Chapter 20
Hyper-heuristics

Peter Ross

20.1 Introduction

Many practical problems are awkward to solve computationally. Whether you are trying to find any solution at all, or perhaps to find a solution that is optimal or close to optimal according to some criteria, exact methods can be unfeasibly expensive. In such cases it is common to resort to heuristic methods, which are typically derived from experience but are inexact or incomplete. For example, in packing and cutting problems a very simple heuristic might be to try to pack the items in some standardized way starting with the largest remaining one first, on the reasonable grounds that the big ones tend to cause the most trouble. But such heuristics can easily lead to suboptimal answers.

Suppose you are given a supply of 6×6 sheets of metal and have to cut them to produce forty 5×1 sheets and ten larger, 4×4, sheets (you are not restricted to guillotine cuts, that is, to edge-to-edge straight cuts). The total area of the required pieces is 360, which suggests that at least 10 sheets are needed. If you start with the biggest ones, by cutting out a 4×4 sheet from a corner, you are doomed to waste material; the sole optimal arrangement per sheet (apart from reflection) is shown in Fig. 20.1, which wastes no material.

However, it is not easy to conceive of a straightforward and broadly useful heuristic that would suggest such a layout. Creating new heuristics, whether by detailed study of sample problems and their solutions or by personal introspection based on past experience, is unreliable and difficult (see for example Neth et al. 2009). More generally, both intuition and experience suggest that any given heuristic

P. Ross (✉)
School of Computing, Edinburgh Napier University, Edinburgh, UK
e-mail: P.Ross@napier.ac.uk

E.K. Burke and G. Kendall (eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, DOI 10.1007/978-1-4614-6940-7_20,

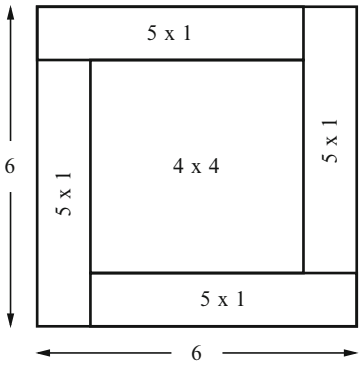

Fig. 20.1 The optimal cutting layout

has some weaknesses and will recommend some bad decisions in certain cases. *Hyper-heuristics* tries to address such issues, in two main ways:

- By exploring whether some suitable combination of existing heuristics can offset the weaknesses of any one of them, so that each is only applied when it is not weak;
- Or, by trying to discover new heuristics through some kind of meta-heuristic search process (tabu search, genetic algorithm, genetic programming, etc.).

Some authors (for example Cowling et al. 2001) describe hyper-heuristics simply as "heuristics to choose heuristics". A recent survey and classification of different approaches can be found in Burke et al. (2009a), which also proposes as a definition that a "hyper-heuristic is an automated methodology for selecting or generating heuristics to solve hard computational problems". Chakhlevitch and Cowling (2008) present another survey with a different classification.

Broadly speaking, some studies aim to produce *constructive* heuristics that will build a solution to a problem step by step: heuristics are used to decide how to extend a partial solution. Other studies aim to produce *refinement* or *perturbing* heuristics that work on a fully-specified candidate solution but try to improve its quality. And some studies aim to produce a hybrid of these, both extending and modifying partial solutions. Constructive methods tend to be fast and have a natural stopping point. Solution-refinement methods can be appreciably slower, perhaps stopping only when no further progress seems likely, but can deliver better final results.

This chapter is an introduction to hyper-heuristics. It discusses the issues that make hyper-heuristics research distinctive, presents some illustrative examples and a brief survey of past research, and offers some suggestions about interesting directions for future research. Finally, there is a selection of useful links to relevant material.

20.2 The Need for Hyper-heuristics

There is a vast literature on methods for tackling problems in combinatorial optimization and operations research (OR), including commercially significant ones such as scheduling and timetabling, vehicle routing, and packing as well as rather more generic (if no less valuable) ones such as constraint satisfaction. In many cases researchers have concentrated their efforts on trying to find one or more solutions that are optimal according to some chosen criteria. However, as Chambers Twentieth Century Dictionary (1974 edition) puts it:

> **optimal**, etc. See **optimism**.

The online Chambers 21st Century Dictionary has updated this to

> adj most favourable; optimum. [. . .].

The earlier version slyly embodies a pragmatic truth: often it is unreasonable to hope to find the very best possible answer. The search may be far too expensive. Or, the optimal answer may be too dependent on the chosen criteria, so that some seemingly small, circumstantial change in the problem or in the criteria renders the discovered optimum useless or seriously sub-optimal. The criteria themselves may be unreasonable or unrealistic. In practice, what is wanted is often good answers, found fairly or very quickly, where "good" means meeting some minimum acceptance criteria or, perhaps, usefully better than present practice can deliver. Also, there is often a requirement that the method of finding such answers must be sustainable for business use.

For example, suppose a large supermarket chain wants to get into the business of providing home deliveries of orders placed over the internet. Their competitors already do it, they feel they must do so as well. Customers will expect that deliveries should be made within some reasonable time window, chosen by the customer from a set of current possibilities. This looks at first sight like a classic instance of a vehicle routing problem with time windows: given a set of deliveries, each to be made within some time window, find the minimum number of vehicles required; the vehicles are assumed all to have a certain average speed and a given maximal capacity. There are many papers available that study such problems; for example, Bräysy (2003) describes a four-stage approach. The first stage uses some route construction heuristics to propose an initial solution. The second stage also uses heuristic methods—it tries to reduce the total number of routes by, repeatedly, selecting a route and seeing if those customers can be squeezed into other routes, perhaps involving some cascade of shifts of customers from those routes to others. The third and fourth stages keep the number of routes fixed but try to improve them, reducing the distance traveled and then exploring changes to a cost function that combines distances and waiting times in case the search has become stuck in a local minimum. The overall aim is to minimize total distance traveled and number of vehicles required, and the approach was tested on a number of standard benchmark problems involving up to 400 customers and also 2 real-world problems with 417 customers each. Although results were very good when compared with previous approaches,

all the approaches took anywhere from half an hour to several hours to solve the real-world instances (see Table 5 in that paper). Those were much more demanding than the benchmark problems because the customers were less spread out, necessitating much more local search effort.

Even if you are not directly interested in vehicle routing problems, it is worth looking at the recent literature because it illustrates various points about solving combinatorial and OR problems:

- The more effective approaches tend to employ two or more stages each of which utilizes one or more heuristics, rather than being monolithic;
- The widely used artificially generated benchmark problems may not be representative of real-world ones;
- If the approach involves some randomized elements, even if only to break ties, then different runs can produce significantly different results and it may be necessary to do several runs because the first one or two may be unreasonably poor—which you don't know until you try.

But there are more pragmatic considerations too. A supermarket probably does not want to spend CPU-hours per day solving one problem for one locality, and doing that for each locality where it operates. Even if it has its own IT department and in-house expertise, it will also have to factor in the cost of maintaining the necessary software and periodically upgrading or replacing it as the situation demands. And the problem itself is not static: solutions can be partly built as orders arrive: latecomers may not be offered the full range of delivery windows if some are already full. There are legal constraints on driver hours; drivers can fall ill at short notice; vehicles can break down; changing traffic patterns will affect average speeds; any delivery can fail because the customer forgets to be there to accept it; and so on. The supermarket is likely to want to be able to produce good, reasonably low-cost answers that are not too fragile in the face of changing situations, rather than solutions that are genuinely optimal with respect to just one or two criteria. Hyper-heuristics offers a possible way to discover an effective, fast process for scheduling the deliveries that can be used day to day, on ever-changing problems, with relatively little effort. Garrido and Castro (2009) used a hybrid hyper-heuristics method to find stable, good-quality solutions for certain kinds of vehicle routing problems.

