Memetic Comp. (2009) 1:205–219
DOI 10.1007/s12293-009-0022-y

SPECIAL ISSUE - REGULAR RESEARCH PAPER

Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework

Mohamed Bader-El-Den · **Riccardo Poli** · **Shaheen Fatima**

Received: 28 June 2009 / Accepted: 18 September 2009 / Published online: 11 October 2009
© Springer-Verlag 2009

Abstract This paper introduces a Grammar-based Genetic Programming Hyper-Heuristic framework (GPHH) for evolving constructive heuristics for timetabling. In this application GP is used as an online learning method which evolves heuristics while solving the problem. In other words, the system keeps on evolving heuristics for a problem instance until a good solution is found. The framework is tested on some of the most widely used benchmarks in the field of exam timetabling and compared with the best state-of-the-art approaches. Results show that the framework is very competitive with other constructive techniques, and did outperform other hyper-heuristic frameworks on many occasions.

Keywords Timetabling · Genetic programming · Hyper-heuristics · Heuristics

1 Introduction

Several of the most successful heuristic and metaheuristic approaches for hard real-world combinatorial problems rely upon the appropriate coordination of low-level heuristics that incorporate domain specific knowledge (e.g., local search operators, constructive methods, genetic operators, etc.) to produce solutions for a specific problem instance. Although these metaheuristics (and hybrids) have demonstrated considerable success in a range of areas they have, on the whole, been manually tailor made for the specific problem in hand.

There are flexible tools, such as iOpt [54], EasyLocall++ [26] and some other theoretical tools [32], which allow the user to discover metaheuristics manually more easily. However, the development effort and cost for new heuristics remains considerable and often results are not directly re-usable for other problems. Also, while these methods are a great asset for practical problem solving, the performance of heuristics and meta-heuristics may vary significantly from one problem instance to another. Furthermore, it is hard to ascertain which heuristic is going to be the most efficient in solving a given set of problems, or a certain problem instance. This degree of unreliability and the requirement for human intervention in the use of heuristics are really the main motivations behind *Hyper-Heuristics*.

Hyper-heuristics (HH) attempt to raise the level of generality at which optimisation methods operate. They can be thought of as heuristics to choose heuristics [18] or as search algorithms that explore the space of problem solvers. We briefly survey hyper-heuristics in the next section, we highlight the commonalities between hyper-heuristics and memetic algorithms in Sect. 1.2, and we look at theoretical issues in relation to hyper-heuristics in Sect. 1.3.

1.1 Hyper-heuristics

In the last few years there has been a dramatic increase in the amount of research done either specifically on

The authors acknowledge financial support from EPSRC (grants EP/C523377/1 and EP/C523385/1).

M. Bader-El-Den (✉) · S. Fatima
Department of Computer Science,
Loughborough University, Loughborough, UK
e-mail: M.B.Bader-EI-Den@Iboro.ac.uk

S. Fatima
e-mail: S.S.Fatima@lboro.ac.uk

R. Poli
School of Computer Science and Electronic Engineering,
University of Essex, Colchester, UK
e-mail: rpoli@essex.ac.uk

hyper-heuristics, or on topics related to hyper-heuristics. In general, the input to any HH framework is a set of heuristics (in most cases these are handcrafted heuristics). The output produced may vary from one framework to another depending on how the framework deals with low-level heuristics.

Different techniques can be classified into three main classes [5] which we briefly review in the following sub-sections.

1.1.1 Heuristic selection frameworks

In this class of hyper-heuristics, the system is provided with a set of preexisting heuristics for solving a certain problem, and it tries to discover the best sequence of application for these heuristics in order to find a solution. In other words, the hyper-heuristic system determines which heuristics to use at different stages in the search process. Many HH techniques fall in this class. Techniques used to build HH frameworks of this type include tabu search [14], case-based reasoning [17], genetic algorithms [25], ant-colony systems [13], and even algorithms inspired to marriage in honey-bees [1].

1.1.2 Heuristic-generating frameworks

Instead of trying to find a good order in which to use the handcrafted input heuristics, the aim here is to discover new heuristics based on the characteristics, behaviours and other attributes of the input heuristics. We will use this approach in this paper to evolve exam timetabling heuristics via genetic programming [34,45]. This approach has previously been used to evolve Boolean satisfiability heuristics [5,30] and specialised one-dimensional bin packing heuristics [11,12]. Also, in [43] a grammar-based genetic programming system was used to evolve multi-objective induction rules, which could be used in the classification of any data sets.

1.1.3 Template-based HH frameworks

In this form of hyper-heuristic the system produces (meta)heuristics by specialising a generic template. The specialisation can take the form of one or more evolved components, which can modify the behaviour of the metaheuristic or heuristic. An example on this approach is the work by [46], where novel algorithms for the one-dimension offline bin packing problem (where the objects to pack are known in advance) were evolved via GP.

1.2 Memetic algorithms and hyper-heuristics

Memetic Algorithms (MAs) are evolutionary population-based algorithms firstly introduced in [39] that combine local search heuristics with genetic operators (crossover, mutation, etc.). MAs have been shown to be orders of magnitude faster than traditional genetic algorithms for some problem domains.

