Exploring Hyper-heuristic Methodologies with Genetic Programming

Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward*

Abstract. Hyper-heuristics represent a novel search methodology that is motivated by the goal of automating the process of selecting or combining simpler heuristics in order to solve hard computational search problems. An extension of the original hyper-heuristic idea is to generate new heuristics which are not currently known. These approaches operate on a search space of heuristics rather than directly on a search space of solutions to the underlying problem which is the case with most meta-heuristics implementations. In the majority of hyper-heuristic studies so far, a framework is provided with a set of human designed heuristics, taken from the literature, and with good measures of performance in practice. A less well studied approach aims to *generate* new heuristics from a set of potential heuristic *components*. The purpose of this chapter is to discuss this class of hyper-heuristics, in which Genetic Programming is the most widely used methodology. A detailed discussion is presented including the steps needed to apply this technique, some representative case studies, a literature review of related work, and a discussion of relevant issues. Our aim is to convey the exciting potential of this innovative approach for automating the heuristic design process.

1 Introduction

Heuristics for search problems can be thought of as "*rules of thumb*" for algorithmic problem solving [53]. They are not guaranteed to produce optimal solutions, rather,

Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, and Ender Ozcan
University of Nottingham, School of Computer Science, Jubilee Campus, Wollaton Road
Nottingham, NG8 1BB, UK
e-mail: {ekb,mrh,gxk,gxo,exo}@cs.nott.ac.uk

John R. Woodward
University of Nottingham, 199 Taikang East Road, Ningbo 315100, China
e-mail: john.woodward@nottingham.edu.cn

* Corresponding author.

C.L. Mumford and L.C. Jain (Eds.): Computational Intelligence, ISRL 1, pp. 177–201.
springerlink.com

the goal is to quickly generate good quality solutions. They are often used when exact methods are unable to be employed in a feasible amount of time. Genetic Programming is a method of searching a space of computer programs, and therefore is an automatic way of producing programs. This chapter looks at the use of Genetic Programming to automatically generate heuristics for a given problem domain. A knowledge of Genetic Programming is assumed, and while a brief introduction is given, readers unfamiliar with the methodology are referred to suitable tutorials and textbooks.

1.1 The Need for Heuristics

Combinatorial problems arise in many disciplines such as artificial intelligence, logistics, operational research, finance and bioinformatics. Prominent examples are tasks such as finding shortest round trips in graphs (the travelling salesman problem), finding models of propositional formulae (Boolean satisfiability), or determining the 3D structure of proteins (the protein folding problem). Other well-known combinatorial problems are found in scheduling, planning, resource and space allocation, cutting and packing, software and hardware design, and genome sequencing. These problems are concerned with finding assignments, orderings or groupings of a discrete set of objects that satisfy certain constraints [30].

Most real-world combinatorial problems such as scheduling and planning, are difficult to solve. The main difficulty arises from the extremely large and/or heavily constrained search spaces, and the noisy/dynamic nature of many real-world scenarios. In practice, we often deal with them using *heuristic methods*, which have no guarantee of optimality and that often incorporate stochastic elements. Over the years, a large variety of heuristic methods have been proposed and are widely applied. Often, heuristics are the result of years of work by a number of experts. An interesting question is how can we automate the design of heuristics, and it is this question which represents the underlying motivation for this chapter. *Hyper-heuristics* [9, 49, 53] are search methodologies for choosing or generating (combining, adapting) heuristics (or components of heuristics), in order to solve a range of optimisation problems. We begin by looking at hyper-heuristics employed across a broad spectrum of applications in more detail.

1.2 Hyper-heuristics

The main feature of hyper-heuristics is that they search a space of heuristics rather than a space of solutions directly. In this sense, they differ from most applications of meta-heuristics, although, of course, meta-heuristics can be (and have been) used as hyper-heuristics. The motivation behind hyper-heuristics is to raise the level of generality at which search methodologies operate. Introductions to hyper-heuristics can be found in [9, 53].

An important (very well known) observation which guides much hyper-heuristic research is that different heuristics have different strengths and weaknesses. A key idea is to use members of a set of known and reasonably understood heuristics to

either: (i) transform the state of a problem (in a constructive strategy), or (ii) perform an improvement step (in a perturbative strategy). Such hyper-heuristics have been successfully applied to bin-packing [54], personnel scheduling [13, 17], timetabling [1, 13, 14, 15, 59], production scheduling [61], vehicle routing problems [52], and cutting stock [58]. Most of the hyper-heuristic approaches incorporate a learning mechanism to assist the selection of low-level heuristics during the solution process. Several learning strategies have been studied such as reinforcement learning [17, 46], Bayesian learning [45], learning classifier systems [54], and case based reasoning [15]. Several meta-heuristics have been applied as search methodologies upon the heuristic search space. Examples are tabu search [13, 14], genetic algorithms [23, 29, 58, 59, 61], and simulated annealing [4, 18, 52]. This chapter focusses on Genetic Programming as a hyper-heuristic for generating heuristics, given a set of heuristic components.

1.3 Genetic Programming

Computers do not program themselves; they need a qualified and experienced programmer who understands the complexity of the task. An alternative to paying a human programmer to design and debug a program, is to build a computer system to evolve a program. This may not only be cheaper, but has the advantage that progress can be made on problem domains where a human programmer may not even have a clear idea of what the programming task is, as there is no formal program specification. Instead, a partial description of the desired program's behaviour could be supplied in terms of its input-output behaviour.

Genetic Programming [41, 42], a branch of program synthesis, borrows ideas from the theory of natural evolution to produce programs. The main components of evolutionary computation are *inheritance* (crossover), *selection* and *variation* (mutation). Inheritance implies that the offspring have some resemblance to their parents as almost all of the offspring's genetic material comes from them. Selection means that some offspring are preferable to others, and it is this selection pressure which defines which individuals are fitter then others. Variation supplies fresh genetic material, so individuals containing genetic material which was not present in either of the parents (or the wider gene pool) can be created. Evolutionary computation can be thought of the interaction of these three components.

Informally, a population of computer programs is generated, and the genetically inspired operations of mutation and crossover are applied repeatedly in order to produce new computer programs. These programs are tested against a *fitness function*, that determines which ones are more likely to survive to future generations. The fittest programs are more likely to be selected to continue in the evolutionary process (i.e. survival of the fittest).

More formally, a multiset of computer programs is generated. Programs are transformed by a number of operations, which typically take one or two computer programs as inputs. A fitness function assigns a value to each program (typically depending on its performance on the problem). A selection function generates a new multiset of programs from the previous multiset. This process is repeated until a

termination condition is satisfied. In other words, Genetic Programming is a method of generating syntactically valid programs, according to some predefined grammar, and a fitness function is used to decide which programs are better suited to the task at hand.

In Genetic Programming, the programs that comprise the population are traditionally represented as tree structures. There are other program structures which can be evolved, such as linear sequences of instructions or grammars. We will briefly introduce tree-based Genetic Programming. Each node in the tree returns a value to its parent node. The leaf nodes are usually input variables providing information about the problem state, or numeric constants. The internal nodes have one or more children nodes, and they return a value obtained by performing operations on the values returned by their children nodes. The trees' internal nodes are referred to as *functions*, and leaf nodes are referred to as *terminals*.

A number of decisions needs to be made before a Genetic Programming run is started. This includes the setting of parameters, such as the size of the population, and the termination condition (which is typically the number of generations). It also includes such as the *function set* and *terminal set*, along with the fitness function, which ultimately drives the evolutionary process. The terminal set is the set of nodes that can be the leaf nodes of the program tree, and as such, they take no arguments. They are the mechanism through which the problem state is made available to the program, and they act as input variables, changing their value as the problem state changes. The example of evolving a program to control a robot to clean a floor is given in [42]. The terminals may be defined as the movements that the robot can make, such as 'turn right', 'turn left', and 'move forward'. Other terminals may provide sensory information, such as how far an obstacle is from the robot. On the other hand, the function set is the set of operations that can be represented by the internal nodes of the tree. For example, they can be arithmetic operators, Boolean logic operators, or conditional operators. The functions of a Genetic Programming tree manipulate the values supplied to the program by the terminals, and as such their defining feature is that they take one or more arguments, which can be the values returned by terminal nodes, or other function nodes.