More generally, this example raises some important issues:

- Some problems are *offline* problems: their full details are available at the start. Others are *online* problems: their details only emerge over time, and a solution must be built incrementally;
- In practice, it will often be important to be able to offer some performance guarantees. These might not be very strong, perhaps just "better than X" rather than (say) "within 5% of optimal according to criterion C";
- There can be major differences between academic and real-world problems, and between academic and real-world concerns.

It is also useful to consider how success is to be judged. Standards in many fields can sometimes be ad hoc. A recent useful classification, which considers heuristic

methods applied to forest planning, can be found in Bettinger et al. (2009); it offers a multi-level categorization:

Level 1: *No validation or performance is established.*
Level 2: *Self-validation* using basic statistics on the distribution of results. This includes worst-case (level 2a), best-case (level 2b) and average-case (level 2c) results, performance variation (level 2d) and sensitivity analysis of parameters (level 2e).
Level 3: *Comparison with other heuristic results.*
Level 4: *Comparison with an estimated global optimum solution*, for example by applying results from extreme value theory.
Level 5: *Comparison with optimal solutions generated for similar problems*, for example by relaxing the problem to get something more tractable.
Level 6: *Comparison with solutions provided by exact techniques.*

When thinking specifically about hyper-heuristics, these levels need to be modified slightly. It is be important to look at performance on a range of test problems, which should be different from any problems used to discover a new candidate heuristic in the first place. After all, the whole rationale of a heuristic is to suggest what to do when faced with an unfamiliar problem, and so it should be evaluated against previously unseen cases. If the discovery process involves iterative testing as well as learning, then (as usual) there should be separate learning, validation and test sets. Each visit to a validation set after a bout of learning produces some feedback that influences later learning episodes, and so the performance on the validation set is likely to increase in subsequent stages. In order to avoid such 'data snooping' the test set should be used only once, at the very end. It can also be useful to include an extra level to the above categorization:

Level 7: *Stress-testing*, for example by trying to devise new problems that "break" the heuristic in some way, delivering poor results or introducing previously unconsidered aspects.

Assuming that the aim is to produce a heuristic that has some general applicability, it is also wise to include some easy problems in the test set as well as hard problems. For some types of problem domain, there can be heuristics that *only* do well on hard problems but which struggle when presented with an easy problem!

The next sections present some illustrative examples of hyper-heuristics.

20.3 Hyper-heuristics for Boolean Satisfiability

20.3.1 The Problem Area

Suppose that x_1, x_2 and x_3 are Boolean variables—that is, each one is either true or false. Find an assignment of true or false to each of the three variables such that the two clauses:

$$x_1 \vee \neg x_2 \vee \neg x_3$$
$$x_1 \vee x_2$$

are both true (where $\vee$ means "or" and $\neg$ means "not"). Clearly, any assignment in which x_1 is true works; also, any other assignment in which x_3 is false and x_2 is true works. This is a trivial example of a Boolean satisfiability problem. In conjunctive normal form, the problem consists of a number of clauses all of which are required to be true, and each clause is a disjunct (that is, joined by $\vee$) of some variables and/or negated variables. If each clause contains at most two variables, the problem can be solved in polynomial time; if the clauses contain three or more variables, the problem type is NP-complete. Boolean satisfiability problems occur in many practical applications, such as automated planning, automated software testing, hardware design and biology (Marques-Silva 2008). For example, a newly-fabricated integrated circuit may contain a fault in which the output of a certain gate is stuck at logic 1 or at logic 0. Such a fault may be detected by generating a set of 0/1 inputs that would, if the circuit were correct, produce a set of outputs different from those actually observed; the task of finding a suitable set of inputs can be formulated as an instance of a Boolean satisfiability problem.

If the problem has n variables then there are 2^n possible assignments so that if n is not small exhaustive search, even with some smart pruning, may not be possible. Various heuristics are commonly used. For example:

GSAT: flip the variable that will produce the greatest increase in the number of satisfied clauses (call this the highest gain). If there is more than one such variable, choose randomly between those variables.

GWSAT(p): with probability p, randomly choose an unsatisfied clause and flip a randomly-chosen variable in it. With probability $(1 - p)$, use GSAT.

WalkSat(p): randomly choose an unsatisfied clause. Find those variables which, if flipped, produce no net change in the number of satisfied clauses. If there is at least one such variable, choose one and flip it. If there are none, then with probability p apply GSAT to the variables in the chosen clause, and with probability $(1 - p)$ choose a random variable from that clause.

20.3.2 The Heuristic Generation Process

Bader-El-Den and Poli (2008) analyzed several such heuristics and created a simple recursive grammar that was capable of expressing them as well as many more candidate heuristics. They then used a grammar-constrained form of genetic programming (Poli et al. 2008) to search for a new heuristic that would give good performance on a range of benchmark problems.

The grammar used the terminals and functions shown in Table 20.1, and the set of functions and terminals also happens to imply the grammar actually used, although this need not always be the case. Some functions that select from a list have an optional argument, shown as {op}, used to break ties; the default is to break ties by

Table 20.1 Functions and terminals for heuristics for Boolean satisfiability (Bader-El-Den and Poli 2008)

Functions	
`FLIP v`	The fixed, top-level-only action: flip the variable `v`
`RANDOM l`	Return a random variable from the list
`MAX_SCR l {op}`	Return the variable with the highest gain.
`MIN_SCR l {op}`	Return the variable with lowest gain.
`SCR_Z l {op}`	Return a variable that, if flipped, produces zero net change in the number of satisfied clauses.
`MAX_AGE l {op}`	Return the variable that has not been flipped for the longest time.
`IFV prob v1 v2`	With the given probability return variable `v1` else `v2`
`IFL prob l1 l2`	With the given probability return list `l1` else `l2`
`PROB`	A probability: one of 0.2, 0.4, 0.5, 0.7, 0.8 or 0.9
Terminals	
`ALL`	A list of all clauses
`ALL_USC`	A list of all currently unsatisfied clauses
`USC`	A random unsatisfied clause (the same at each appearance in the whole expression)
`RAND_USC`	A random unsatisfied clause (not necessarily the same each time)
`TIE_RAND`	Flag: break ties at random
`TIE_AGE`	Flag: break ties by selecting the least-recently flipped.
`TIE_SCR`	Flag: break ties by gain.
`NOT_ZERO_AGE`	Flag: skip the just-flipped variable when breaking ties.

random choice. Where appropriate, a clause is treated as a list of variables, whether negated or not, that it contains. For example, GWSAT(0.5) could be expressed in this language as

```
(FLIP (IFV 0.5 (MAX_SCR ALL TIE_RAND) (RANDOM USC))).
```

The initial randomly generated population had some existing good heuristics injected into it, thus providing some known-good material from which to breed. At each stage, each member of the population was evaluated on a set of training problems by using it repeatedly to modify an initial assignment of false to each variable in the problem. Fitness was based on the number of problems solved, the number of flips done and the size of the heuristic.

The training and testing problems were all taken from a standard benchmark collection of hard problems (Satlib 2012), in particular those involving between 25 and 100 variables each, with 3 variables per clause (3-SAT), and all known to have at least one solution. Overall, the results were very encouraging: generated heuristics, trained on one subset of problem, also performed well on other subsets and produced results "that are on par with some of the best-known SAT solvers". Figure 20.2 shows an example of a generated heuristic.