It is particularly important in relation to this paper to mention the work by Krasnogor [35,36] who introduced the concept of *self-generation* in memetic algorithms. In a self-generation MA the evolutionary algorithm evolves concurrently both solutions and the low-level local search moves needed to solve the problem of interest. In tests with the *NK-Landscape* and the *Maximum Contact Map Overlap* problem, this combination of local search strategies and solutions allowed a memetic algorithm to outperform genetic algorithms and MAs with human-designed local searchers [36].

There are strong links between HHs and MAs. A HH could be seen as a version of an adaptive memetic algorithms [42] where the memes are considered as low-level heuristics and the HH as the strategy that manages the choice of these memes at any given time, depending on the characteristics of the memes and the search space. In [40] MAs were categorised into three generations, placing standard HH frameworks in the second generation. The framework presented in this paper can be placed in the third generation, alongside co-evolutionary [51] and self-generating [36] memetic methods.

1.3 Hyper-heuristics and no-free lunch

An important question one needs to ask when using search algorithms to explore the space of heuristics, problem solvers and other search algorithms is whether this meta search will be subjected by the same constraints as ordinary search. In particular, the question is whether there is hope to be successful at automatising the process of heuristic production via hyper-heuristics in the light of the No-Free Lunch (NFL) theory for search [58].

In very simple terms, the original version of NFL states that, if one evaluates algorithms on all possible problems, the average performance of all search algorithms is the same. Note that random search is one such algorithm. So, effectively no algorithm can beat random search on all the possible problems. More recent research [50] has shown that NFL is valid “if and only if” the set of the fitness functions one is interested in optimising is closed under permutation (a reordering of the fitness values associated to the elements of the search space).

An interesting recent development [44] indicates that automatising the process of heuristic production with hyper-heuristics may actually be possible. In particular, [44] showed is that in the case of hyper-heuristics, in many practical situations, the set of fitness functions is not closed under permutation and so NFL may not apply. In other words, the search

for superior hyper-heuristics is not necessarily doomed to fail.

2 The exam timetabling problem

The exam timetabling problem is a common combinatorial optimisation problem which all educational institutions need to face. Although the problem's details tend to vary from one institution to another, the core of the problem is the same: there is a set of exams (tasks), which have to be assigned to a predefined set of slots and rooms (resources).

2.1 Formal statement of the problem

We use the following formulation for the exam timetabling problem. The problem consists of a set of n exams $E = \{e_1, \ldots e_n\}$, a set of m students $S = \{s_1, \ldots s_m\}$, a set of q time slots $P = \{p_1, p_2, \ldots p_q\}$ and a registration function $R : S \to E$, indicating which student is attending which exam. Seen as a set, $R = \{(s_i, e_j) : 1 \le i \le m, 1 \le j \le n\}$, where student s_i is attending exam e_j. A scheduling algorithm assigns each exam to a certain slot. A solution then has the form $O : E \to P$ or, as a set, $O = \{(e_k, p_l) : 1 \le k \le n, 1 \le l \le q\}$.

The problem is similar to the graph colouring problem but it includes extra constraints, as shown in [55] (more on this later).

2.2 Timetabling constraints

Constraints are categorised into two main types:

Hard constraints: Violating any of these constraints is not permitted since it would lead to an unfeasible solution.
Soft constraints: These are desirable but not crucial requirements. Violating any of the soft constraints will only affect the solution's quality.

All hard constraints are equally important, while the importance of soft constraints can vary.

Most constraints relate to the main entities of the problem: students, exams, rooms and slots. The most common constraints used in timetabling are the following:

Clashing: no students are to be scheduled to take exams in the same time slot.
Capacity: the number of students in an exam should not exceed the room's capacity.
Total capacity: this limits the total number of students in all rooms that can take exams in one slot.
Exam capacity: some institutions set a maximum number of exams that can take place in the same slot.
Exam availability: some exams may need to be set in certain slots.
Room availability: this may limit the use of some rooms in certain sessions.
Pairwise exam constraints: some exams may need to satisfy pairwise scheduling constraints.
Exam/room compatibility: some exams may require rooms with special resources.

There is no clear-cut distinction between soft and hard constraints: a constraint that is soft for one institution could be considered hard by another. For example, the constraint that students cannot attend two exams in the same slot is hard for most institutions. However, some other institutions may accept violations of this if only few students are affected, because a special exam session can be set up for those students. Also, an institution with a large number of rooms and a large number of staff relative to the number of students may not care much about the number of exam that could take place in parallel at the same time. On the other hand, a small institution with a limited number of rooms or staff in relation to the student population may give this constraint a much greater importance.

2.3 Cost functions

Usually a cost function is used to calculate the degree of undesirability of violating each of the soft constraints. This is particularly convenient because, in most cases, changing soft constraint combinations and the importance (weight) of constraints does not require modifications in the timetabling algorithm itself. Only updating the cost function that evaluates the quality of solutions is required.

