schulenburg, Hyper-heuristics: an emerging direction in modern search technology. in *Handbook of Metaheuristics*, ed. by F. Glover, G. Kochenberger (Kluwer, Dordrecht, 2003), pp. 457–474
15. E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, J. Woodward, Exploring hyper-heuristic methodologies with genetic programming. in *Computational Intelligence: Collaboration, Fusion and Emergence, Intelligent Systems Reference Library*. ed. by C. Mumford, L. Jain (Springer, Berlin, 2009), pp. 177–201
16. E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, J. Woodward, *Handbook of Metaheuristics, International Series in Operations Research & Management Science, vol. 146, chap. A Classification of Hyper-heuristic Approaches*. (Springer 2010), Chapter 15, pp. 449–468
17. E.K. Burke, M. Hyde, G. Kendall, J. Woodward, Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*. (London, UK 2007), pp. 1559–1565
18. E.K. Burke, M.R. Hyde, G. Kendall, Grammatical evolution of local search heuristics. IEEE Transactions on Evolutionary Computation **16**(3), 406–417 (2012)
19. E.K. Burke, M.R. Hyde, G. Kendall, J. Woodward, A genetic programming hyper-heuristic approach for evolving two dimensional strip packing heuristics. IEEE Transactions on Evolutionary Computation **14**(6), 942–958 (2010)
20. E.K. Burke, M.R. Hyde, G. Kendall, J. Woodward, Automating the packing heuristic design process with genetic programming. Evol. Comput. **20**(1), 63–89 (2012)
21. E.K. Burke, G. Kendall, J.D. Landa-Silva, R. O'Brien, E. Soubeiga, An ant algorithm hyperheuristic for the project presentation scheduling problem. in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation, vol. 3*. (2005), pp. 2263–2270
22. E.K. Burke, G. Kendall, E. Soubeiga, A tabu-search hyperheuristic for timetabling and rostering. J. Heuristics **9**(6), 451–470 (2003)
23. E.K. Burke, B. McCollum, A. Meisels, S. Petrovic, R. Qu, A graph-based hyper-heuristic for educational timetabling problems. Eur. J. Oper. Res. **176**, 177–192 (2007)
24. E.K. Burke, S. Petrovic, R. Qu, Case based heuristic selection for timetabling problems. J. Sched. **9**(2), 115–132 (2006)
25. J. Cano-Belmán, R. Ríos-Mercado, J. Bautista, A scatter search based hyper-heuristic for sequencing a mixed-model assembly line. J. Heuristics **16**, 749–770 (2010)
26. E. Cantu-Paz, C. Kamath, An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems. IEEE Trans. Syst. Man Cybern. Part B Cybern. **35**(5), 915–927 (2005)
27. K. Chakhlevitch, P.I. Cowling, Hyperheuristics: Recent developments. in *Adaptive and Multilevel Metaheuristics Studies in Computational Intelligence, vol. 136*, ed. by C. Cotta, M. Sevaux, K. Sörensen (Springer, Berlin, 2008), pp. 3–29
28. A. Chandra, X. Yao, Ensemble learning using multi-objective evolutionary algorithms. J Math. Model. Algorithms **5**, 417–445 (2006)

29. P.C. Chen, G. Kendall, G. Vanden Berghe, An ant based hyper-heuristic for the travelling tournament problem. in *Proceedings of IEEE Symposium of Computational Intelligence in Scheduling (CISched 2007)*, (2007), pp. 19–26
30. P. Cowling, G. Kendall, E. Soubeiga, A hyperheuristic approach for scheduling a sales summit. in *Selected Papers of the Third International Conference on the Practice And Theory of Automated Timetabling, PATAT 2000, LNCS* (Springer, Konstanz, Germany, 2000), pp. 176–190
31. P. Cowling, G. Kendall, E. Soubeiga, A hyperheuristic approach for scheduling a sales summit. in *Selected Papers of the Third International Conference on the Practice And Theory of Automated Timetabling, PATAT 2000* (Springer, Berlin, 2001), pp. 176–190