There is an important distinction which can be drawn between an optimisation problem and a learning problem. In the former, we seek the highest quality solution with respect to some evaluation function. An example is the minimisation of a function, where we seek a value of x such that $f(x)$ is a minimum. In the latter, we seek a solution which optimises the value of a target function on the validation data, which is independent of the training data. For example, consider function regression, where we seek a representation of $f(x)$, given a set of training data, but tested on a second set of data to confirm its ability to generalise. Typically, Genetic Programming is used as described in the second example. However, as we shall see, this distinction is apparent when we consider the difference between *reusable* heuristics (which need to be tested on a second set of examples to confirm their status as reusable heuristics), and *disposable* heuristics (which are only used on a single set of examples, without reuse in mind). For more details, see [42].

There are numerous tutorials, introductory articles and text books on Genetic Programming. See the series of books by Koza [38, 39, 40, 41] and the book by Banzhaf et al. [5]. Also [42] and [50] are more recent introductory texts. Introductory articles can also be found in most current textbooks on machine learning.

Genetic Programming can be employed as a hyper-heuristic. It can operate on a set of terminals and functions at the meta-level. Figure 1(a) shows a standard hyper-heuristic framework presented in [9, 17]. Figure 1(b) shows how Genetic Programming might be employed in this capacity. The base-level of a Genetic Programming hyper-heuristic includes the concrete functions and terminals associated with the problem. Across the domain barrier, abstract functions and terminals in the meta-level can be mapped to concrete functions and terminals in the base-level.

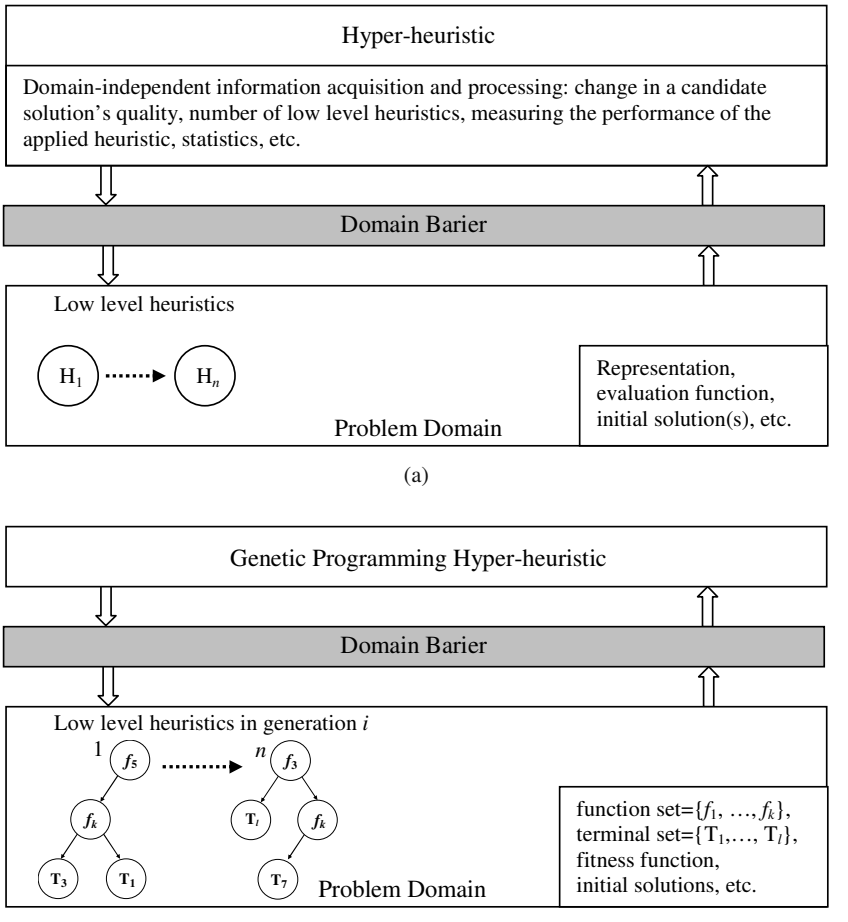

Fig. 1 (a) A generic and (b) a Genetic Programming hyper-heuristic framework

1.4 Chapter Outline

The outline of the remainder of this chapter is as follows. In Section 2, the use of Genetic Programming as a Hyper-heuristic is introduced. In Section 3, two cases studies are examined, namely the applications of Boolean Satisfiability and Online Bin Packing. In Section 4, some of the current literature concerning the automatic generation of heuristics is covered. Section 5 summarises and concludes the chapter.

2 Genetic Programming as a Hyper-heuristic

In this section, we examine a number of issues concerning the use and suitability of Genetic Programming to generate heuristics. A fundamental point concerning the scalability of this method is stated. As this methodology borrows ideas from human designed heuristics, which are then used as primitives to construct the search space of Genetic Programming, we are then in the enviable position of being able to guarantee heuristics which perform at least as good as human designed heuristics. Finally, we outline the basic approach to using Genetic Programming to generate heuristics.

2.1 Suitability of Genetic Programming as a Hyper-heuristic

A number of authors [5, 38, 39, 40, 41, 42] have pointed out the suitability of Genetic Programming over other machine learning methods to automatically produce heuristics. We list these advantages here (in no particular order).

- Genetic Programming has a variable length encoding. Often, we do not know (in advance) the optimal length of representation for heuristics for the given problem domain.
- Genetic Programming produces executable data structures. Heuristics are typically expressed as programs or algorithms.
- Humans can easily identify the good features of the problem domain which form the terminal set of a Genetic Programming approach.
- Human designed heuristics can readily be expressed in the language used to construct the search space of Genetic Programming. A function set, relevant to the problem domain can be determined without too much difficulty. In addition, the Genetic Programming system could also be supplemented with a grammar.

Of course, there are a number of disadvantages of using Genetic Programming to generate heuristics. For example, each time a Genetic Program is run it will give a different "best-of-run" heuristic, so it needs to be run multiple times, in order to gain a feel for the quality of the heuristics which it can produce. Other disadvantages include the, often unintuitive values for parameters, which are typically found through a trial and error process.

2.2 The Basic Approach

Given a problem domain, the application of Genetic Programming to generate heuristics can be undertaken as follows. Many of the steps described here are the same as those one would be required to go through in the construction of a normal Genetic Programming application (e.g. function regression). The main difference, which may not usually be required in a normal application of Genetic Programming, is to decide how the heuristic function is applied to the given problem domain.

1. **Examine currently used heuristics.** Here, we see if currently used heuristics can be described in a common framework, in which each existing heuristic is a special case. These could be either human designed or produced by other machine learning approaches. This step is not trivial and can involve the detailed understanding of the workings of a number of diverse existing heuristics, which may work in very different ways, in order to essentially arrive at the "big picture", or a generalisation of the heuristics used for the problem. Often, these human designed heuristics are the result of years of work by experts, so this process can be difficult.
2. **A framework for the heuristics to operate in.** We are concerned here with the question of how the heuristics are to be applied to an instance of the problem from the given domain. In general, this will be very different depending on the problem domain. It may be the case that many heuristics are applied in the same way, so it may be efficient to apply evolved heuristics in the same fashion. For example, many local search heuristics for the Boolean satisfiability problem fit into the same framework (see [25]).
3. **Decide on the terminal set.** Here, we decide on a set of variables which will express the state of the problem. These will appear as some of the terminals to the Genetic Programming system. Other terminals will also be needed. In particular, random constants are useful.
4. **Decide on the function set.** We need to know how the variables will be connected or composed together. This set of functions will form the function set of the Genetic Programming system. As with the problem of parameter setting (described below), it is worth revisiting this choice as the development progresses.
5. **Identify a fitness function.** A fitness function needs to be defined for the problem. Often, a simple naive fitness function does not perform very well, and introducing some parameters may help find a more suitable one.
6. **Run the Genetic Programming approach.** Often, a Genetic Programming system will not produce good solutions on a first run as poor parameters are chosen. This is especially the case with the novice practitioner. It is therefore essential that different parameter settings are thoroughly investigated.