The following simple function was used in [20] to evaluate the cost of constraint violations:

$$C(t) = \frac{1}{S} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} [w(|p_i - p_j|)a_{ij}] \tag{1}$$

where N is the total number of exams in the problem, S the total number of students, a_{ij} is the number of students attending both exams i and j, p_i is the time slot where exam i is scheduled, $w(|p_i - p_j|)$ returns $2^{5-|p_i-p_j|}$ if $|p_i - p_j| \le 5$, and 0 otherwise.[1] We will adopt this cost function in this work. Of course, solutions with a lower cost are more desirable.

[1] This means that in the most undesirable situation, i.e., when a student has two exams scheduled one after the other, i will increase the cost function by a large value, namely $2^{5-1} = 16$. This factor rapidly decreases (following a negative exponential profile) as the size of the gap between exams increases.

3 Related work

Many different techniques have been used to tackle the timetabling problem. These include: constraint-based methods, graph-based methods, population-based methods, hyper-heuristics, multi-criteria optimisation and techniques based on local search. For space limitations in this section we will review the three classes of timetabling techniques which are most tightly related to our work: graph-based methods, hyper-heuristics and evolutionary techniques.

3.1 Graph-based methods

As mentioned in the previous section, the timetabling problem could be viewed as a graph-colouring problem, but with constraints [55]. As illustrated in Fig. 1a, each exam can be seen as a node in a graph and clashes between pairs of exams can be seen as edges connecting corresponding nodes. Time slots can then be seen as colours we want to use to label the nodes. Figures 1b–1d illustrate three possible solutions to the problem in Fig. 1a.

The similarity between timetabling and graph-colouring allows the use of graph-colouring heuristics in solving timetabling problems. Graph colouring heuristics are constructive heuristics. The corresponding timetabling heuristics work by first ranking exams in terms of how difficult to schedule they are. Then they simply assign exams to slots one after another starting with most difficult exams. Most ranking strategies are inspired by graph colouring heuristics. These include: Largest Degree [8], Largest Enrolment [59], Saturation Degree [7], Colour Degree [20] and Largest Weighted Degree [20]. These heuristics will be discussed in more detail in Sect. 5.1. [20] provides a performance comparison between the above heuristic on a number of different graph colouring and exam timetabling problems and a comparison between heuristics with and without backtracking.

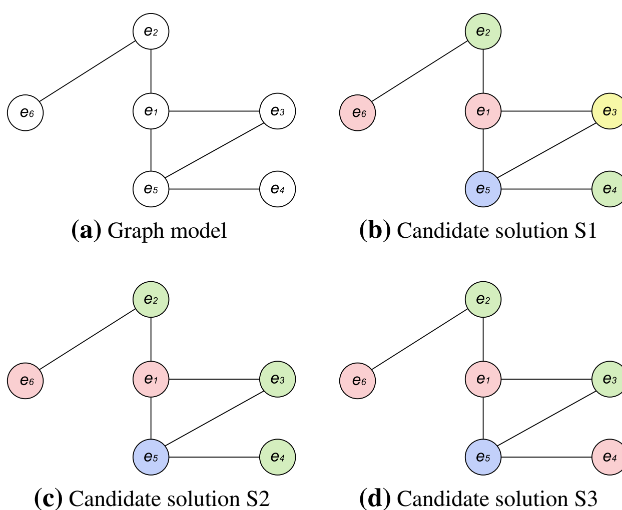

Fig. 1 A graph model for a sample exam timetabling problem **(a)**, along with three candidate solutions for it **(b–d)**. Solutions in **(c)** and **(d)** will typically be considered superior since they require the use of only three time slots (colours), while **(b)** requires four

In [23] a neural network framework for timetabling was presented. The system dynamically changes the ordering criteria as the solution is constructed through the use of a neural network which updates the difficulty ranking of each of the remaining exams based on the current state of the solution. The neural network's training set was constructed by storing the intermediate timetables obtained during the application of three graph-based heuristics.

Fuzzy techniques were used in [4] to blend different graph based heuristics. In each experiment two out of three graph-based heuristics were combined using a fuzzy system as follows. The ranking of each exam was calculated first using each of the two heuristics. These values were then fuzzified using three fuzzy membership functions (small, medium, high). By using fuzzy rules and defuzzification, a new rank was calculated for each exam. Different shapes for the membership function were tested and compared. Tuning the fuzzy membership functions was required on each test case in order to get competitive results.

3.2 Evolutionary methods

The genetic algorithm (GA) is one of the most frequently used and successful evolutionary algorithms in the area of timetabling. In general, there are two ways of using GAs for timetabling. In the first approach, the GA works directly in the solution space, i.e., each chromosome represents a timetable. One important issue in this approach is how to represent a timetable. Different techniques have been investigated through the years: a recent survey can be found in [49] (which an updated version of [22]). In the second approach, the GA works as a hyper-heuristic framework, i.e., the chromosome represents a sequence of heuristics to be applied for finding solutions rather than simply representing a solution. In most cases the length of the chromosome string is equal to the number of events to be scheduled and each heuristic in the string indicates how the corresponding event should be scheduled. This approach could also be seen as a memetic algorithm. In the remainder of this section we will survey techniques based on the first approach, while the second approach will be further discussed in the next section, alongside other hyper-heuristic techniques.