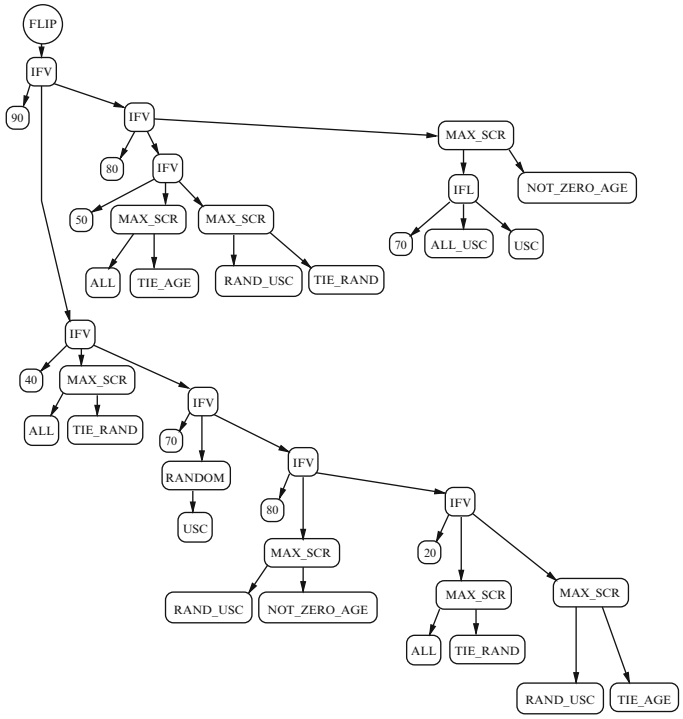

Fig. 20.2 A generated heuristic for 3-SAT problems, from Bader-El-Den and Poli (2008) (amended)

20.3.3 Remarks

Bader-El-Den and Poli (2008) also introduced the notion of a *disposable* heuristic, namely a heuristic intended to be used on a specific sort of subset of a more general family of problems. In this case, the aim was to focus on relatively small problems with three variables per clause that are known to have solutions. In this usage *disposable* means not fully general-purpose, just tailored to a subclass of some kind. Later work by different authors has occasionally abused this notion of *disposable* by focusing on just one problem, trying to generate a heuristic that will solve only a single instance. But, as mentioned earlier, one of the hallmarks of a heuristic is that it should usefully be applicable to unseen problems.

This example is a good illustration of hyper-heuristics: it starts with some well-studied heuristics and devises a search space that includes them and many more variants. Fukunaga (2008) describes a similar study, also using genetic programming but involving a more expressive grammar and a larger search space, which was able to generate heuristics competitive with some of the best SAT algorithms to date. See Fukunaga (2002) for a shorter overview of the grammar and some early results using a much-simplified form of genetic programming.

A somewhat different approach is outlined by Bittle and Fox (2009). They use a version of the SOAR cognitive architecture to generate a constructive hyper-heuristic comprising a large number of condition-action rules, in which the conditions encode aspects of the current state of a solution that is under construction, and the action encodes how to select a variable and a value for it.

20.4 Hyper-heuristics for Timetabling

20.4.1 The Problem Area

Timetabling problems are often highly constrained and incompletely specified. In a typical problem the task is to timetable a number of weekly lectures or a number of exams. Some events can only happen in specific rooms, perhaps because they require unusual facilities or because the room must be large enough to hold all the participants. Each person (staff or student) may only be available at some limited times. There are also *soft* constraints that should be honored if possible but can be violated if necessary. For example, it may be desirable to try to space events out or to cluster certain events, but travel timings may make it inconvenient to put some pairs of events too close together. And, of course, it may not be possible to have a full and accurate specification by the time that the timetable has to be finalized.

Heuristics for timetabling problems are similar in spirit to those for SAT and for SAT's close relation, constraint satisfaction problems. In constraint satisfaction problems, each variable has a domain of possible values and there are constraints between pairs of variables, typically in the form of a pair-specific set of disallowed pairs of values. In all three problem areas, solutions are typically constructed by repeatedly choosing a variable and then choosing a value for it, in some heuristic fashion.

It is also common to view timetabling as a graph-coloring problem at heart. Events are represented as nodes and two nodes are linked by an edge if they cannot, for whatever reason, be placed in the same timeslot. The basic task is then to color the nodes of the graph in such a way that no two nodes linked by an edge have the same color—the set of colors represents the set of timeslots. There is a large body of theory about graph coloring, see for example Kubale (2004).

20.4.2 The Heuristic Generation Process

Burke et al. (2007b) used up to six heuristics based on graph-coloring notions to choose events. A chosen event would then be inserted into the lowest-cost available timeslot, cost being determined by the soft constraints. The possible heuristics were:

1. LD: (largest degree) choose the event with the largest number of hard constraints;
2. LWD: (largest weighted degree) as LD, but the hard constraints are weighted in some way rather than being regarded as equal;
3. CD: (color degree) choose the event that has the most hard constraints involving already-placed events;
4. LE: (largest enrolment) choose the event involving the most people;
5. SD: (saturation degree) choose the event with the fewest available timeslots;
6. RO: (random order) choose at random.

and different mixes of these were tried.

A solution is represented by a sequence of these heuristics, applied in turn to build a complete solution. Each heuristic was used to place two events before moving on to the next heuristic in the sequence, thus if there were n events to be scheduled the sequence would be $\lfloor \frac{n}{2} \rfloor$ long (no selection method is needed for the final event). The decision to use each heuristic twice was guided by the empirical observation that, if the list was n long then runs often occurred, and then by some experimental exploration of possible choice of repetition factor (2–5).

To begin with, tabu search was used to find the best sequence of heuristics for a family of benchmark timetabling problems. Each sequence was evaluated by constructing a complete timetable, then using a local search to try to improve the timetable further by moving events around and evaluating that improved result. If a sequence produced an invalid timetable, the offending sub-sequence of heuristics would be added to the tabu list.

Later work (Qu and Burke 2009) tried replacing tabu search by steepest descent, iterated local search and variable neighborhood search. The latter two were found to be better than tabu search for the purpose. Ochoa et al. (2009) analyzed the fitness landscape, observing that there were many plateaus in the landscape but it was also globally convex, suggesting that the landscape does contain the sort of information that could help to guide a search towards the optimum.

Cowling and Chakhlevitch (2007) tackled practical personnel scheduling problems by using a large set of low-level perturbing heuristics, some of which choose an event to try altering and others of which decide how to alter a chosen event. The problems involved 50 or more training staff to be scheduled to handle between 147 and 224 training events in 16 different locations over some 3-month period; when scheduled by hand, they were taking around 9 days of a manager's time. Cowling and Chakhlevitch explored a broad range of ways of combining the heuristics using greedy or mildly greedy (*peckish*) and tabu search strategies. They conducted extensive experiments and concluded that hyper-heuristic methods were able to produce very good solutions, and in a very small fraction of the time previously taken by managers.

20.4.3 Remarks

One of the observations in Burke et al. (2007b) was that, assuming enough search time was allowed, it tended to be beneficial to include random ordering (RO) as one of the heuristics. But it is important to remember that if a solution method includes any non-deterministic components then different runs on the same data can produce different results, and it may then be important to factor in the cost of doing multiple runs in order to sample the space of possible outputs. If speed or reproducibility are of primary importance in your application area, it may be wise to omit any random-based component from the final product.

However, in the development stage, it can be helpful to try including a random-choice heuristic. If its inclusion improves the results, that is suggestive evidence that the other heuristics considered by the search process all failed to offer the good choice that the random-choice heuristic actually made. In that case, perhaps the basic set of heuristics or heuristic ingredients needs to be modified in some way.

20.5 Hyper-heuristics for Packing

20.5.1 The Problem Area

Given a set of containers, usually all of the same size, how can you pack a given set of objects into them so as to use as few containers as possible? There are many variants on this basic theme. The containers may be one-, two- or three-dimensional. In two or three dimensions, the objects may be rectilinear or may be of any shape. There may be a cost or weight associated with each object so that the task is instead to minimize that cost or weight while packing as many items as possible. The task may be to cut specified shapes from stock material, using only guillotine cuts, or using only cuts parallel to the edges of the material. A taxonomy of such problems can be found in Dyckhoff (1990); see also Martello and Toth (1990) and Coffman et al. (1996).