3 Case Studies

We examine two examples in detail in order to illustrate the basic methodology (generating heuristics for Boolean satisfiability and online bin packing). In both cases,

we describe the problem, a number of currently used human created heuristics, and some design questions about using Genetic Programming to generate heuristics. In the first example, evolving a local search heuristic for the Boolean satisfiability problem, a number of the design decisions (e.g. what variables are needed to express the problem, and a framework in which to express possible heuristics) seem reasonably straightforward, as similar choices were made by two independent authors [3, 25]. In the second example, these choices appear to be a little more difficult. The aim of this section, therefore, is to take the reader step by step through the process and raise a number of issues that will arise during the steps needed to apply Genetic Programming to generate heuristics. These domains have been chosen as they are well known problems, which both have published results concerning the automatic generation of heuristics.

3.1 Boolean Satisfiability – SAT

The Boolean satisfiability problem is the problem of finding the true/false assignments of a set of Boolean variables, to decide if a given propositional formula or expression (in conjunctive normal form) can be satisfied (i.e. does there exist values for the variables which make the expression evaluate to true). The problem is denoted as SAT. It is a classic NP-complete problem [27]. For example, the formula with three clauses $(a \vee b \vee \neg c) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$ is satisfiable as the formula evaluates to true when (a = true, b = false, c = true). However, the formula $a \wedge \neg b \wedge (\neg a \vee b) \wedge (a \vee b \vee c)$ is not satisfiable, as an assignment of the variables, such that the formula is true does not exist. A clause is referred to as *broken*, if all the variables in the clause are evaluated to be false under a given assignment. For example, in the formula $(\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee \neg c)$, there are two broken clauses under the assignment (a = true , b = false, c = true): $(\neg a \vee b \vee \neg c)$ and $(\neg a \vee \neg c)$.

3.1.1 Existing Heuristics

Fukunaga [24] lists a number of well known local search heuristics which have been proposed in the SAT literature.

- *GSAT* selects a variable from the formula with the highest net gain. Ties are decided randomly.
- *HSAT* is the same as GSAT, but it decides ties in favour of maximum age, where age of a variable indicates the total number of bit-flips from the time when a variable was last inverted.
- *GWSAT(p)* (also known as "GSAT with random walk") randomly selects a variable with probability p in a randomly selected broken clause; otherwise, it is the same as GSAT.
- *Walksat(p)* picks a broken clause, and if any variable in the clause has a negative gain of 0, then it selects one of these to be flipped. Otherwise, it selects a random variable with probability p in the clause to flip, and selects a variable with

probability $(1-p)$ in the clause with minimal negative gain (breaking ties randomly). Otherwise, it selects a random variable with probability p in the clause to flip, and selects a variable with probability $(1-p)$ in the clause with minimal negative gain (breaking ties randomly).

Other heuristics, such as, *Novelty*, *Novelty+* and *R-Novelty* are also discussed in [24].

3.1.2 Framework for Heuristics

Fukunaga [24] first examines the original local search heuristic GSAT, and also its many variants (including GSAT with random walk, and Walksat). Then, a template is identified which succinctly describes the most widely used SAT local search algorithms. This framework is also adopted by Bader-El-Den and Poli [3]. In this template, the set of Boolean variables are initially given random truth assignments. Repeatedly, a variable is chosen according to a variable selection heuristic and its value is inverted. This new assignment of values is then tested to see if it satisfies the Boolean expression. This is repeated until some cut off counter is reached. Notice that in this framework, only a single Boolean variable is selected to be inverted. An interesting alternative would be for the variable selection heuristic to return, not a single variable, but a subset of variables.

3.1.3 Identifying the Terminal Set

Fukunaga describes a number of factors in identifying which Boolean variables might be advantageous to invert. Let B_0 be the number of broken clauses in the expression, under the current variable assignment. Let B_1 be the number of broken clauses in the expression, under the current variable assignment, but when variable V is flipped. Let T, be the variable assignment and T^1 be the variable assignment when V is flipped. By looking at the number of clauses that become satisfied or unsatisfied when V is flipped, we can define a number of gain metrics. The net gain of V is $B_1 - B_0$. The negative gain of V is the number of clauses satisfied in T but broken in T^1. The positive gain of V is the number of clauses satisfied in T^1 but broken in T. Another example of a factor which can be used is the "age" of a variable (i.e. the number of inversions from the time when a variable was last inverted). These will form some of the terminals of the Genetic Programming system. For a complete list of terminals see [25].

3.1.4 Identifying the Function Set

Some heuristics are hybrid, in the sense that they are a combination of two existing heuristics. The "composition" (or blending) of two heuristics is achieved by first testing to determine if a condition is true, then if the test is passed apply heuristic1 else apply heuristic2. This composition operator therefore gives us a way to combine already existing heuristics. An example of the testing condition may simply

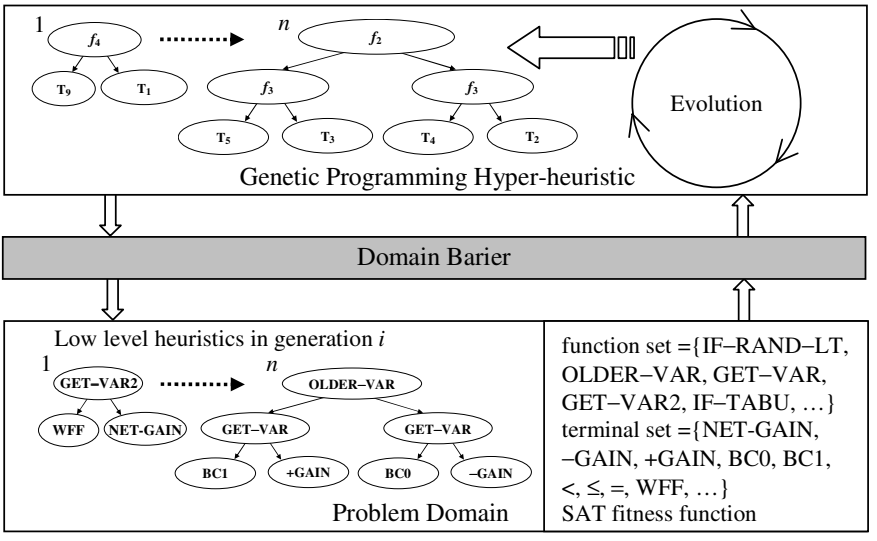

Fig. 2 Genetic Programming as a hyper-heuristic. At the meta-level Genetic Programming refers to abstract functions $\{f_1, f_2, ...\}$, and terminals $\{T_1, T_2, ...\}$. At the base-level these are given concrete meaning. For example, f_1 = IF–RAND–LT , f_2 = OLDER–VAR , T_1 = NET–GAIN, T_2 = –GAIN, etc

be "(random number ≤ 0.2)". Having identified a template for local search and a method of identifying the utility of inverting a given variable, Fukunaga then defined a language in which most of the previously human designed heuristics can be described, but more importantly, it can also be used to describe new novel heuristics. For a complete list of functions see [25].

3.1.5 Identifying a Fitness Function

The fitness function works as follows. First, the heuristic is tested on 200 problem instances consisting of 50 variables and 215 clauses. The heuristic is allowed 5000 inversions of the Boolean variables. If more than 130 of these local searches were successful, then the heuristic is run on 400 problem instances consisting of 100 variables with 430 clauses. The heuristic is allowed 20000 inversions of the Boolean variables. The idea of using smaller and larger problems, is that poor candidate heuristics can be culled early on (very much like brood selection, reported in [5]).

$$\begin{aligned} fitness = \ & (\text{number of 50 variable successes}) \\ & + 5(\text{number of 100 variable successes}) \\ & + 1/(\text{mean number of flips in successful runs}) \end{aligned} \tag{1}$$

The second term carries a weight of 5, as performance on these instances is much more important. In the case of a tie-break, the last term differentiates these

heuristics. It should be noted that the fitness function takes a large number of parameters, and reasonable values for these should be arrived at with a little experimentation.