An investigation on the use of GAs in timetabling was presented in [47], where the differences in the performance of different algorithms around phase-transition regions were studied. The work showed that some simple evolutionary algorithms could outperform hill-climbing-based algorithms in regions associated with certain phase transitions, but not others. A continuation of this study was presented in [48].

This showed that using a direct mapping in a GA for timetabling is not a very good approach. The study highlighted the failure of a number of algorithms, including GAs, in solving some moderately constrained instances, while GAs were able to solve all the lightly and heavily constrained instances.

In [28], a GA representation based on the grouping character of the graph colouring problem was presented. The fitness function used to guide this Grouping GA was defined on the set of partitions of vertices. The algorithm was applied to a set of hard-to-colour graph examples and some real-world timetabling problems. Results were not very competitive with state of the art heuristics. However, the algorithm required much less computation power than other similar methods.

A GA was used in [53]. This was based on a new encoding system, called Linear Linkage Encoding (LLE) [27], which differs from the traditional Number Encoding system for graphs. The authors developed special operators to suit the new LLE encoding. Results showed that LLE is promising for grouping problems such as graph colouring and timetabling.

A hybrid multi-objective evolutionary algorithm was presented in [24]. The framework was used to tackle the uncapacitated exam proximity problem. Traditional genetic operators were replaced by local-search ones, based on a simplified variable neighbourhood descent metaheuristic. A repair operator was introduced to deal with unfeasible timetables. The framework did not require tuning across a number of test cases.

3.3 Hyper-heuristics methods

Tabu Search hyper-heuristics for timetabling were used in [15,31,33]. The main idea behind Tabu Search is to avoid repeating the evaluation of recently visited solutions. This is done by using a short-term memory structure that stores such solutions.

In [33] the authors employed Tabu Search as a high-level metaheuristic to search through a space of heuristics for university course timetabling and nurse rostering problems. The heuristics used in the study were improving heuristics that work on a previously constructed solution. In [15], instead, the low-level heuristics used were constructive heuristics (graph colouring heuristics). A Tabu Search approach was employed to search for permutations of graph heuristics, which were used for constructing timetables in exam and course timetabling problems.

The representation of the solutions in these approaches as a string of heuristics is similar to the representation used in [52] where a GA was used to find combinations of heuristics that perform well on certain problems. In that work each individual was represented using multiple strings. Strings had the same length as the number of exams in the problem instance. Each entry in the first string represented a code indicating which exam selection heuristic to use. The main difference between the two approaches is in how they move in the search space. In the Tabu Search based methods mentioned above a number of local search heuristics were used beside random moves, while in the GA algorithm proposed in [52], crossover and mutation were used.

Case Based Reasoning (CBR) was used in [17] in order to construct timetabling solutions. Heuristics that worked well in previous similar situations were memorised in a case base and were retrieved for solving the problem at hand. Knowledge discovery techniques were employed in two distinct scenarios. Firstly they were used to model problem solving situations along with specific heuristics for the targeted problems. Secondly they were used to refine the case base and discard cases which proved not to be useful.

A comparison between different hybridisation techniques for graph-based timetabling heuristics was presented in [10]. An empirical study analysing seven heuristic selection approaches, combined with five acceptance criteria was presented in [6]. The study reported that different combinations of selection and acceptance heuristics worked well on the tested instances.

A combination of hill-climbing and memetic algorithms transplanted into a hyper-heuristic framework was presented in [29]. A comparison between the proposed techniques and a self-adaptive hyper-heuristic with different selection and acceptance heuristics was provided. The study showed that a memetic algorithm hyper-heuristic which used a single hill climber at a time performed the best among the other tested techniques.

The authors in [3] used a perturbation-based algorithm for hybridisation among a number of parametrised heuristics. The approach was tested on a set of highly constrained exam timetabling problems.

4 A genetic programming hyper-heuristic framework

In this paper we will use a grammar-based Genetic Programming Hyper-heuristic framework (GPHH) for evolving constructive heuristics for timetabling, focusing in particular on exam timetabling. The system is based on a grammar derived from a collection of graph colouring heuristics that have previously been shown to be effective in constructing timetables. Also, the grammar contains slot allocation heuristics as we will describe in detail in the next section.

The main advantage of GPHH method over other HH methods is that GP allows us to make use of conditional branching, looping and other components. So, not only it can find different combinations of heuristics that perform well on a problem, it can also discover new kinds of heuristics with no human interaction.