As before, the basic constructive approach is iterative: select an item and decide where to put it, repeatedly. Some approaches create a rating heuristic that evaluates any given placement of an item, and this rating heuristic is applied at each step to every remaining item in every possible position. Many heuristics have been suggested, for example:

- (For offline problems) select items, largest first, and put them in the first container that will hold them. If M is the minimum number of containers needed, this is known to require no more than $11M/9+4$ containers (Johnson 1973);
- (For offline problems) taking largest items first, pack a container until it is at least one-third full, then conduct a search for any single item that will fill the container, or else a combination of two or three items that will fill the bin;

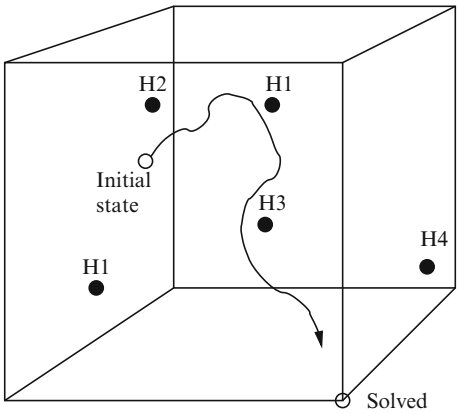

Fig. 20.3 A messy GA approach: the basic idea

- (For online problems) put the next item into the fullest container that will accept it.

There are also fairly obvious perturbing heuristics. For example, if a container is nearly full, such that it will not accept any other item, then search the other containers for a single item or any pair of items that will fill it.

20.5.2 Heuristic Generation Processes

Ross et al. (2003) describe a distinctive approach to solving offline one-dimensional packing problems, using a hyper-heuristic approach to create a very fast constructive heuristic. The idea is to associate heuristics with points or regions in a simplified problem-state space, for example encoding the current state of the partial solution as a vector of five real numbers: the proportion of small items, medium items, large and huge items remaining to be packed, and finally the proportion of the total number of items still to be packed. The definitions of *small*, *medium*, *large* and *huge* are somewhat ad hoc. A messy GA (Goldberg et al. 1989) is used to try to find a number of points in this five-dimensional space, each with an associated heuristic, that can be used to guide the packing process. Each chromosome contains a variable number of genes, and each gene is a block containing a five-dimensional vector and a named heuristic that can be regarded as the *label* for that five-dimensional point. A chromosome therefore describes a set of labeled points in five-dimensional space, as suggested by Fig. 20.3 and is decoded into a complete packing by the algorithm shown in Fig. 20.4.

Fitness of each chromosome was based on using it for a number of training problems, using a rolling regime of sampling problems rather than trying every training problem every time. The single final *algorithm* was able to deliver very creditable results on each of a large number of test problems. Moreover, since the simplified state space could be divided up into a suitably large number of cubes, each marked

```
Until( every item has been packed ) {
   Encode the current problem state;
   In the simplified state-space, find
     the nearest labeled point to the
     current state;
   Apply the heuristic on that label;
   Update the problem state;
}
```

Fig. 20.4 A state-space-guided packing algorithm

```
Fitness = 0;
for( each training problem ) {
  for( each item in the problem, in order ) {
    rating = negativeInfinity;
    for( each container ) {
       value = heuristic( item, container );
       if( value > rating ) {
         rating = value;
         bestContainer = container;
       }
    }
    pack item in bestContainer;
  }
  Fitness += solutionQuality( problem );
}
```

Fig. 20.5 Applying a candidate rating heuristic

with a chosen one of its nearest labeled points, a very slightly modified form of this algorithm could be applied to any new problem without any search being involved whatever.

A somewhat similar messy GA approach was used by Terashima-Marín et al. (2010) to tackle two-dimensional stock-cutting problems involving both regular and irregular shapes. Stock-cutting problems are simply packing problems involving the cutting of specified shapes out of standardized stock material; practical examples include tasks such as stamping car body parts out of sheet steel. Each block in the chromosome contained a vector identifying a point in an eight-dimensional simplified state space, and also two heuristics—one to select the next shape to be cut and one to decide whereabouts on the stock sheet it should be cut from.

Burke et al. (2007a) tackled the online version of the one-dimensional packing problem by using genetic programming to evolve a rating heuristic to decide where best to put the next item. An adequately large array of containers was used, but only non-empty ones counted at the end. Any candidate heuristic was evaluated by using it to tackle a number of training problems; each problem was handled as shown in Fig. 20.5.

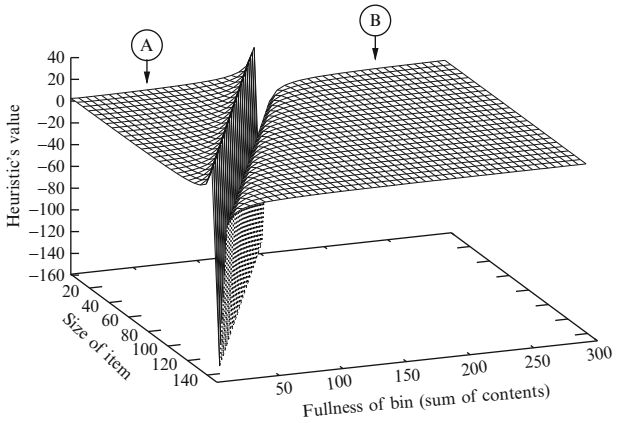

Fig. 20.6 A plot of the generated rating function for online packing

Unusually, candidate heuristics were allowed to overfill containers, but any such packing was penalized heavily so that the system learns not to overfill. The genetic programming used the standard $+, -, \times, /, \leq$ functions, where "/" is protected division that returns 1 if the denominator is zero and "$\leq$" returns 1 or -1. The terminals included the size of the item S, the fullness of the container F (that is, the sum of all that it currently holds) and the capacity C of the container. The `solutionQuality` function was

$$\texttt{solutionQuality} = \begin{cases} 1 - \frac{\Sigma(F_i/C)^2}{n} & : \text{ if legal} \\ \text{big constant} & : \text{ if not,} \end{cases}$$

where F_i is the fullness of container i and there are n non-empty containers. The approach was developed using a number of modest-sized benchmark packing problems, all using containers with size $C = 150$. Burke et al. suggested that a heuristic developed using a specific subset of the problems tended to show rather less good performance on other subsets with different characteristics, and gave the following example of a generated rating function:

$$\frac{2S+F}{S+F} + \frac{C}{((\frac{F}{C} \leq 2C-F)+(C-S-F))}.$$

Figure 20.6 shows the value of this rating function for values of F from 1 to 300 (since the container could be overfilled beyond its capacity of 150 by a bad heuristic) and values of S from 1 to 150. The peak of the ridge lies along the line $S+F = C$. Either by careful analysis of the formula, or by inspecting a large table of the formula's values, it can be seen that the region marked A slopes smoothly upward from the origin to the ridge, and the region marked B all lies strictly below any point in region A. This means that the heuristic rating function will never allow a container to be overfilled, because points with $S+F > C$ all have lower values than

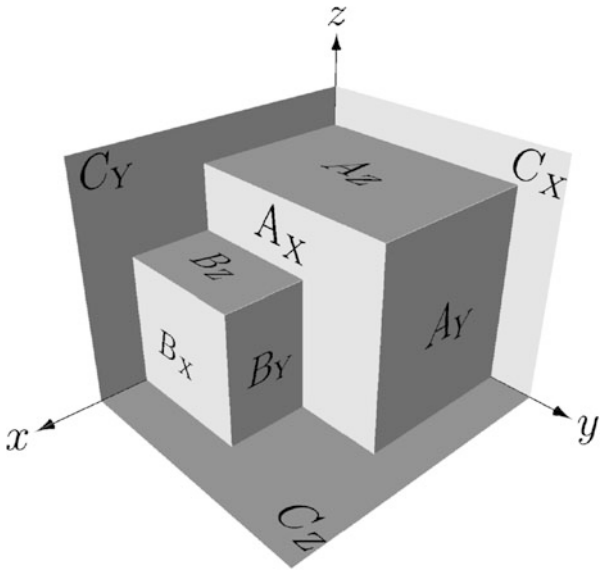

Fig. 20.7 Three-dimensional packing

any points satisfying $S + F \leq C$. Assuming that the items are never larger than the container, the rating function is exactly equivalent to the sensible online heuristic that says "put the item in the fullest container that can accept it".