32. L. Cruz-Reyes, C. Gómez-Santillán, J. Pérez-Ortega, V. Landero, M. Quiroz, A. Ochoa, *Intelligent Systems, chap. Algorithm Selection: From Meta-Learning to Hyper-Heuristics*. (InTech, 2012), pp. 77–102
33. A. Cuesta-Cañada, L. Garrido, H. Terashima-Marin, Building hyper-heuristics through ant colony optimization for the 2d bin packing problem. in *Knowledge-Based Intelligent Information and Engineering Systems*. ed. by R. Khosla, R. Howlett, L. Jain (Springer, Berlin, 2005), p. 907
34. R. Curry, P. Lichodzijewski, M. Heywood, Scaling genetic programming to large datasets using hierarchical dynamic subset selection. IEEE Trans. Syst. Man Cybern. Part B Cybern. **37**(4), 1065–1073 (2007)
35. C. Dimopoulos, A.M.S. Zalzala, Investigating the use of genetic programming for a classic one-machine scheduling problem. Adv. Eng. Softw. **32**(6), 489–498 (2001)
36. L.S. Diosan, M. Oltean, Evolving evolutionary algorithms using evolutionary algorithms. in *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation, GECCO '07*. (New York, NY, USA, 2007), pp. 2442–2449
37. K.A. Dowsland, E. Soubeiga, E.K. Burke, A simulated annealing hyper-heuristic for determining shipper sizes. Eur. J. Oper. Res. **179**(3), 759–774 (2007)
38. B. Edmonds, *Meta-genetic programming: Co-evolving the operators of variation*. Tech. rep., Centre for Policy Modelling, Manchester Metropolitan University (1998)
39. B. Edmonds, Meta-genetic programming: Co-evolving the operators of variation. Turk. J. Elec. Engin. **9**(1), 13–29 (2001)
40. A.E. Eiben, Z. Michalewicz, M. Schoenauer, J.E. Smith, Parameter control in evolutionary algorithms. IEEE Trans. Evol. Comput. **2**(3), 124–141 (1999)
41. A. Elyasaf, A. Hauptman, M. Sipper, Ga-freecell: evolving solvers for the game of freecell. in *Proceedings of the 13th annual conference on Genetic and evolutionary computation, GECCO '11*. (ACM, New York, NY, USA, 2011), pp. 1931–1938
42. A. Elyasaf, A. Hauptman, M. Sipper, Evolutionary design of freecell solvers. IEEE Trans. Comput. Intell. AI Games **4**(4), 270–281 (2012)
43. H.L. Fang, P. Ross, D. Corne, A promising genetic algorithm approach to job shop scheduling, rescheduling, and open-shop scheduling problems. in *5th International Conference on Genetic Algorithms* ed. by S. Forrest (Morgan Kaufmann, San Mateo, 1993), pp. 375–382
44. R. Fikes, N.J. Nilsson, Strips: a new approach to the application of theorem proving to problem solving. in *IJCAI*. (1971), pp. 608–620
45. H. Fisher, G.L. Thompson, Probabilistic learning combinations of local job-shop scheduling rules. in *Industrial Scheduling*, ed. by J.F. Muth, G.L. Thompson (Prentice-Hall, Inc, New Jersey, 1963), pp. 225–251
46. D. Floreano, P. Durr, C. Mattiussi, Neuroevolution: from architectures to learning. Evol. Intel. **1**, 47–62 (2008)
47. A. Frank, A. Asuncion, UCI machine learning repository (2010). http://archive.ics.uci.edu/ml
48. P.W. Frey, D.J. Slate, Letter recognition using holland-style adaptive classifiers. Mach. Learn. **6**, 161–182 (1991)
49. A.S. Fukunaga, Automated discovery of local search heuristics for satisfiability testing. Evol. Comput. **16**(1), 31–61 (2008)
50. M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the theory of NP-Completeness*. (W.H. Freeman and Company, San Fransisco, 1979)
51. P. Garrido, M. Riff, Dvrp: A hard dynamic combinatorial optimisation problem tackled by an evolutionary hyper-heuristic. J. Heuristics **16**, 795–834 (2010)
52. C.D. Geiger, R. Uzsoy, H. Aytug, Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. J. Sched. **9**(1), 7–34 (2006)