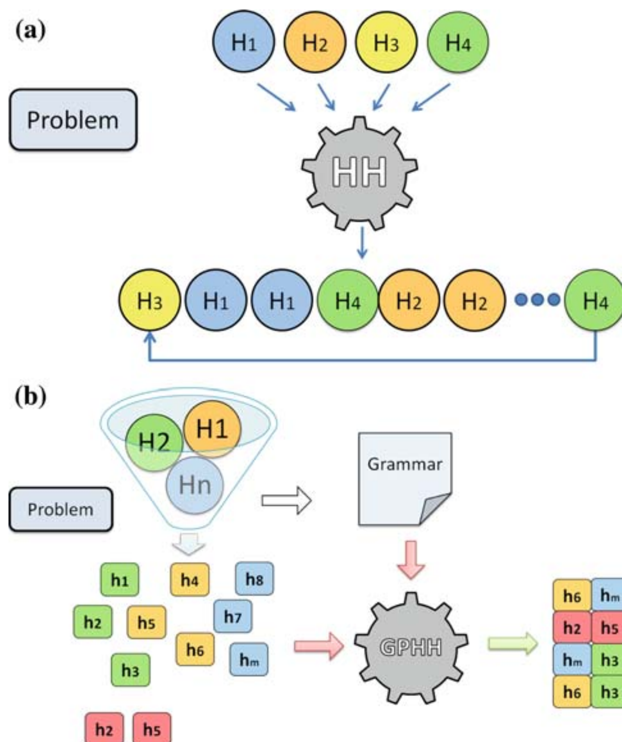

Fig. 2 The structure of a typical HH framework **(a)** and of GPHH **(b)**

Figure 2 shows a comparison between the GPHH structure and standard HH frameworks, where in standard HHs the target is to find a good sequence of application of different input heuristics. The application of GPHH requires the following steps:

Heuristic selection: a number of hand-crafted heuristics are selected.

Decomposition and grammar: the selected heuristics are decomposed into basic components. A grammar showing how these components could be combined together is then constructed.

Evolution: GP is used to evolve heuristics using the grammar.

We discuss these steps in more detail in the following sub-sections.

4.1 Heuristic selection

In the last decades heuristics have been widely used and have successfully contributed to the solution of a number of combinatorial optimization problems. As a result, for almost every such problem, scientists have designed and tested a large number of specific heuristics. In GPHH we simply start by selecting a suitable set of hand-crafted heuristics, which have previously been presented in the literature as solvers for the targeted problem.

Naturally these heuristics differ from one another performance-wise. In GPHH we consider heuristics with different performance levels, including some that are outperformed by other heuristics in many scenarios. The reason is that these heuristics are not used in GPHH as they are. Instead, they are decomposed and their components are used, as explained in the following sections. By not limiting ourselves to a small set of high-performance heuristics we provide the system with a richer variety of components. This diversity plays an important role in discovering and evolving new heuristics. For example, in some cases one or more components from a low-performance heuristic, when combined with one or more components from a high-performance heuristic, may result in a heuristic that outperforms both heuristics [41]. Indeed, various recently developed hand-crafted heuristics are hybridisations, compositions or combinations of previous heuristics.

4.2 Decomposition and grammar

In GPHH, after selecting a number of suitable hand-crafted heuristics, instead of directly feeding these heuristics into the hyper-heuristic system, the heuristics are first decomposed into their basic components. During the decomposition process, it is important not to lose information on how these components were connected with each other. We capture this information using a grammar which allows GPHH to create valid combinations of the components of heuristics.

To illustrate how the idea of decomposing heuristics into smaller components would work, let us consider a slightly simpler problem: the task of trying to pack a number of differently sized boxes into a truck. It is difficult to find the best way of packing, yet people have invented heuristics which allow them to do a reasonable job most of the time. These include the "Largest First" heuristic which suggests to load the large boxes first and then try to fit the smaller ones in the remaining gaps. Another heuristic might select the "Largest First" strategy with probability p, while it might use a "Smallest First" strategy or would select a random box with probability $1 - p$. One can see that these different heuristics can be decomposed into more basic components, such as selection, conditional branching, conditions, etc.

GPHH is a generic tool, in the sense that given a grammar for heuristics and an appropriate fitness function it can evolve solutions to a variety of problems. However, in general it is not possible to formalise the heuristic decomposition process leading to the grammar required by GPHH, since this varies from one problem to another and from one set of heuristics to another. However, it is possible to highlight some of the important points that one should keep in mind while constructing a grammar for GPHH. First of all, a good understanding of the problem is essential. Full awareness of the heuristics

behaviour and their characteristics is also crucial. Another important point is to keep a balance between the flexibility of the grammar and the search space size: the more detailed the grammar is, the freer GPHH to evolve radically new heuristics. On the other hand, adding more details, options, and flexibility into the grammar may correspondingly increase the size of the search space that GPHH needs to explore, with a corresponding demand for computational resources and time to find good new heuristics.

4.3 Evolution

GPHH uses a form of genetic programming [34,45] to evolve new heuristics. Pseudo-code describing the evolution process in GPHH is shown in Algorithm 1.

We should note that the ordinary initialization, crossover and mutation operations used in GP can easily generate invalid (ungrammatical) individuals. There are multiple ways of solving this problem. Firstly, one can use a function that repairs invalid individuals. However, this is not easy to achieve in complex GP representations. Moreover, this is known to cause biases in the search. An alternative is enforcing constraints on the GP individuals through the use of a type system as was done in strongly typed GP [38]. However, strongly typed GP is not always effective in modelling complex constraints that are not naturally based on types [45]. Another way to model constraints is through a grammar which is capable of presenting both *type* and *syntactic* constraints efficiently. For example, in the work presented in [56,57] the grammar was integrated within GP through the use of *derivation trees* as members of the population.