Allen et al. (2009) have extended this idea to three-dimensional packing of rectangular objects in rectangular containers. A more complicated notion of packing is used. Figure 20.7 shows a view of a container with two items already packed. There are five places where the next object might be placed, namely the five places where three back-surfaces meet. Items should be fully supported from below rather than overlapping the lower items: an item placed on surface A_z should not overhang B_z or C_z. If nothing can be used to infill a gap, such as the gap to the left of B_x, then the gap is deemed to be filled and the supporting surface B_z is deemed to expand over the gap. However, the basic notion, of trying every item in every possible corner and in every possible distinctive orientation, remains the same. The functions used in the genetic programming are more elaborate than in the one-dimensional case, and are not repeated here.

20.5.3 Remarks

See Burke et al. (2009b) for a survey of genetic programming applied in hyper-heuristics research. Arguably, there are numerous papers which have used genetic programming to create a good rating function and yet which do not mention hyper-heuristics. For example, Hauptman and Sipper (2005) used genetic programming to

create a good board evaluation function for certain kinds of chess end-game, namely those in which either side has a queen or a rook or both. A large number of chess-specific terminals were used, such as "is my king protecting one of my pieces?", "number of legal moves for opponents king" and "distance of my king from edge of board" as well as basic logical operations such as and/if/or/not. Each function was evaluated by having it play against some other members of the population with a randomly-generated initial end-game position. The final result was able to draw against a world-class (2004) chess program. Hyper-heuristics is still a fairly new topic; perhaps in future more such examples will come to be known as instances of the topic.

However, if you want to explore hyper-heuristics, packing is a better place to start. There are many benchmark problems available, with many results available for comparison. And as yet, not many authors have tried using hybrid hyper-heuristics for online problems—that is, not only deciding where to put the next item but also revisiting earlier decisions from time to time so as to shuffle various items between containers. After all, that is what human supermarket bag-packers often do, while trying to pack bags fast and avoid undue delay at the end.

In earlier work, Ross et al. (2002) also tried using an extended classifier system as the search engine for one-dimensional packing. Classifier systems try to discover a good set of condition-action rules by simulated evolution, but early systems tended strongly to favor rules that were very general. XCS, the extended classifier system (Wilson 1995), avoids that trap by focusing more on the prediction accuracy of a rule than on the number of test cases it covered, and has been extremely successful in areas such as data mining. See also Marin-Blazquez and Schulenburg (2007).

20.6 A Little History

Although hyper-heuristics has only emerged as a recognizable topic area within the last decade or so, the basic concepts are certainly older than that. For example, the COMPOSER system (Gratch et al. 1993) planned communication schedules between Earth-orbiting satellites and ground stations, with a maximum interval between communications with a given satellite. The ground stations that could be used were constrained by satellite orbits. The scheduler used heuristic methods to try to build a schedule, deciding which unsatisfied constraints to focus on next and how to try to satisfy them. Because there were several possible heuristics to use in each case, the system used a simple hill-climbing approach to investigate combinations of them, testing each on 50 different problems, and was able to discover an effective combination. Wah et al. (1995) developed an early heuristics-learning system named *Teacher* and discussed a number of the issues raised in this chapter.

The Adaptive Constraint Engine (ACE, Epstein et al. 2002) also implemented some interesting ideas that are now being explored in hyper-heuristics research. ACE used reinforcement learning to try to discover good heuristics for constraint satisfaction problems, using an *advisor-based* architecture named FORR, which was

an acronym for "FOr the Right Reasons". Each of a number of advisor components looked after some specific principle, such as "prefer the variable with the fewest remaining values in its domain", and generated explicit comments about the suitability of its principle for the legal actions in any given state. ACE then learned what to do in each state, based on such comments. ACE built upon some of the ideas embodied in cognitive architectures such as SOAR (SOAR 2012) and ACT-R (ACT-R 2012).

20.7 Some Research Issues

20.7.1 No Free Lunch?

It would be natural to wonder just how good any generated heuristic could be. Perhaps some hyper-heuristic composition of other heuristics will manage to avoid the specific weaknesses of each of the ingredients, but might the composition not have new weaknesses of its own? Or do hyper-heuristics offer the prospect of some kind of "free lunch" to operations researchers?

Consider a finite-search problem defined on a finite domain. This is not a restrictive supposition; in fact, virtually all computer-based searching is done on such problems, because only finitely many values can be represented in IEEE floating point arithmetic, or in other formats within a finite-memory computer. Wolpert (1995) proved a famous "No Free Lunch" theorem that said that, averaged over all such problems, all search algorithms that do not revisit already-visited points have exactly the same average performance. This result is not as surprising as it might seem. Nearly all problems have no exploitable structure that could be used to guide a search—they can only be defined by a large lookup table, and nothing about the values would tell you where the optima actually lie. You can easily generate examples of intractable search problems for yourself. For example, start with $f(x) = x^2$ in the interval $x \in 0\ldots100$ and then, in your program, redefine $f(\pi^2 + 23)$ to be 10,000,000 instead of ≈1,080.410893. Now the maximum of this modified function $f()$ lies at $x = (\pi^2 + 23)$ (or rather at the closest value to this that the computer can represent internally) rather than at $x = 100$, but how could you find it without sampling that one specific value? Nothing about the other values of the function will tell you that this amazing spike is lurking there. But in practice we are not interested in such random- or random-seeming functions; the problems we humans are interested in have some internal structure and some degree of predictability or continuity about them. We therefore want to find algorithms that can perform well on the subset of problems that arise naturally in some context; the difficulty lies in characterizing that subset properly in such a way that we can design a really effective algorithm for the members of that subset. Hyper-heuristics sidesteps these issues of characterization and design to some degree by instead conducting a search for an adequately effective algorithm.

Schumacher et al. (2001) showed that the No Free Lunch theorem applies to smaller sets of problems than *all* problems defined on a finite domain, and in particular it applies to a set of problems that is closed under permutation of the set of values. To get an idea of what that means, consider the domain $x \in [1,2,3,4,5]$ and the set of values $[23,47,51,55,93]$. There are 120 different ways of defining a function $f()$ on that domain by assigning these values to $f(1)\ldots f(5)$, thus producing a set of problem (functions) closed under permutation of the values. But more significantly, Schumacher et al. showed that the No Free Lunch theorem only applies to such problem sets; if the problem set is not closed under permutation, then the theorem is not true and there are some algorithms that have better average performance than others. As Poli and Graff (2009) have pointed out, this means that hyper-heuristics-generated algorithms may turn out to be better than average if the problem set to which they are applied is not closed under permutation.

In practice, this is still more of theoretical than practical interest, but it does highlight the research question of how best to characterize the set of problems on which a generated heuristic is capable of showing particularly good performance. There has been some practical work done on such issues already—see Sect. 20.5.2—but much more could be done. One possible approach is to investigate systematically how the performance of a generated heuristic varies as it is applied to other sets of problems in the same area but with different characteristics. Another approach might be to use search techniques to try to create problems that cause the heuristic to stumble in some way.

20.7.2 Search Methods

Many different search techniques have been tried already, such as genetic programming, tabu search, straightforward genetic algorithms, variable-neighborhood search, simulated annealing, classifier systems, ant colony methods, particle swarm methods and so on.

Genetic programming is typically used to create a rating function of some kind but the general technique still suffers from some weaknesses that researchers are trying to address, such as the issue of *bloat*: the creation of overly-elaborate functions that may be unreasonably opaque to the end users or which might bias the search undesirably. As illustrated earlier, sometimes a generated function is relatively simple and can be analyzed to simplify it even further. But it is less easy to analyze a function tree of the kind shown in Fig. 20.2 in order to try to simplify it. Various kinds of tree-pruning methods used in data mining could be used to explore whether a generated heuristic is either overly complicated or perhaps even too specific to the training examples.