3.2 Online Bin Packing

The online bin packing problem can be described as the problem of being given a sequence of items and assigning each one to a bin as it arrives, such that the minimum number of bins is required [55]. There is an unlimited supply of bins, each with the same finite capacity which must not be exceeded. We do not know in advance either the sizes of the items, or the total number of items. This is in contrast to the offline version of the problem where the set of items to be packed is available from the start.

3.2.1 Existing Heuristics

A number of examples of heuristics used in the online bin packing problem are described below: In each case, if the item under consideration does not fit into an existing bin, then the item is placed in a new bin.

- *Best-Fit* [51]. Puts the item in the fullest bin which can accommodate it.
- *Worst-Fit* [16]. Puts the item in the emptiest bin which can accommodate it.
- *Almost-Worst-Fit* [16]. Puts the item in the second emptiest bin.
- *Next-Fit* [36]. Puts the item in the last available bin.
- *First-Fit* [36]. Puts the item in the left-most bin.

It should, of course, be noted that this list of heuristics is not exhaustive. The selection is simply intended to illustrate some of the currently available heuristics, and provide a background against which we can build a framework. The reader is referred to the following article if they are particularly interested in the domain of online bin packing . Here the HARMONIC algorithms are discussed (which include HARMONIC, REFINED HARMONIC, MODIFIED HARMONIC, MODIFIED HARMONIC 2, and HARMONIC+1). All of these algorithms are shown to be instances of a general class of algorithm, which they call SUPER HARMONIC.

3.2.2 A Framework for Heuristics

In [10, 11, 12], heuristics are evolved for the online bin packing problem. In the first paper [10], a number of existing heuristics are listed. Interestingly these heuristics do not fit neatly into a single framework. In this paper, the decision was made to apply the evolved heuristic to the bins and place the current item under consideration, into the first bin which receives a positive score. Using this method of applying heuristics to problem instances, a heuristic equivalent to the "first-fit" heuristic was evolved. The "first-fit" heuristic places an item in the first bin into which it fits (the order of the bins being the order in which they were opened). In this framework, the

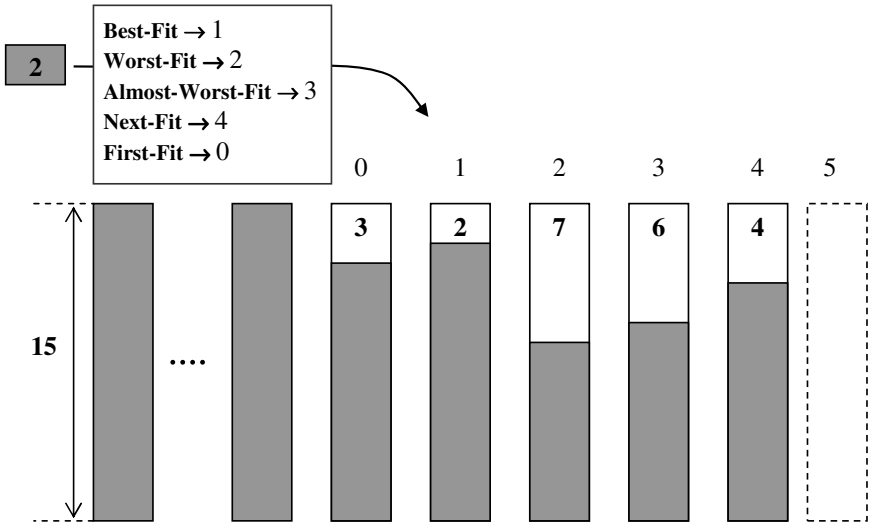

Fig. 3 The figure shows the chosen bin for a number of heuristics. The bin capacity is 15, and the space remaining in the open bins (in order of index 0, 1, 2, 3, 4, 5) is 3, 2, 7, 6, 4, 15. The current item to be packed has size 2. "Best-fit", for example would place the item in bin 1, leaving no space remaining. "First-fit", for example would place the item in bin 0, leaving 1 unit of space

```
For each item
  int binIndex := 0
  For each bin b in A
    output := evaluate Heuristic
    If (output > 0 )
      return binIndex
    End If
  End For
  place item in bin binIndex
End For
```

Fig. 4 An item is considered for each bin in turn, until a positive score is obtained. Thus the heuristic may not be evaluated on all bins, for a given item. The item is placed in the bin which gives the first positive score. This method of applying heuristics differs fundamentally from the method described in figure 5

heuristic may not be evaluated on all of the bins when an item is being placed (i.e. only the bins up until the bin that receives a positive score will be considered).

In [11], it was decided that the heuristic would be evaluated on all the open bins, and the item placed in the bin that receives the maximum score. This has the advantage that the heuristic is allowed to examine all of the bins (and, therefore, has more information available to it to make a potentially better decision). It also has the

```
For each item
  int currentMaximumScore = -∞
  int binIndex := 0
  For each bin b
    output := evaluate Heuristic
    If (output > currentMaximumScore )
      currentMaximumScore = output
      binIndex := b
    End If
  End For
  place item p in bin binIndex
return binIndex
```

Fig. 5 In this framework, the item is placed in the bin which gives the maximum score according to the heuristic. This method of applying heuristics differs fundamentally from the method described in figure 4

disadvantage that it will take longer on average to apply, as it will, in general, examine more bins (though this aspect of the evolved heuristic's performance was not studied). In this framework, heuristics were evolved which outperformed the human designed heuristic "best-fit". The "best-fit" heuristic places an item in the bin which has the least space remaining when the item is placed in the bin (i.e. it fits best in that bin).

The two search spaces created by these frameworks are very different. In the first case, the "first-fit" heuristic can be expressed, but "best-fit" cannot. In the second case, the "best-fit" heuristics can be expressed but "first-fit" cannot. The first framework cannot express "best-fit", as not all of the bins may be examined. That is, the evaluation of the heuristic is terminated as soon as a positive score is obtained. The second framework cannot express "first-fit" as a bin which receives a larger score may exist after one which receives a positive score. That is, an earlier bin may receive a smaller positive score, but this is overridden when a larger score is obtained. Further effort could be put into constructing a more general framework in which both of these heuristics could be expressed.

So far, just two frameworks have been considered which could be used to apply heuristics to the online bin packing problem. There are many different ways a heuristic could be applied.

- They can differ in the order in which the bins are examined. For example, left to right, right to left, or some other order.
- They can differ in the order we start to examine the bins. For example, start at a random bin and cycle through the bins until each bin has been examined, or start at some distance from the last bin that received an item.
- They can differ in the score used to decide which bin is employed. For example, place the item in the bin which got the second highest score, or alternatively place the item in the bin which gets the maximum then the next item in the bin that gets the minimum score; in effect we are switching between two placement strategies.

There is also the question of where to place an item when there is a draw between two bins (e.g. the item could be placed in a fresh bin, or it could be placed in a bin using an existing human designed heuristic). The point is that there are plenty of opportunities to design different ways of applying heuristic selection functions to a problem domain. Therefore, instead of presenting Genetic Programming with a single framework, it is possible to widen this and allow a number (or combination) of different frameworks for Genetic Programming to explore. One interesting way to tackle this would be to cooperatively co-evolve a population of heuristics and the frameworks in which they are applied.

It is also worthwhile pointing out that a heuristic evolved under one framework is unlikely to perform well under another framework, so a heuristic really consists of two parts; the heuristic function and the framework describing how the heuristic is applied to a problem instance. In Genetic Programming, we are usually just interested in the function represented by a program, and the program does not need a context (e.g. in the case of evolving electrical circuits, the program is the solution). However, if we are evolving heuristics, we need to provide a context or method of applying the Genetic Programming-program. This additional stage introduces a considerable difference.