Algorithm 1 Evolutionary process in GPHH

Randomly initialize the population G using the Grammar.
for $i = 1$ to noGenerations **do**
  for $j = 1$ to noIndividuals **do**
    for $k = 1$ to noProblemsInTrainingSet **do**
      apply individual(j) from G on instance(k)
    end for
    calculate fitness for individual(j)
  end for
  create new empty generation G'
  for $m = 1$ to crossoverRatio*100/2 **do**
    select two individual from G based on fitness {parents}
    crossover the individuals creating two new individuals {children}
    insert new individuals in G'
  end for
  for $m = 1$ to mutationRatio*100 **do**
    select one individual from G based on fitness
    mutate the individual
    insert new individual in G'
  end for
  $G = G'$
end for

To illustrate this representation, let us consider the following example grammar for generating simple `if` statements:

```
<s>    ::=  if (<con>) then <exp> [else <exp>]
<con>  ::=  <exp> ≤ <exp> | <exp> ≥ <exp>
<exp>  ::=  <var> | <exp> <op> <exp>
<op>   ::=  +  |  −  |  ×  |  ÷
<var>  ::=  x  |  y  |  z
```

An example of a GP derivation tree, using the abovementioned grammar, is shown in Fig. 3. The evolved sentence (program) is represented by the leaves of the trees (read from left to right), while the internal nodes represent how these leaves are derived. The benefits of derivation trees are that

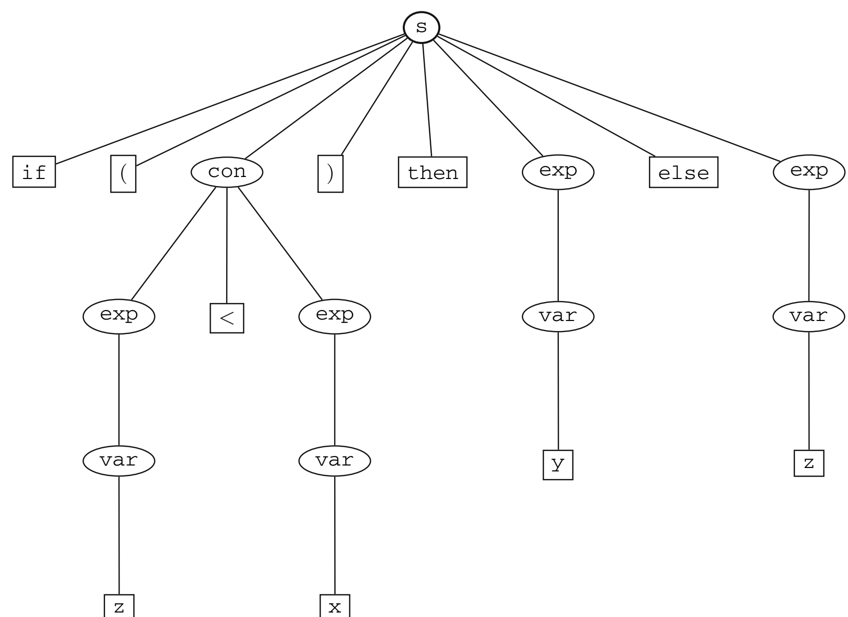

Fig. 3 A derivation tree representing the program `if (z < x) then y else z`

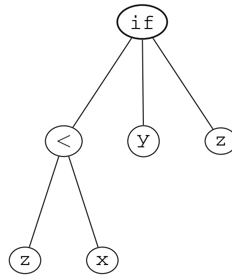

Fig. 4 The individual in Fig. 3, but represented as a normal GP tree

they are easy to initialise and that performing genetic operations is straightforward. For example, within crossover all one needs to ensure is that only sub-trees rooted at identical internal nodes are swapped. On the other hand, because information concerning how each leaf is generated is kept in the tree, this causes the tree size to be relatively large when compared to a standard GP representation of the same individual. This is illustrated in Fig. 4 which shows the same program as in Fig. 3 but represented in the normal GP style. Moreover, executing a derivation tree is more complex than executing an ordinary GP: first the evolved sentence needs to be gathered from the leaves, then it is transformed into a tree, and finally it is executed.

For these reasons, in GPHH a new, hybrid way of using grammars in GP was introduced. The representation used is the same as ordinary GP (see example in Fig. 4). However, a special initialization process that directly generates GP trees from the given grammar ensures all initial individuals are valid.

Because in standard GP trees information about how the nodes are generated from the grammar is not available, we also use a new specialised crossover operator which, after selecting a crossover point, uses the grammar (in a reverse way) to generate the derivation sequence for that crossover point "on the fly". This makes it possible to then efficiently check the compatibility between crossover points. A similar approach is used in mutation. The implementation details of these operators are not discussed here because of their complexity.