The *messy GA* and classifier systems approaches mentioned in Sects. 20.5.2 and 20.5.3 typically produce a rather different sort of output, which directly associates a choice of existing heuristic with some problem-specific conditions. Few people have yet done much investigation of these search methods in the context of

hyper-heuristics—see Terashima-Marín et al. (2008) for an example that tackles hard constraint satisfaction problems and see Terashima-Marín et al. (2010) for an example that tackles regular- and irregular-shaped stock-cutting problems. Essentially these methods generate sets of condition-action rules, but the actions do not have to be simply about choosing a heuristic. In a hybrid system, the actions might also be about when to switch from a constructive phase to a solution-perturbing phase or vice versa, or might be about altering the scope of a search process, and so on.

The example described at the start of Sect. 20.3.2 searched a space defined by a grammar, but the grammar in question was essentially defined by the type of information involved. Grammatical evolution (O'Neill 2003, 2012) could be used to explore more varied kinds of grammar, although there are open questions about how best to conduct the search through a grammar-defined space (Castle and Johnson 2010).

As yet, little has been done on distributed and parallel versions of hyper-heuristic search, but see Biazzini et al. (2009) and León et al. (2009) for examples.

20.7.3 Representation Issues

Various researchers have begun to explore how the choice of representation affects the results, for instance by studying which base-level heuristics to include or omit (Burke et al. 2007b) or which problem-features to include (Ross et al. 2003). In many instances the choice of representation has been guided by a study of existing heuristics, although the use of sensitivity analysis to investigate what really matters is still relatively rare. It is also conceivable that the choice of representation could itself be heuristically guided and problem specific. Think of trying to solve a rectangular jigsaw puzzle. It is commonplace to start with the four corner pieces and then the edge pieces. Thereafter, the choice of what to focus on tends to be guided by the specific details of the puzzle itself. For instance, if the puzzle contains lots of relatively undifferentiated blue sky then that might best be left until most of the rest of the puzzle has been completed, because then there will be more guidance available from the surrounding pieces. The same sort of notion might be applied in hyper-heuristics.

20.7.4 Performance Guarantees

Although real-world users of heuristics often like to have performance guarantees, little has as yet been done to try to generate heuristics that have formally provable performance bounds, and indeed relatively few human-generated heuristics seem to offer such guarantees either. Section 20.5.1 mentioned one example: the "largest first" heuristic that uses no more than $11M/9 + 4$ containers where M is the minimum.

One possible way to make progress in this area might be to try combining hyper-heuristics ideas with ideas from the study of *parameterized algorithmics*, sometimes also referred to as *fixed-parameter algorithms* (Niedermeier 2006; Downey and Fellows 1999). The key idea in fixed-parameter algorithms is to introduce an additional, fixed parameter that will help to focus the search for a solution and will significantly simplify the process. Here is an example. Imagine that you have been asked to help modernize a large national railway system—see Weihe (1998) for a motivating real-world example. It is very costly to have a ticket office and ticket-checking barriers at every station; all you really need is to have such facilities at one end or the other of every possible journey. So you need to find the fewest possible number of stations at which to install such facilities; this is known as the *vertex cover problem* and it is known to be NP-hard. In graph theory terms, if a graph has N nodes, representing stations, and E edges, representing journeys, then the task is to find a minimal subset of nodes such that every edge meets at least one of them. Clearly the size of such a subset will be at most $\lfloor E/2 \rfloor$ but might well be much smaller, and even some kind of non-exhaustive search could be very costly. But if you introduce a new parameter k and search only for covers of size at most k, the search can be made much more focused. This is because any node that has $e > k$ edges attached to it must be in the cover; if it were not in the cover, then all of the e nodes at the other ends of those edges would have to be in the cover, thus making the cover larger than k in size. By studying the degree sequence—that is, the ordered sequence of the number of edges meeting each node—it is often possible to home in on a modest number of possible choices for k, each of which may involve a far less costly search than in the unparameterized case and which involve an explicit bound on the solution size.

It may be possible to unite hyper-heuristic ideas with specific ideas about parameterized algorithms in order to generate heuristics with specific performance characteristics. This also raises the possibility of using hyper-heuristic ideas to generate new kinds of metaheuristic algorithms which will be able to announce their performance characteristics when applied to any specific problem.

20.8 Getting Started: The HyFlex Framework

The best way to learn more about hyper-heuristics is by trying some practical experiments. The HyFlex software framework (Ochoa et al. 2012) is a convenient tool for the purpose, written in Java. At the time of writing it includes four kinds of problem:

- One-dimensional bin-packing problems;
- Satisfiability problems;
- Personnel scheduling problems, not unlike the timetabling problems described earlier;
- Permutation flow shop problems. These are work-scheduling problems in which there are n jobs to be completed and each job needs to visit m machines, in each case visiting machine 1 first, machine 2 second and so on (so that the work-flow

is the same for each job), but possibly visiting each machine for differing job-specific amounts of time. Therefore the issue is to find that permutation of the n jobs which minimizes the total time to completion.

In each case a number of benchmark problems is supplied, and also a number of heuristics. The heuristics are black boxes: they can be applied to a problem but their internal workings are not available at the level of the system's API. There are four kinds of heuristic: mutational/perturbing ones; ruin–recreate ones that make large-scale changes by partially destroying a solution and rebuilding it; local search ones that stop when a local optimum is found or a stopping condition is met; and crossover ones that construct a new solution in some way from two current ones.

The API provides hooks to initialize and manage a population of candidate solutions, set and monitor a time limit, and apply any of the relevant heuristics. Two parameters, described as *depth of search* and *intensity of mutation*, make it possible to control certain aspects of the heuristics' operations. The main task is therefore to create your own program that learns through experience which heuristic to apply, with what parameter settings.

The framework provides a simple and convenient entry-point into some of the practical aspects of hyper-heuristics. However, it enforces a strict separation between the domain-specific aspects, hidden from the end user behind a so-called domain barrier, and a domain-independent form of hyper-heuristic exploration. For large real-world applications, it is often desirable to use far more domain information than a tool such as HyFlex can conveniently offer.

20.9 Tricks of the Trade

Hyper-heuristics is an excellent area for research: there are still many issues awaiting thorough explorations. This section lists some advice.

20.9.1 The Problem Area

If you have a choice, look for an area where a hyper-heuristic approach is likely to bring useful benefits. Finding new ways to slingshot a spaceship around the inner planets may be fun and that research may have valuable theoretical consequences, but the space agencies are not necessarily interested in fast-and-cheap, good-enough algorithms at the mission-planning stage. On the other hand, there may be useful practical scope for hyper-heuristic approaches in dynamic robot control applications: a moving robot may not be able to afford to wait for the completion of some expensive optimization routine. Whatever the area, remember that you will need a significant number and variety of problem instances.

If your focus is on very practical problems, perhaps provided by a commercial partner, you may not have that many problem instances to work with. It is possible to

generate additional examples, both by using random methods to create completely new problems and by making random changes of various sizes in existing problems. In many areas of combinatorics, generating random problems by some naive method may create examples that are not particularly hard, so it can be worthwhile to put some thought into how to generate problems. It can also pay to do a systematic study to try to discover what makes certain problems especially hard, at least for certain heuristics or search methods. It is sometimes possible to create problems for which you know the optimal answer. For example, in two-dimensional packing, you can start by dissecting the shape to be packed into a number of pieces, and then change some pieces a little by removing small amounts, such that the total amount removed adds up to less than the smallest piece. Thus all the pieces will still be needed.