53. L. Georgiou, W.J. Teahan, jGE: a java implementation of grammatical evolution. in *Proceedings of the 10th WSEAS International Conference on Systems*. (World Scientific and Engineering Academy and Society (WSEAS), 2006), pp. 410–415
54. M. Ghallab, C.K. Isi, S. Penberthy, D.E. Smith, Y. Sun, D. Weld, *PDDL - The Planning Domain Definition Language*. Tech. Rep. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
55. F. Glover, Future paths for integer programming and links to artificial intelligence. Comput. Opert. Res. **13**(5), 533–549 (1986)
56. H.J. Goldsby, B.H. Cheng, Avida-mde: a digital evolution approach to generating models of adaptive software behavior. in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, GECCO '08* (2008), pp. 1751–1758
57. J. Gratch, S. Chien, Adaptive problem-solving for large-scale scheduling problems: a case study. J. Artif. Intel. Res. **4**, 365–396 (1996)
58. J. Grefenstette, Optimization of control parameters for genetic algorithms. IEEE Trans. Syst. Man Cybern. **16**(1), 122–128 (1986)
59. J.J. Grefenstette, Optimization of control parameters for genetic algorithms. IEEE Trans. Syst. Man Cybern. **SMC-16**(1), 122–128 (1986)
60. A. Hauptman, A. Elyasaf, M. Sipper, A. Karmon, Gp-rush: using genetic programming to evolve solversforthe rushhour puzzle. in *Genetic and evolutionary computation (GECCO 2009)*. (ACM, 2009), pp. 955–962
61. M.R. Hyde, E.K. Burke, G. Kendall, Automated code generation by local search. J. Oper. Res. Soc. (2012). doi:10.1057/jors.2012.149
62. A. Keleş, A. Yayimli, A.C. Uyar, Ant based hyper heuristic for physical impairment aware routing and wavelength assignment. in *Proceedings of the 33rd IEEE conference on Sarnoff*. (Piscataway, NJ, USA, 2010), pp. 90–94
63. Y. Kodratoff, D. Sleeman, M. Uszynski, K. Causse, S. Craw, Building a machine learning toolbox. in *Enhancing the Knowledge Engineering Process*, ed. by Steels L., Lepape (1992), pp. 81–108
64. J.R. Koza, *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. (The MIT Press, Massachusetts, 1992)
65. O. Kramer, Evolutionary self-adaptation: a survey of operators and strategy parameters. Evol. Intel. **3**, 51–65 (2010)
66. N. Krasnogor, S. Gustafson, A study on the use of "self-generation" in memetic algorithms. Nat. Comput. **3**(1), 53–76 (2004)
67. C.W.G. Lasarczyk, P. Dittrich, J.C.F. Bioinformatics, W. Banzhaf, Dynamic subset selection based on a fitness case topology. Evol. Comput. **12**, 223–242 (2004)
68. J. Levine, D. Humphreys (2003) Learning action strategies for planning domains using genetic programming. in *EvoWorkshops*. (2003), pp. 684–695
69. J. Levine, H. Westerberg, M. Galea, D. Humphreys, Evolutionary-based learning of generalised policies for ai planning domains. in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09*. (ACM, New York, 2009), pp. 1195–1202
70. K. Leyton-Brown, E. Nudelman, Y. Shoham, Learning the empirical hardness of optimization problems: The case of combinatorial auctions. in *Principles and Practice of Constraint Programming - CP 2002, Lecture Notes in Computer Science, vol. 2470*, ed. by P. Van Hentenryck (Springer, Berlin, 2002), pp. 91–100
71. J. Maturana, F. Lardeux, F. Saubion, Autonomous operator management for evolutionary algorithms. J. Heuristics **16**, 881–909 (2010)
72. D.V. McDermott, Pddl2.1 - the art of the possible? commentary on fox and long. J. Artif. Intell. Res. (JAIR) **20**, 145–148 (2003)
73. D. Michie, D. Spiegelhalter, C. Taylor (eds), *Machine Learning, Neural and Statistical Classification*. (Ellis Horwood, Chichester, 1994)