The main benefits of GPHH's way of using grammars over standard derivation trees are the following:

- GPHH requires much smaller trees to represent programs. This is particularly beneficial when using GP for evolving complex programs, since their derivations trees are potentially very large.
- GPHH trees are much easier and faster to execute since they are already in the appropriate form for recursive interpretation. Saving computation time is very important aspect in GP, particularly when it is used as a HH since the evaluation of evolved heuristics is very time-consuming.

5 GPHH for exam timetabling

In this section we customise GPHH to the exam timetabling problem. More specifically we describe the choice of heuristics we made (Sects. 5.1, 5.2), we provide the grammar used for this problem class (Sect. 5.3), we describe the constraints and fitness function we used (Sect. 5.4) and, finally, we provide details of the GP set up adopted for our evolutionary runs (Sect. 5.5).

5.1 Exam selection heuristics

The heuristics used within GPHH in this paper are the constructive graph-colouring heuristics proposed by Carter et al. in [20], namely: Largest Degree-based selection (LD), Largest Enrolment-based selection (LE), Least Saturation-degree-based selection (SD), Largest Weighted-degree-based selection (LWD), and Random Selection (RS). These heuristics have been very successfully used in constructing timetables directly [20] and also withing other hyper-heuristics [4,15] and other frameworks for constructing timetables.

Two exams are considered to be in conflict with each other if they cannot be scheduled in the same slot. Two exams are conflicting if there are students registered for both. In the largest degree heuristic, LD, exams with the most conflicts are selected first. The largest enrolment heuristic, LE, selects first the exams with the largest number of students. The saturation degree heuristic, SD, selects exams with the least number of available slots first. This is the only heuristic that involves updating. That is, assigning an exam to a slot will make this slot unavailable for all other exams in conflict with this exam and, so, the count of available slots for them needs to be decreased by one. Largest weighted degree, LWD, is the same as the LD heuristic, but if there is more than one exam with the same number of conflicts, the tie is broken in favour of the exam with most students. Random Selection, RS, is the simplest heuristic in this group: it selects a random exam from the list of unscheduled exams. This heuristic brings a degree randomness into the scheduling process, which, in some cases, helps produce better results. On the other hand, randomness may cause the system not to produce the same results when running the same group of heuristics multiple times. To overcome this problem typically users run the same heuristics combination more than once in order to ensure robustness.

Figure 5 shows a fragment of the grammar used in GPHH listing our exam selection heuristics. The grammar is designed so that different combinations of heuristics can be nested together. So, we have implemented some selection heuristics in such a way that, rather than returning a single element from an input list of exams (an

```
<exm>    ::=  random <eList> |
              first <eList>
<eList>  ::=  max-conflict <eList> |
              least-slot <eList> |
              max-students <eList> |
              all-exams
```

Fig. 5 Fragment of the GPHH grammar listing the set of heuristics used for selecting exams

eList), they may return a list of exams. The LD, LE and SD heuristics, which are represented in the grammar as max-conflict, max-student and least-slot, respectively, do this. The primitive all-exams returns a list of all exams not scheduled yet. The primitive random implements the SD heuristic. The LWD heuristic has no direct representation in the grammar as it can simply be obtained by nesting other heuristics as follows: random(max-student(max-conflict(all-exams))). In this statement, all-exams returns a list of all exams not scheduled yet to max-conflict. If there is more than one exam with the highest number of conflicts, this primitive returns a list including multiple elements. The tie is then resolved by max-student in favour of the exam(s) with the highest number of students. If there is still a tie, this is resolved randomly (by random).

The ability of recursively composing heuristics provided by our grammar gives GPHH the flexibility to evolve fairly sophisticated heuristics, such as the LWD exemplified above.

5.2 Slot selection heuristics

The slot selection heuristics assign one of the available slots to a previously selected exam. The following heuristics are used in GPHH for this: Least-Cost-based selection (LC), Least Blocking-based selection (LB), Busiest-based selection (BU), Least-busy-based Selection (LU), and Random Selection (RS).

The LC heuristic selects the slot that causes the least increase in the solution's cost. The LB heuristic selects the slot which causes the least decrease in total number of available slots for all other unscheduled conflicting exams. The idea behind the BU heuristic is to select the slot which is the busiest in terms of other exams, in order to keep as much space as possible available for the remaining exams. This can be particularly effective in breaking ties between slots that have the same cost. The LU heuristic is the opposite of BU: it selects the slot which is the least occupied with other exams. The RS heuristic simply selects a random slot.