3.2.3 Identifying the Terminal Set

The question of which variables to use to describe the state of a problem instance is also important, as these will form some of the "terminals" used in Genetic Programming. In the first stages of this work, the authors used the following variables; S the size of the current item, C the capacity of a bin (this is a constant for the problem) and, F the fullness of a bin (i.e. what is the total cost of all of the items occupying that bin).

It was later decided that three variables could be replaced by two; S the size of the current item and, E $(= C - F)$ the emptiness of a bin (i.e. how much space is remaining in the bin, or how much more cost can be allocated to it before it exceeds its capacity). These two variables are not as expressive as the first set, but are expressive enough to produce human competitive heuristics. The argument is that it is not the capacity or fullness of a bin which is the deciding factor of whether to put an item in a bin, but the remaining capacity, or emptiness E, of a bin. In fact, the capacity of a bin was fixed for the entire set of experiments, so could be considered as a constant. In other words, the output of the heuristic based on this pair of variables, could be semantically interpreted as how suitable is the current bin, given its emptiness, and the size of the item we would like to place in the bin.

This pair of variables can be replaced by a single variable, R $(= E - S)$ the space remaining in a bin if the current item were placed in the bin. The output of a heuristic based solely on this single variable could not be interpreted as in the previous case, but rather as the following question: If the current item were placed in the current bin, what is the utility of the remaining space?

So far, we have only considered variables describing the current item and current bin. However, there are other variables which could be introduced. Other examples of variables which could be stored are

- the item number (i.e. keep a counter of how many items have been packed so far)
- the minimum and maximum item size seen so far (as more items are packed, these bounds will diverge)
- the average and standard deviation of item size seen so far (these could provide a valuable source of statistical information on which to base future decisions).

All of this information can be made available to the evolving heuristic.

3.2.4 Identifying the Function Set

In [10], the function set $\{+, -, x, \%, abs, \leq\}$ was used, where *abs* is the absolute operator and % is protected divide. There are a few points worth considering with this chosen function set. Firstly, $\leq$ returns -1 or +1, rather than 0 or 1, which is normally associated with this relational operator. This was to satisfy the property of closure, that the output of any function in the function set, can be used as the input of any function in the function set. Secondly, this function set is sufficient to express some the human designed heuristics described (namely "first-fit" and "best-fit").

Protected divide (%) is often used in Genetic Programming, as if the denominator is zero, then the function is undefined (i.e. its value tends to infinity). Typically, protected divide returns 0 or 1. However, this choice does not reflect the idea that the quotient could be a very large number. Thus, in [11], a much larger value was returned.

In [12], $\leq$ was removed from the function set as it was effectively redundant. This is because, as the evolved heuristic function is enveloped in a loop which returns the index of the maximum scoring bin, any test for 'less than' can be done by the loop. The aim of this discussion is to outline the difficulty in choosing a function set for the given problem domain.

3.2.5 Identifying a Fitness Function

The fitness function to determine the quality of a solution is shown in Equation 2 [22], where n = number of bins used, F_i = fullness of bin i, and C = bin capacity

$$Fitness := \begin{cases} high\ penalty\ value, & \text{if illegal solution} \\ 1 - \left(\frac{\sum_{i=1}^{n} (F_i/C)^k}{n} \right), & \text{if legal solution} \end{cases} \quad (2)$$

It returns a value between 0 and 1, with 0 being the best result where all bins are filled completely, and 1 representing completely empty bins.

In the bin packing problem, there are many different solutions which use the same number of bins. If the fitness function were simply the number of bins used, then there would be a plateau in the search space that is easily reached, but difficult to escape from [22]. Using equation 2 as a fitness function helps the evolutionary process by

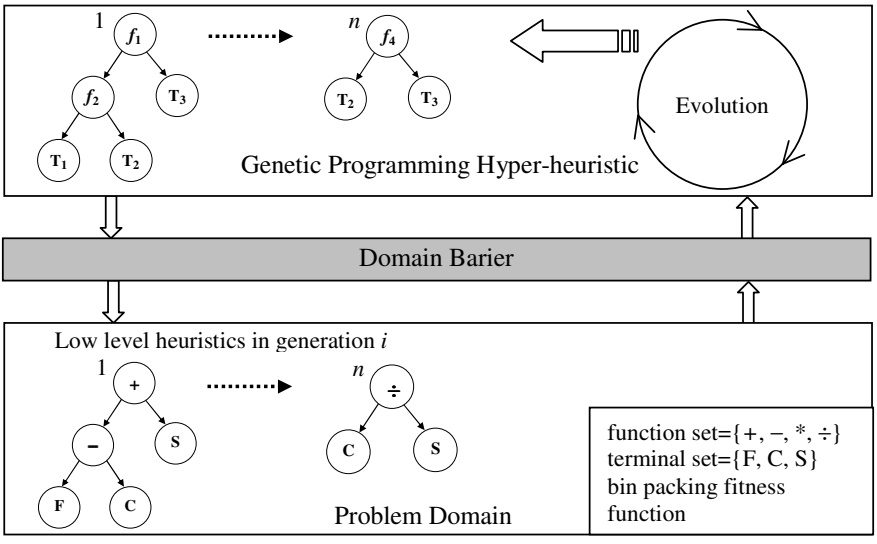

Fig. 6 Genetic Programming as a hyper-heuristic. At the meta-level Genetic Programming refers to abstract functions $\{f_1, f_2, ...\}$, and terminals $\{T_1, T_2, ...\}$. At the base-level these are given concrete meaning. For example, $f_1 = +$, $f_2 = -$, $T_1 = \text{F}$, $T_2 = \text{C}$, etc

differentiating between two solutions that use the same number of bins. The fitness function proposed by Falkenauer rewards solutions more if some bins are nearly full and others nearly empty, as opposed to all the bins being similarly filled.

The constant k in equation 2 determines how much of a premium is placed on nearly full bins. The higher the value of k, the more attention will be given to the almost filled bins at the expense of the more empty ones. A value of $k = 2$ was deemed to be the best in [22] so this is the value we use here.

4 Literature Review

In this section, we briefly discuss the area, in order to give the reader a flavour of what has been attempted to date. We include some work specifically using Genetic Programming as a hyper-heuristic. We also include some work on other areas which are similar in the sense that they use a meta-level in the learning system, and can tackle multiple problems. We now briefly review two areas of the machine learning literature which could also be considered in the context of hyper-heuristics. The first is learning to learn, and the second is self-adaptation.

4.1 Genetic Programming Hyper-heuristics for Generating Reusable Heuristics

Keller et al. [37] use a grammar to describe promising meta-heuristics for the travelling salesman problem. Primitives in the grammar may represent manually created

meta-heuristics, low level heuristics, or component parts of them. There are a number of heuristics used in this system, including heuristics which swap two or three edges in the solution, and an iterative heuristic which executes another heuristic a maximum of 1000 times unless an improvement is seen. The execution of the meta-heuristic is a sequential execution of a list of heuristics and so generates a candidate solution to the given problem from a random initial route. Tours whose lengths are highly competitive with the best real-valued lengths from the literature are found using this grammar based Genetic Programming.

In a series of papers, Burke et al. [10, 11, 12] examine the viability of using Genetic Programming to evolve heuristics for the online bin packing problem. Given a sequence of items, each must be placed into a bin in the order it arrived. At each decision point, we are only given the size of the current item to be packed. In [10], an item is placed into the first bin which receives a positive score according to the evolved heuristic. Thus, the heuristic may not be evaluated for all bins, as it is terminated as soon as a positive score is obtained. This approach produces a heuristic which performs better than the human designed "first-fit" heuristic.