20.9.2 Success Criteria

You should not simply be trying to produce better results than any recently published ones. Hyper-heuristic methods are unlikely ever to beat hand-crafted, CPU-intensive problem-specific methods. Your new hyper-heuristic methods may of course produce better results than some earlier hyper-heuristic methods, but that should not be your final goal. As a scientist, your aim should be to develop a new and greater understanding of what is going on, and to tell others about what you learned. Rather than simply obtaining good results, try to explore what made the difference and also, what the limitations of your methods are. More specifically, if your interest is in developing good-enough, fast-enough methods for some class of problems, there is going to be some kind of trade-off between speed and quality. Despite many decades of research, that kind of Pareto frontier is still poorly charted territory. It can be helpful to think carefully at the outset about the success criteria for your research, not least because you then have a clear goal to work towards. For example, a good hyper-heuristic might be one that is significantly cheaper to use than the alternative of trying every available heuristic in turn, and also provides results that can be better than the best heuristic but never significantly worse. Once you have found a good hyper-heuristic, investigate what features contributed to its success, for example by seeing what happens if you omit individual ingredients. Remember Danth's Law, paraphrased here as *"if you are forced to resort to declaring victory, you have probably already lost"*.

20.9.3 On-line or Off-line

Many papers discuss some class of off-line problems, in which all relevant information is available at the start, and others discuss on-line problems and assume that the task is to build a solution by deciding what to do with the latest information. Few authors have yet considered hybrid approaches in which, when new information arrives, a very limited time is spent on reconsidering the whole of the solution so

far and possibly reorganising it. This would make good use of the time between arrivals.

20.9.4 A Good Set of Heuristic Ingredients

Finding a good set of heuristics, or a set of heuristics components, can be tricky. Having too large a set can cripple the search process by making the search space too enormous. Look for heuristics that complement each other in some way. In solution-constructing approaches, remember that a single application of a heuristic does not have to do just one step of the construction: a heuristic can also be of the form *do* ... *until* ... or some other such looping construct. In solution-modifying approaches, it may be helpful to include "destructive" ingredients that may shift the focus to some other part of the space.

20.9.5 Fitness

Although the ultimate aim may be to find some algorithm that is capable of performing well on a wide variety of problems, it is not always necessary to evaluate each candidate on every problem. The computational burden can be reduced by evaluating each candidate on some randomly selected subset of problems, biasing the selection in favour of those problems that have participated least in the evaluations so far, and perhaps also in favour of those that have been hard.

Of course, you should adopt good practices such as having separate sets of problems for training and for eventual validation, or use cross-validation. It is also good practice to keep a set of problems apart for use as a final test set, that ideally should only be used once: you should try to avoid "data snooping", sometimes referred to as "data dredging", as far as you can.

20.9.6 A Good Set of Tools

There are many tool-kits available that implement search methods such as genetic algorithms, genetic programming or tabu search. It takes time to learn to use such tool-kits properly, and you will probably still need to do some programming work to add the features that you need, such as specific heuristics. It can often be worthwhile to build your own system rather than relying on a toolkit: that way, you get only what you need, your system does what you want, and you learn more than you would if you relied on someone else's ideas and skills. Aim to become a good programmer: it takes time, but is a very valuable talent.

Resist any temptation to build your own very elaborate graphical user interface (GUI). GUIs are limited when it comes to handling and analyzing large amounts of

data (and GUI-building is often simply a form of work-avoidance). Hyper-heuristic research often generates very large amounts of data, that are better handled using a powerful scripting language such as *Perl* or *Python*. Open-source tools such as *gnuplot* or the python-specific *matplotlib* are excellent for graph plotting, and *graphviz* is excellent for tree or network visualization; becoming thoroughly comfortable with such tools does take time but pays big dividends.

20.9.7 Attitude

Try to be skeptical about your results. Subtle programming errors, whether made by you or by someone else, can affect your results, so try to find ways to check them that do not depend on reusing critical parts of your code.

Aim to keep up to date with related research, such as developments in search technology as well as more specific things such as new ideas in your chosen problem areas. Reviewers routinely reject papers that claim good results on the basis of a comparison with outdated results or with obsolete algorithms. Try to be your own strongest critic.

Sources of Additional Information

This section lists some places to look.

- The *Handbook of Metaheuristics* (Glover and Kochenberger 2003) contains a lot of information about different kinds of heuristics, and includes a chapter about hyper-heuristics (Burke et al. 2003).
- The ASAP group at Nottingham University has a good website (ASAP 2012) that includes research publications about hyper-heuristics and links to timetabling problems and other resources.
- The *Journal of Heuristics*, published by Kluwer, contains many papers about heuristic methods generally. The tables of contents and the abstracts of papers are available online; full papers are available to subscribers to Kluwer Online.
- The *European Journal of Operational Research* also contains may papers relating to heuristics and to problems that might be tackled by hyper-heuristic methods. Again, abstracts are freely available online.
- The Metaheuristics Network site at www.metaheuristics.org provides information about various meta-heuristic techniques, references to papers and links to sets of problems in several areas: quadratic assignment, maximum-satisfiability, timetabling, scheduling, vehicle routing and an industrial hose-optimization problem. The aim of the Metaheuristics Network is to conduct scientific comparisons of performance between various metaheuristic techniques in different problem areas. Although hyper-heuristic methods are not explicitly considered, the site is

valuable because the problems have been generated or contributed by the members and performance results are being made available.
- The *OR-Library* (OR-library 2012) is a large repository of benchmark OR problems.
- Kendall (2012) has a useful online bibliography of papers about hyper-heuristics.
- The European Space Agency (ESA 2012) has some difficult spacecraft trajectory optimization problems available online.
- There is an online collection of frequency assignment problems (FAP 2012).
- See University of Melbourne data (2012) for some real-world university exam timetabling problems.

References

Allen S, Burke EK, Hyde M, Kendall G (2009) Evolving reusable 3D packing heuristics with genetic programming. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 931–938

ASAP research group (2012) http://www.asap.cs.nott.ac.uk/

Bader-El-Den MB, Poli R (2008) Generating SAT local-search heuristics using a GP framework. In: Monmarche N et al (eds) Artificial Evolution: Proceedings of the 8th International Conference EA 2007, Tours, France. Springer LNCS 4926, 37–49, 2008

Bettinger P, Sessions J, Boston K (2009) A review of the status and use of validation procedures for heuristics used in forest planning. Int J Math Comput Forest Nat Resour Sci 1:26–37. http://mcfns.com

Biazzini M, Bánhelyi B, Montresor A, Jelasity M (2009) Distributed hyper-heuristics for real parameter optimization. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 1339–1346

Bittle S, Fox M (2009) Learning and using hyper-heuristics for variable and value ordering in constraint satisfaction problems. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 2209–2212

Bräysy O (2003) A reactive variable neighborhood search for the vehicle routing problem with time windows. INFORMS J Comput 15:347–368

Burke E, Hart E, Kendall G, Newall J, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. In: Glover F, Kochenberger G (eds) Handbook of meta-heuristics. Kluwer, Dordrecht, pp 457–474

Burke EK, Hyde M, Kendall G, Woodward J (2007a) Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In: Proceedings of the GECCO 2007, London. ACM, New York, pp 1559–1565

Burke EK, McCollum B, Meisels A, Petrovic S, Qu R (2007b) A graph-based hyper-heuristic for educational timetabling problems. Eur J Oper Res 176:177–192

Burke EK, Hyde M, Kendall G, Ochoa G, Ozcan E, Woodward J (2009a) A classification of hyper-heuristics approaches. In: Gendreau M, Potvin J-Y (eds) Handbook of metaheuristics, 2nd edn. Springer, Berlin, pp 449–468

Burke EK, Hyde MR, Kendall G, Ochoa G, Ozcan E, Woodward JR (2009b) Exploring hyper-heuristic methodologies with genetic programming. In: Mumford CL, Jain LC (eds) Computational intelligence: collaboration, fusion and emergence. Springer, Berlin, pp 177–201

Castle T, Johnson CG (2010) Positional effect of crossover and mutation in grammatical evolution. In: Esparcia-Alcazar AI et al (eds) Proceedings of the EuroGP 2010, Istanbul. LNCS 6021. Springer, Berlin, pp 26–37

Chakhlevitch K, Cowling PI (2008) Hyperheuristics: recent developments. In: Cotta C, Sevaux M, Sörensen K (eds) Adaptive and multilevel metaheuristics. Studies in computational intelligence, 136. Springer, Berlin, pp 3–29