74. S. Minton, Automatically configuring constraint satisfaction problems: a case study. Constraints **1**(1), 7–43 (1996)
75. T. Mitchell, *Machine Learning (Mcgraw-Hill International Edit), 1st edn*. (McGraw-Hill Education, New York, (ISE Editions), 1997)
76. A.Y. Ng, Preventing overfitting of cross-validation data. in *Proceedings of the Fourteenth International Conference on Machine Learning, ICML '97*. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997), pp. 245–253

77. M. Nicolau, libGE: Grammatical evolution library for c++. Available from: http://waldo.csisdmz.ul.ie/libGE (2006)
78. G. Ochoa, M. Hyde, T. Curtois, J. Vazquez-Rodriguez, J. Walker, M. Gendreau, G. Kendall, B. McCollum, A. Parkes, S. Petrovic, E. Burke, HyFlex: A Benchmark Framework for Cross-domain Heuristic Search **7245**, 136–147 (2012)
79. G. Ochoa, R. Qu, E.K. Burke, Analyzing the landscape of a graph based hyper-heuristic for timetabling problems. in *Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2009).* (Montreal, Canada, 2009)
80. G. Ochoa, J.A. Váquez-Rodríguez, S. Petrovic, E.K. Burke, Dispatching rules for production scheduling: a hyper-heuristic landscape analysis. in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2009).* (Montreal, Norway, 2009)
81. G. Ochoa, J. Walker, M. Hyde, T. Curtois, Adaptive evolutionary algorithms and extensions to the hyflex hyper-heuristic framework. in *Parallel Problem Solving from Nature - PPSN 2012, vol. 7492*. (Springer, Berlin, 2012), pp. 418–427
82. C. Ofria, C.O. Wilke, Avida: A software platform for research in computational evolutionary biology. Artif. Life **10**(2), 191–229 (2004)
83. M. Oltean, Evolving evolutionary algorithms using linear genetic programming. Evol. Comput. **13**, 387–410 (2005)
84. M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, A. Brabazon, GEVA: Grammatical evolution in Java. SIGEVOlution **3**(2), (2008)
85. E. Özcan, B. Bilgin, E.E. Korkmaz, A comprehensive analysis of hyper-heuristics. Intell. Data Anal. **12**(1), 3–23 (2008)
86. G.L. Pappa, A.A. Freitas, Automatically evolving rule induction algorithms tailored to the prediction of postsynaptic activity in proteins. Intell. Data Anal. **13**(2), 243–259 (2009)
87. G.L. Pappa, A.A. Freitas, *Automating the Design of Data Mining Algorithms: An Evolutionary Computation Approach*. (Springer, Berlin, 2009)
88. D. Pisinger, S. Ropke, A general heuristic for vehicle routing problems. Comput. Oper. Res. **34**, 2403–2435 (2007)
89. R. Poli, L. Vanneschi, W.B. Langdon, N.F. McPhee, Theoretical results in genetic programming: the next ten years? Genet. Program. Evolvable Mach. **11**(3-4), 285–320 (2010)
90. D. Posada, K.A. Crandall, Modeltest: testing the model of *dna* substitution. Bioinformatics **14**(9), 817–818 (1998)
91. R. Qu, E.K. Burke, Hybridisations within a graph based hyper-heuristic framework for university timetabling problems. J. Oper. Res. Soc. **60**, 1273–1285 (2009)
92. R.B. Rao, D. Gordon, W. Spears, For every generalization action, is there really an equal and opposite reaction? Analysis of the conservation law for generalization performance. in *Proc. of the 12th International Conference on Machine Learning*. (Morgan Kaufmann, 1995), pp. 471–479
93. I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. (Frommann-Holzboog, Stuttgart, 1973)
94. J.R. Rice, The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976)
95. L. Rokach, Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography. Comput. Stat. Data Anal. **53**(12), 4046–4072 (2009)
96. P. Ross, Hyper-heuristics. in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, chap. 17*, ed. by E.K. Burke, G. Kendall (Springer, Berlin, 2005), pp. 529–556