In [11], a similar approach is used. However, this time, the heuristic is allowed to examine all bins, and the item is placed in the bin which receives the maximum score. This produces a heuristic which is competitive with the human designed heuristic "best-fit". The difference between these two approaches, illustrates that the framework to evaluate the heuristics is a critical component of the overall system. In [11], the performance of heuristics on general and specialised problem classes is examined. They show that, as one problem class is more general than another, then the heuristic evolved on the more general class is more robust, but performs less well than the specialised heuristic on the specialised class of problem. This is, intuitively, what one would expect.

In [12], evolved heuristics are applied to much larger problem instances than they were trained on, but as the larger instances come from the same class as the smaller training instances, performance does not deteriorate and indeed, the approach consistently outperforms the human designed best-fit heuristic. The paper makes the important distinction between the nature of search spaces associated with direct and indirect methods. With direct methods, the size of the solution necessarily grows with the size of the problem instance, resulting in combinatorial explosion, for example, when the search space is a permutation. However, when the search space consists of programs or heuristics, the size of a program to solve a given class of problem is fixed as it is a generalisation of the solution to a class of problem (i.e. the solution to a class of problem is independent of the size of an instance).

Drechsler et al. [19], instead of directly evolving a solution, use Genetic Programming to develop a heuristic that is applied to the problem instance. Thus the typically large run-times associated with evolutionary runs have to be invested only once in the learning phase. The technique is applied to a problem of minimising Binary Decision Diagrams. They state that standard evolutionary techniques cannot be applied due to their large runtime. The best known algorithms used for variable ordering are exponential in time, thus heuristics are used. The heuristics which are developed by the designer often fail for specific classes of circuits. Thus it would

be beneficial if the heuristics could learn from previous examples. An earlier paper is referred to where heuristics are learnt using a genetic algorithm [20], but it is pointed out that there are problems using a fixed length encoding to represent heuristics. Experiments show that high quality results are obtained that outperform previous methods, while keeping low run-times.

Fukunaga [24, 25] examines the problem domain of Boolean satisfiability (SAT). He shows that a number of well-known local search algorithms are combinations of variable selection primitives, and he introduces CLASS (Composite heuristic Learning Algorithm for SAT Search), an automated heuristic discovery system which generates variable selection heuristic functions. The learning task, therefore, is designing a variable selection heuristic as a meta-level optimisation problem.

Most of the standard SAT local search procedures can be described using the same template, which repeatedly chooses a variable to invert, and calculates the utility in doing so. Fukunaga identifies a number of common primitives used in human designed heuristics e.g. the gain of flipping a variable (i.e. the increase in the number of clauses in the formula) or the age of a variable (i.e. how long since it was last flipped). He states that *"it appears human researchers can readily identify interesting primitives that are relevant to variable selection, the task of combining these primitives into composite variable selection heuristics may benefit from automation"*. This, of course, is particularly relevant for Genetic Programming.

In the CLASS language, which was shown to able to express human designed heuristics, a composition operator is used which takes two heuristics and combines them using a conditional if statement. The intuition behind this operator is that the resulting heuristic blends the behaviour of the two component heuristics. The importance of this composition operator is that it maintains the convergence properties of the individual heuristics, which is not true if Genetic Programming operators were used. CLASS successfully generates a new variable selection heuristic, which is competitive with the best-known GSAT/Walksat-based algorithms. All three learnt heuristics were shown to scale and generalise well on larger random instances; generalisation to other problem classes varied.

Geiger et al. [28] present an innovative approach called SCRUPLES (scheduling rule discovery and parallel learning system) which is capable of automatically discovering effective dispatching rules. The claim is made that this is a significant step beyond current applications of artificial intelligence to production scheduling, which are mainly based on learning to select a given rule from among a number of candidates rather than identifying new and potentially more effective rules. The rules discovered are competitive with those in the literature. They state that a review of the literature shows no existing work where priority dispatching rules are discovered through search. They employ Genetic Programming, as each dispatching rule is viewed as a program. They point out that, Genetic Programming has a key advantage over more conventional techniques such as genetic algorithms and neural networks, which deal with fixed sized data structures. Whereas Genetic Programming can discover rules of varying length and for many problems of interest, such as scheduling problems, the complexity of an algorithm which will produce the

correct solution is not known a-priori. The learning system has the ability to learn the best dispatching rule to solve the single unit capacity machine-scheduling problem. For the cases where no dispatching rules produced optimal solutions, the learning system discovers rules that perform no worse than the known rules.

Stephenson et al. [57] apply Genetic Programming to optimise priority or cost functions associated with two compiler heuristics; predicted hyper block formation (i.e. branch removal via prediction) and register allocation. Put simply, priority functions prioritise the options available to a compiler algorithm. Stephenson et al. [57] state "*Genetic Programming is eminently suited to optimising priority functions because they are best represented as executable expressions*". A caching strategy, is a priority function that determines which program memory locations to assign to cache, in order to minimise the number of times the main memory must be accessed. The human designed "least recently used" priority function is outperformed by results obtained by Genetic Programming. They make the point that by evolving compiler heuristics, and not the applications themselves, we need only apply our process once, which is in contrast to an approach using genetic algorithms. In addition they emphasise that compiler writers have to tediously fine tune priority functions to achieve suitable performance, whereas with this method, this is effectively automated.

4.2 Genetic Programming Hyper-heuristics for Generating Disposable Heuristics

Bader-El-Din et al. [3] present a Genetic Programming hyper-heuristic framework for the 3-SAT problem domain. Their aim is not to evolve reusable heuristics, but to solve a set of problem instances. The evolved heuristics are essentially disposed of and are considered as a by-product of the evolutionary process. Human designed heuristics are broken down into their constituent parts, and a grammar is used to capture the structure of how the constituents relate to each other. The constituent parts, along with the grammar, are used to construct a search space, which contains (by definition) the human designed heuristics. The resulting space is searched using Genetic Programming. Although the initial population of heuristics were randomly generated and included no handcrafted heuristics as primitives, individuals representing such heuristics were created in the initial population in almost all experiments (i.e. heuristics equivalent to human designed heuristics were found by random search). This is due to their simple representation in the grammar defined in the system.

4.3 Learning to Learn

Learning to learn [60] is similar to using Genetic Programming as a hyper-heuristic to solve a class of problem. Rather than trying to learn a single instance of a problem, a class of related problems is tackled. The key idea is to have two explicit levels

in the learning algorithm, a meta-level and a base-level. The base-level is associated with learning a function, just like regular supervised learning in the single task case. The meta-level is responsible for learning properties of these functions (i.e. invariants or similarities between the problem instances). Thus, the meta-level is responsible for learning across the distribution of problems. Any machine-learning paradigm could be used at the base-level or meta-level.

4.4 The Teacher System

An interesting related project at the interface between machine learning and engineering was termed "Teacher" [62, 63] (an acronym for TEchniques for the Automated Creation of HEuRistics), which was designed as a system for learning and generalising heuristics used in problem solving. The objective was to find, under resource constraints, improved heuristic methods as compared to existing ones, in applications with little (or non-existent) domain knowledge. The Teacher system employed a genetic-based machine learning approach, and was successfully applied to several domains such as: process mapping, load balancing on a network of workstations, circuit placement, routing and testing. The system addressed five important general issues in learning heuristics [62]: "(1) *decomposition* of a problem solver into smaller components and integration of heuristic methods designed for each smaller component; (2) *classification* of an application domain into subdomains so that the performance can be evaluated statistically for each; (3) *generation* of new and improved heuristic methods based on past performance information and heuristics generated; (4) *evaluation* of each heuristic method's performance; and (5) *performance generalization* to find heuristic methods that perform well across the entire application domain".