Coffman EG, Garey MR, Johnson DS (1996) Approximation algorithms for bin packing: a survey. In: Hochbaum D (ed) Approximation algorithms for NP-hard problems. PWS, Boston, pp 46–93

Cowling PI, Chakhlevitch K (2007) Using a large set of low-level heuristics in a hyperheuristic approach to personnel scheduling. In: Dahal KP, Tan KC, Cowling PI (eds) Evolutionary scheduling. Studies in computational intelligence, 49. Springer, Berlin, pp 543–576

Cowling P, Kendall G, Soubeiga E (2001) A hyperheuristic approach for scheduling a sales summit. In: PATAT 2000, Konstanz. LNCS 2079. Springer, Berlin, pp 176–190

Downey RG, Fellows MR (1999) Parameterized complexity. Springer, New York

Dyckhoff H (1990) A topology of cutting and packing problems. Eur J Oper Res 44:145–159

Epstein SL, Freuder EC, Wallace RJ, Morozov A, Samuels B (2002) The adaptive constraint engine. In: Van Hentenryck P (ed) Principles and Practice of Constraint Programming – CP 2002, Ithaca. LNCS 2470. Springer, Berlin, pp 525–540

ESA (2012) Global trajectory optimisation problems. C++ and Matlab code available. http://www.esa.int/gsp/ACT/inf/op/globopt.htm

Frequency assignment problems (2012) fap.zib.de/

Fukunaga A (2002) Automated discovery of composite SAT variable-selection heuristics. In: Proceedings of the AAAI 2002, AAAI Press, Edmonton, pp 641–648

Fukunaga AS (2008) Automated discovery of local search heuristics for satisfiability testing. Evol Comput 16:31–61

Garrido P, Castro C (2009) Stable solving of CVRPs using hyperheuristics. In: Proceedings of the GECCO 2009, Montreal, pp 255–262

Glover F, Kochenberger G (eds) (2003) Handbook of meta-heuristics. Kluwer, Dordrecht

Goldberg DE, Deb K, Kargupta H, Harik G (1989) Messy genetic algorithms: motivation, analysis and first results. Complex Syst 3:493–530

Gratch J, Chein S, de Jong G (1993) Learning search control knowledge for deep space network scheduling. In: Proceedings of 10th international conference on machine learning, Amherst, pp 135–142

Hauptman A, Sipper M (2005) GP-endchess: using genetic programming to evolve chess endgame players. In: Keijzer M et al (eds) Proceedings of the 8th EuroGP, Lausanne. LNCS 3447. Springer, Berlin, pp 120–131

Johnson DS (1973) Near-optimal bin-packing algorithms. PhD thesis, MIT Department of Mathematics

Kendall G (2012) A bibliography of hyper-heuristics and related approaches. http://www.cs.nott.ac.uk/~gxo/hhbibliography.html

Kubale M (ed) (2004) Graph colorings. AMS, Providence

León C, Miranda G, Segura C (2009) A memetic algorithm and a parallel hyper-heuristic island-based model for a 2D packing problem. In: Rothlauf F (ed) Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 1371–1378

Marin-Blazquez JG, Schulenburg S (2007) A hyper-heuristic framework with XCS: learning to create novel problem-solving algorithms constructed from simpler algorithmic ingredients. In: Learning classifier systems. LNCS 4399. Springer, Berlin, pp 193–218

Marques-Silva J (2008) Practical applications of Boolean satisfiability. In: Workshop on discrete event systems, Gothenberg, pp 74–80

Martello S, Toth P (1990) Knapsack problems. Algorithms and computer implementations. Wiley, New York

Neth H et al. (2009) Analysis of human search strategies. Technical report, Large Knowledge Collider Consortium. Deliverable 4.2.2. www.larkc.eu

Niedermeier R (2006) Invitation to fixed-parameter algorithms. Oxford lecture series in mathematics and its applications, 31. Oxford University Press, Oxford/New York

Ochoa G, Qu R, Burke EK (2009) Analyzing the landscape of a graph based hyper-heuristic for timetabling problems. In: Proceedings of the GECCO 2009, Montreal. ACM, New York, pp 341–348

Ochoa G, Hyde M, Curtois T, Vazquez-Rodriguez JA, Walker J, Gendreau M, Kendall G, McCollum B, Parkes AJ, Petrovic S, Burke EK (2012) HyFlex: A benchmark framework for cross-domain heuristic search. In: Hao J-K, Middendorf M (eds), European conference on evolutionary computation in combinatorial optimisation EvoCOP 2012. LNCS 7245. Springer, Berlin, pp 136–147

O'Neill M, Ryan C (2003) Grammatical evolution: evolutionary automatic programming in an arbitrary language. Springer, Berlin

O'Neill M (2012) The grammatical evolution page. http://www.grammatical-evolution.org

OR-library (2012) http://people.brunel.ac.uk/~mastjjb/jeb/orlib/

Poli R, Graff M (2009) There is a free lunch for hyper-heuristics, genetic programming, and computer scientists. In: Vanneschi L et al (eds) Genetic programming. Proceedings of the 12th EuroGP, Tübingen. LNCS 5481. Springer, Berlin, pp 195–207

Poli R, Langdon WB, McPhee N (2008) A field guide to genetic programming. Lulu, Raleigh. Also available as a free PDF from lulu.com

Qu R, Burke EK (2009) Hybridisations within a graph based hyper-heuristic framework for university timetabling problems. J Oper Res Soc 60:1273–1285

Ross P, Schulenburg S, Marín-Blázquez JG, Hart E (2002) Hyper-heuristics: learning to combine simple heuristics in bin packing problems. In: Langdon WB et al. (eds) Proceedings of the GECCO 2002, New York. Morgan Kaufman, San Mateo, pp 942–948

Ross P, Marín-Blázquez JG, Schulenburg S, Hart E (2003) Learning a procedure that can solve hard bin-packing problems: a new GA-based approach to hyper-heuristics. In: Cantú-Paz E et al (eds) Proceedings of the GECCO 2003, Chicago. LNCS 2724. Springer, Berlin, pp 1295–1306

Satlib—the satisfiability library (2012) http://www.satlib.org

Schumacher C, Vose MD, Whitley LD (2001) The no free lunch and problem description length. In: Proceedings of the GECCO 2001, San Francisco. Morgan Kaufman, San Mateo, pp 565–570

The ACT-R home page (2012) http://act-r.psy.cmu.edu/

The SOAR home page (2012) http://sitemaker.umich.edu/soar/home

Terashima-Marín H, Ortiz-Bayliss JC, Ross P, Valenzuela-Rendón M (2008) Using hyper-heuristics for the dynamic variable ordering in hard constraint satisfaction problems. In: Proceedings of the MICAI 2008, Atizapán de Zaragoza. LNCS 5317. Springer, Berlin, pp 407–417

Terashima-Marín H, Ross P, Farías-Zárate CJ, López-Camacho E, Valenzuela-Rendón M (2010) Generalized hyper-heuristics for solving 2D regular and irregular packing problems. Ann Oper Res 179:369–392

University of Melbourne (2012) http://www.or.ms.unimelb.edu.au/timetabling/. Exam timetabling data

Wah BW, Ieumwananonthachai A, Chu LC, Aizawa A (1995) Genetics-based learning of new heuristics: rational scheduling of experiments and generalization. IEEE Trans Knowl Data Eng 7:763–785

Weihe K (1998) Covering trains by stations or the power of data reduction. In: Battiti R, Bertossi AA (eds) Proceedings of the ALEX 1998, pp 1–8. http://rtm.science.unitn.it/alex98/book/weihe.ps.gz

Wilson SW (1995) Classifier systems based on accuracy. Evol Comput 3:149–175

Wolpert D, MacReady WG (1995) No free lunch theorems for search. Technical report SFI-TR-92-02-010, Santa Fe Institute