97. P. Ross, J.G. Marín-Blázquez, Constructive hyper-heuristics in class timetabling. in *IEEE Congress on Evolutionary Computation*. (2005), pp. 1493–1500
98. P. Ross, S. Schulenburg, J.G. Marin-Blazquez, E. Hart, Hyper-heuristics: learning to combine simple heuristics in bin-packing problem. in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'02* (2002)
99. O. Roux, C. Fonlupt, Ant programming: or how to use ants for automatic programming. in *Proceedings of ANTS'2000*, ed. by M. Dorigo, E. Al (Brussels, Belgium, 2000), pp. 121–129
100. A. Salehi-Abari, T. White, Enhanced generalized ant programming. in *Proceedings of the 2008 Genetic and Evolutionary Computation Conference GECCO*. (ACM Press, 2008), pp. 111–118
101. C. Schaffer, A conservation law for generalization performance. in *Proc. of the 11th International Conference on Machine Learning*. (Morgan Kaufmann, 1994), pp. 259–265
102. R. Schapire, The strength of weak learnability. Mach. Learn. **5**, 197–227 (1990)

103. H.P. Schwefel, *Numerische Optimierung von Computer-Modellen Mittels der Evolutionstrategie, ISR, vol. 26*. (Birkhaeuser, Basel/Stuttgart, 1977)
104. Y. Shan, R. McKay, D. Essam, H. Abbass, A survey of probabilistic model building genetic programming. in *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*, ed. by M. Pelikan, K. Sastry, E. Cantu-Paz (Springer, Berlin, UK, 2006), pp. 121–160
105. K. Smith-Miles, Towards insightful algorithm selection for optimisation using meta-learning concepts. in *Proc. of IEEE International Joint Conference on Neural Networks IJCNN 2008*. (2008), pp. 4118–4124
106. K.A. Smith-Miles, Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. **41**, 6:1–6:25 (2008)
107. L. Spector, *Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems, vol. 8* (Springer, Berlin, 2010), pp. 17–33
108. K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies. Evol. Comput. **10**, 99–127 (2002)
109. T. Stutzle, S. Fernandes, *New Benchmark Instances for the QAP and the Experimental Analysis of Algorithms, Lecture Notes in Computer Science, vol. 3004*. (Springer, Berlin/Heidelberg, 2004), pp. 199–209
110. E.G. Talbi, *Metaheuristics: From Design to Implementation*. (Wiley, London, 2009)
111. J.C. Tay, N.B. Ho, Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. Comput. Ind. Eng. **54**, 453–473 (2008)
112. H. Terashima-Marin, E.J. Flores-Alvarez, P. Ross, Hyper-heuristics and classifier systems for solving 2D-regular cutting stock problems. in *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2005*. (2005), pp. 637–643
113. H. Terashima-Marin, P. Ross, C.J. Farias Zarate, E. Lopez-Camacho, M. Valenzuela-Rendon, Generalized hyper-heuristics for solving 2D regular and irregular packing problems. Ann. Oper. Res. **179**(1), 369–392 (2010)
114. A. Vella, D. Corne, C. Murphy, Hyper-heuristic decision tree induction. in *Nature Biologically Inspired Computing, 2009. NaBIC 2009*. (World Congress on, 2009), pp. 409 – 414
115. R. Vilalta, Y. Drissi, A perspective view and survey of meta-learning. Artif. Intell. Rev. **18**, 77–95 (2002)
116. G. Wäscher, H. Haußner, H. Schumann, An improved typology of cutting and packing problems. European Journal of Operational Research **183**(3), 1109–1130 (2007)
117. D.H. Wolpert, Stacked generalization. Neural Netw. **5**, 241–259 (1992)
118. D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization. IEEE Trans. Evol. Comput. **1**(1), 67–82 (1997)
119. X. Yao, Evolving artificial neural networks. Proc. IEEE **87**(9), 1423–1447 (1999)