4.5 Related Areas

Heuristic search represents a major research activity at the interface of Operational Research and Artificial Intelligence. It provides the core engine for real-world applications as diverse as timetabling, planning, personnel and production scheduling, cutting and packing, space allocation, and protein folding. Several researchers have recognised that a promising direction for developing improved and automated search techniques is to integrate learning components that can adaptively guide the search. Many techniques have independently arisen in recent years that exploit either some form of learning, or search on a search space of algorithm configuration, to improve problem-solving and decision making. A detailed review of these techniques is beyond the scope of this chapter. However, we mention here some related areas of research: adaptation and self-adaptation of algorithm parameters [2, 21, 32], algorithm configuration [33], racing algorithms [8], reactive search [6, 7], adaptive memetic algorithms [34, 35, 43, 44, 47, 48, 56], and algorithm portfolios [26, 31].

5 Summary and Conclusion

Often, in the field of computational search, a single problem (sometimes even a single instance) is tackled. This chapter describes work which is motivated by the goal of moving away from this situation. The work described here is attempting to "raise the level of generality". This offers a number of long term advantages. In particular, we can obtain more general systems rather than problem specific approaches. Moreover, we can achieve this more cheaply in terms of resource(s) used; i.e. a computer system is much cheaper to run than a team of heuristic designers is to employ.

5.1 The Need for Automatic Heuristic Generation

Real-world intractable problems demand the use of heuristics if progress is to be made in reasonable time. Therefore, the practical importance of heuristics is unquestionable, and how heuristics are produced then becomes an important scientific question. Many of the current heuristics in use today are the result of years of study by experts with specialist knowledge of the domain area. Therefore, one may pose the question;

Instead of getting experts to design heuristics, perhaps they would be better employed designing a search space of heuristics (i.e. all possible heuristics or a promising subset of heuristics) and a framework in which the heuristics operate, and letting a computer take over the task of searching for the best ones.

This approach shows a clear division of labour; Humans, taking on the innovative and creative task of defining a search space. Computers take on the chore of searching this vast space. Due to the fact that humans often still need to play an important part in this process, we should strictly refer to this methodology as a *semi*-automated process.

One of the advantages of this methodology is that if the problem specification were to change, the experts who engage in hand designing heuristics, would probably have to return to the drawing board, possibly approaching the problem from scratch again. This would also be the situation with the search for automatically designed heuristics, with one important difference. As the search process is automated this would largely reduce the cost of having to create a new set of heuristics. In essence, by employing a method automated at the meta-level, the system could be designed to tune itself to the new problem class presented to it.

A paradigm shift has started to occur in search methodologies over the past few years. Instead of taking the rather short term approach of tackling single problems, there is a growing body of work which is adopting the more long term approach of tackling the general problem, and providing a more general solution.

5.2 Supplementing Human Designed Heuristics

Typically in the "meta-heuristics to choose heuristics" framework, the heuristics are human designed, and therefore have all the strengths of human designed heuristics,

but also all of the weaknesses. In contrast, machine generated heuristics will have their own strengths and weaknesses. Thus, as one of the motives of hyper-heuristics is to combine heuristics, this would offer a method where manually and automatically designed heuristics can be used side by side. It may also be possible to evolve heuristics specifically to complement human designed heuristics in a hyper-heuristic context, where an individual heuristic does not need to be good on its own, but is a good team player in the environment of the other heuristics. Again this is another example of cooperation between humans and computers.

References

1. Asmuni, H., Burke, E.K., Garibaldi, J.M., Mccollum, B.: A novel fuzzy approach to evaluate the quality of examination timetabling. In: Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling, pp. 82–102 (2006)
2. Bäck, T.: An overview of parameter control methods by self-adaption in evolutionary algorithms. Fundam. Inf. 35(1-4), 51–66 (1998)
3. Bader-El-Din, M.B., Poli, R.: Generating SAT local-search heuristics using a GP hyper-heuristic framework. In: Monmarché, N., Talbi, E.-G., Collet, P., Schoenauer, M., Lutton, E. (eds.) EA 2007. LNCS, vol. 4926, pp. 37–49. Springer, Heidelberg (2008)
4. Bai, R., Kendall, G.: An investigation of automated planograms using a simulated annealing based hyper-heuristic. In: Ibaraki, T., Nonobe, K., Yagiura, M. (eds.) Metaheuristics: Progress as Real Problem Solver. Operations Research/Computer Science Interface Serices, vol. 32, pp. 87–108. Springer, Heidelberg (2005)
5. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, San Francisco (1998)
6. Battiti, R.: Reactive search: Toward self–tuning heuristics. In: Rayward-Smith, V.J., Osman, I.H., Reeves, C.R., Smith, G.D. (eds.) Modern Heuristic Search Methods, pp. 61–83. John Wiley & Sons Ltd, Chichester (1996)
7. Battiti, R., Brunato, M.: Reactive search: Machine learning for memory-based heuristics. In: Gonzalez, T.F. (ed.) Approximation Algorithms and Metaheuristics, ch. 21, pp. 1–17. Taylor and Francis Books/CRC Press, Washington (2007)
8. Birattari, M.: The problem of tuning metaheuristics as seen from a machine learning perspective. Ph.D. thesis, Universite Libre de Bruxelles (2004)
9. Burke, E.K., Hart, E., Kendall, G., Newall, J., Ross, P., Schulenburg, S.: Hyper-heuristics: An emerging direction in modern search technology. In: Glover, F., Kochenberger, G. (eds.) Handbook of Metaheuristics, pp. 457–474. Kluwer, Dordrecht (2003)
10. Burke, E.K., Hyde, M.R., Kendall, G.: Evolving bin packing heuristics with genetic programming. In: Runarsson, T.P., Beyer, H.-G., Burke, E.K., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN 2006. LNCS, vol. 4193, pp. 860–869. Springer, Heidelberg (2006)
11. Burke, E.K., Hyde, M.R., Kendall, G., Woodward, J.: Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In: Thierens, D., et al. (eds.) Proceedings of the 9th annual conference on Genetic and evolutionary computation GECCO 2007, vol. 2, pp. 1559–1565. ACM Press, London (2007)
12. Burke, E.K., Hyde, M.R., Kendall, G., Woodward, J.R.: The scalability of evolved on line bin packing heuristics. In: Srinivasan, D., Wang, L. (eds.) 2007 IEEE Congress on Evolutionary Computation, pp. 2530–2537. IEEE Computational Intelligence Society/IEEE Press, Singapore (2007)

13. Burke, E.K., Kendall, G., Soubeiga, E.: A tabu-search hyperheuristic for timetabling and rostering. J. of Heuristics 9(6), 451–470 (2003)
14. Burke, E.K., McCollum, B., Meisels, A., Petrovic, S., Qu, R.: A graph-based hyper-heuristic for educational timetabling problems. Eur. J. of Oper. Res. 176, 177–192 (2007)
15. Burke, E.K., Petrovic, S., Qu, R.: Case based heuristic selection for timetabling problems. J. of Sched. 9(2), 115–132 (2006)
16. Coffman Jr., E.G., Galambos, G., Martello, S., Vigo, D.: Bin packing approximation algorithms: Combinatorial analysis. In: Du, D.Z., Pardalos, P.M. (eds.) Handbook of Combinatorial Optimization, pp. 151–207. Kluwer, Dordrecht (1998)
17. Cowling, P., Kendall, G., Soubeiga, E.: A hyperheuristic approach to scheduling a sales summit. In: Burke, E., Erben, W. (eds.) PATAT 2000. LNCS, vol. 2079, pp. 176–190. Springer, Heidelberg (2001) (selected papers)
18. Dowsland, K.A., Soubeiga, E., Burke, E.K.: A simulated annealing hyper-heuristic for determining shipper sizes. Eur. J. of Oper. Res. 179(3), 759–774 (2007)
19. Drechsler, N., Schmiedle, F., Grosse, D., Drechsler, R.: Heuristic learning based on genetic programming. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 1–10. Springer, Heidelberg (2001)
20. Drechsler, R., Becker, B.: Learning heuristics by genetic algorithms. In: ASP-DAC 1995: Proceedings of the 1995 conference on Asia Pacific design automation (CD-ROM), p. 53. ACM, New York (1995)
21. Eiben, A.E., Hinterding, R., Michalewicz, Z.: Parameter control in Evolutionary Algorithms. IEEE Trans. on Evol. Comput. 3(2), 124–141 (1999)
22. Falkenauer, E., Delchambre, A.: A genetic algorithm for bin packing and line balancing. In: Proc. of the IEEE 1992 International Conference on Robotics and Automation, pp. 1186–1192 (1992)
23. Fang, H.L., Ross, P., Corne, D.: A promising hybrid GA/heuristic approach for open-shop scheduling problems. In: Eur. Conference on Artificial Intelligence (ECAI 2004), pp. 590–594 (1994)
24. Fukunaga, A.S.: Automated discovery of composite sat variable-selection heuristics. In: AAAI/IAAI, pp. 641–648 (2002)
25. Fukunaga, A.S.: Automated discovery of local search heuristics for satisfiability testing. Evol. Comput. 16(1), 31–61 (2008)
26. Gagliolo, M., Schmidhuber, J.: Learning dynamic algorithm portfolios. Ann. of Math. and Artif. Intell. 47(3-4), 295–328 (2006)
27. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Series of Books in the Mathematical Sciences. W. H. Freeman, New York (1979)
28. Geiger, C.D., Uzsoy, R., Aytug, H.: Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. J. of Sched. 9(1), 7–34 (2006)
29. Hart, E., Ross, P., Nelson, J.: Solving a real-world problem using an evolving heuristically driven schedule builder. Evol. Comput. 6(1), 61–80 (1998)
30. Hoos, H.H., Sttzle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann / Elsevier (2005)
31. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Sci. 275, 51–54 (1997)
32. Hutter, F., Hamadi, Y., Hoos, H.H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 213–228. Springer, Heidelberg (2006)

33. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: AAAI, pp. 1152–1157. AAAI Press, Menlo Park (2007)
34. Jakob, W.: HyGLEAM – an approach to generally applicable hybridization of evolutionary algorithms. In: Guervós, J.J.M., Adamidis, P.A., Beyer, H.-G., Fernández-Villacañas, J.-L., Schwefel, H.-P. (eds.) PPSN 2002. LNCS, vol. 2439, pp. 527–536. Springer, Heidelberg (2002)
35. Jakob, W.: Towards an adaptive multimeme algorithm for parameter optimisation suiting the engineers' needs. In: Runarsson, T.P., Beyer, H.-G., Burke, E.K., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN 2006. LNCS, vol. 4193, pp. 132–141. Springer, Heidelberg (2006)
36. Johnson, D., Demers, A., Ullman, J., Garey, M., Graham, R.: Worst-case performance bounds for simple one-dimensional packaging algorithms. SIAM J. on Comput. 3(4), 299–325 (1974)
37. Keller, R.E., Poli, R.: Linear genetic programming of parsimonious metaheuristics. In: Srinivasan, D., Wang, L. (eds.) 2007 IEEE Congress on Evolutionary Computation, pp. 4508–4515. IEEE Computational Intelligence Society/IEEE Press, Singapore (2007)
38. Koza, J.R.: Genetic Programming: on the Programming of Computers by Means of Natural Selection. The MIT Press, Boston (1992)
39. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge (1994)
40. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic Programming III: Darwinian Invention and Problem solving. Morgan Kaufmann, San Francisco (1999)
41. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence (Genetic Programming). Springer, Heidelberg (2003)
42. Koza, J.R., Poli, R.: Genetic programming. In: Burke, E.K., Kendall, G. (eds.) Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, pp. 127–164. Springer, Boston (2005)
43. Krasnogor, N., Gustafson, S.: A study on the use of 'self-generation' in memetic algorithms. Nat. Comput. 3(1), 53–76 (2004)
44. Krasnogor, N., Smith, J.E.: Emergence of profitable search strategies based on a simple inheritance mechanism. In: Proceedings of the 2001 Genetic and Evolutionary Computation Conference, pp. 432–439. Morgan Kaufmann, San Francisco (2001)
45. Mockus, J.: Application of bayesian approach to numerical methods of global and stochastic optimization. J. of Glob. Optim. 4(4), 347–366 (1994)
46. Nareyek, A.: Choosing search heuristics by non-stationary reinforcement learning. In: Resende, M.G.C., de Sousa, J.P. (eds.) Metaheuristics: Computer Decision-Making, ch. 9, pp. 523–544. Kluwer, Dordrecht (2003)
47. Ong, Y.S., Keane, A.J.: Meta-lamarckian learning in memetic algorithms. IEEE Trans. on Evol. Comput. 8, 99–110 (2004)
48. Ong, Y.S., Lim, M.H., Zhu, N., Wong, K.W.: Classification of adaptive memetic algorithms: a comparative study. IEEE Trans. on Syst. Man and Cybern. Part B 36(1), 141–152 (2006)
49. Ozcan, E., Bilgin, B., Korkmaz, E.E.: A comprehensive survey of hyperheuristics. Intell. Data Anal. 12(1), 3–23 (2008)
50. Poli, W.B.R., Langdon, N.F.M.: A Field Guide to Genetic Programming. Lulu Enterprises, UK (2008)
51. Rhee, W.T., Talagrand, M.: On line bin packing with items of random size. Math. Oper. Res. 18, 438–445 (1993)

52. Ropke, S., Pisinger, D.: A unified heuristic for a large class of vehicle routing problems with backhauls. Eur. J. of Oper. Res. 171(3), 750–775 (2006)
53. Ross, P.: Hyper-heuristics. In: Burke, E.K., Kendall, G. (eds.) Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, ch. 17, pp. 529–556. Springer, Heidelberg (2005)
54. Ross, P., Marin-Blazquez, J.G., Schulenburg, S., Hart, E.: Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyper-heuristics. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2003, pp. 1295–1306. Springer, Heidelberg (2003)
55. Seiden, S.S.: On the online bin packing problem. J. ACM 49(5), 640–671 (2002)
56. Smith, J.E.: Co-evolving memetic algorithms: A review and progress report. IEEE Trans. in Syst. Man and Cybern. Part B 37(1), 6–17 (2007)
57. Stephenson, M., O'Reilly, U., Martin, M., Amarasinghe, S.: Genetic programming applied to compiler heuristic optimisation. In: Proceedings of the Eur. Conference on Genetic Programming, pp. 245–257. Springer, Heidelberg (2003)
58. Terashima-Marin, H., Flores-Alvarez, E.J., Ross, P.: Hyper-heuristics and classifier systems for solving 2D-regular cutting stock problems. In: Beyer, H.G., O'Reilly, U.M. (eds.) Proceedings of Genetic and Evolutionary Computation Conference, GECCO 2005, Washington DC, USA, June 25-29, pp. 637–643. ACM, New York (2005)
59. Terashima-Marin, H., Ross, P., Valenzuela-Rendon, M.: Evolution of constraint satisfaction strategies in examination timetabling. In: Proc. of the Genetic and Evolutionary Computation Conf. GECCO 1999, pp. 635–642. Morgan Kaufmann, San Francisco (1999)
60. Thrun, S., Pratt, L.: Learning to learn: Introduction and overview. In: Thrun, S., Pratt, L. (eds.) Learning To Learn. Kluwer Academic Publishers, Dordrecht (1998)
61. Vazquez-Rodriguez, J.A., Petrovic, S., Salhi, A.: A combined meta-heuristic with hyper-heuristic approach to the scheduling of the hybrid flow shop with sequence dependent setup times and uniform machines. In: Proceedings of the 3rd Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA 2007), pp. 506–513 (2007)
62. Wah, B.W., Ieumwananonthachai, A.: Teacher: A genetics-based system for learning and for generalizing heuristics. In: Yao, X. (ed.) Evol. Comput., pp. 124–170. World Scientific Publishing Co. Pte. Ltd, Singapore (1999)
63. Wah, B.W., Ieumwananonthachai, A., Chu, L.C., Aizawa, A.: Genetics-based learning of new heuristics: Rational scheduling of experiments and generalization. IEEE Trans. on Knowl. and Data Eng. 7(5), 763–785 (1995)