The part of the GPHH grammar representing these slot-selection heuristics is shown in Fig. 6. The grammar allows these heuristics to be nested, like the exam selection heuristics, via the use of slot-lists.

```
<slt>    ::=  random <sList> |
              first <sList>
<sList>  ::=  least-cost <sList> |
              least-busy <sList> |
              most-busy <sList> |
              least-blocking <sList> |
              all-slots
```

Fig. 6 Part of the GPHH grammar which is responsible for selecting slots

```
S        ::=  assign <exm> <slt>

<exm>    ::=  random <eList> |
              first <eList>

<eList>  ::=  max-conflict <eList> |
              least-slot <eList> |
              max-students <eList> |
              all-exams |
              if <cond><eList><eList>

<slt>    ::=  random <sList> |
              first <sList>

<sList>  ::=  least-cost <sList> |
              least-busy <sList> |
              most-busy <sList> |
              least-blocking <sList> |
              all-slots |
              if <cond><sList><sList>

<cond>   ::=  <prob> | <size>

<size>   ::=  vSmall | small |
              mid | large

<prob>   ::=  20 | 40 | 50 |
              70 | 90
```

Fig. 7 The complete GPHH grammar for the evolution of exam timetabling heuristics

5.3 GPHH full grammar

The full GPHH grammar developed for exam timetabling is shown in Fig. 7. In addition to the features previously described, the grammar contains two types of conditional branching instructions. The first type is probabilistic branching, based on various fixed probability (namely 20, 40, 50, 70 and 90%). The second is based on how far the evolved heuristic has proceeded in the construction of a timetable. This is achieved through the use of the following specialised terminals: vSmall, which returns true if less than 25% of the exams are scheduled, small which is true from 25 to 50%, mid from 50 to 75% and large when more than 75% of the exams are scheduled.

We added this second form of branching because the performance of fixed heuristics varies throughout the

construction of a solution. Some may be efficient in the early stages, while others are more effective at refining nearly complete timetables. For example, the SD exam selection heuristic, which selects the exam with the least number of available slots, may not be efficient at the beginning of the timetable construction, when most or all of the slots are still empty and available for exams. At this stage, a heuristic such as the LD exam selection heuristic may be more effective because it selects exams based on the number of conflicts with other exams. On the other hand, SD may be more effective at later stages where many slots are blocked with other exams, and there are some exams with very few slots available. Consequently, at that stage is reasonable to attempt to schedule these exams first.

Using a form of branching that considers how far the evolved heuristic have proceeded in constructing a timetable allows GPHH to decide when and how to use different heuristics, thereby providing extra flexibility over systems (such as those based on Tabu search or on GAs), where heuristics are used sequentially in the fixed order prescribed by the representation.

5.4 Constraints and fitness

In the application of GPHH to the exam timetabling problem we adopted the most common hard and soft constraints in the literature in order to be able to compare our results with the largest number of previously published results.

The hard constraints we consider in this paper are represented by the "conflicts" of scheduling two exams with common students into the same time slot. The soft constraints are concerned with spreading out each student's exams over the timetable so that students will not have to sit exams that are too close to each other. The objective is to schedule all of the exams into the time slots, while minimising the cost on the violations of the soft constraints.

In this work the cost of constraint violation was calculated using Carter's cost function in Eq. 1.

The fitness function we used is the following modification of Eq. (1):

$$f = \left[\frac{1}{S}\sum_{i=1}^{M-1}\sum_{j=i+1}^{M} w(|p_i - p_j|)a_{ij}\right] + \alpha(N - M) \qquad (2)$$

where, again, N is the total number of exams in the problem and S is the total number of students, M is the total number of exams that have successfully been scheduled, and α is constant. The objective is to minimise f (i.e., the lower the fitness value the better the individual).

The first part of the Eq. (2) is almost the same as the cost function in Eq. (1). The second part adds an extra penalty for each exam the heuristic (individual) has not been able to schedule (note that $N - M$ is the number of unscheduled exams). Even though solutions with unscheduled exams are considered to be invalid solutions, this extra penalty for unscheduled exams is introduced to give GP a better ability to differentiate between individuals based on how close they are to outputting a complete timetable.

5.5 Genetic programming setup

The GP system initialises the population by randomly drawing nodes from the function and terminal sets. As one can easily infer from the grammar in Fig. 7, our function set includes the primitives `assign`, `random`, `first`, `max-conflict`, `least-slot`, `max-students`, `if`, `least-cost`, `least-busy`, `most-busy` and `least-blocking`. The GP terminal set, instead, includes the primitives `all-exams`, `all-slots`, `vSmall`, `small`, `mid`, `large`, `20`, `40`, `50`, `70` and `90`. Initialisation is performed by a specialised version of the GROW method [34,45] which takes the grammar into account. For example, the root of the tree must be the function `assign` and this is not allowed anywhere else.

After initialisation, the population is manipulated by selection, reproduction, crossover and mutation operators. More specifically, GPHH uses tournament selection, with tournament size of 5 (reselection is permitted). We use reproduction (cloning) with a rate of 0.1. The crossover rate is 0.8. Crossover is a specialised form of sub-tree crossover which works as follows. A random crossover point is selected in the first parent, then the grammar is used to select the crossover point from the second parent. This is randomly selected from all valid crossover points. If no point is available, the process is repeated again from the beginning until crossover is successful. Mutation is applied with a rate of 0.1. This operates by selecting a random node from the parent (including the root of the tree), deleting the sub-tree rooted there, and then regenerating it randomly, based on the grammar, as in the initialisation phase.

In our experiments we used population of size 50 evolved for 50 generations. Each individual was executed once. However, to reduce stochastic effects on runs, the best performing individuals were run for an extra two times. Individuals that had not been affected by any genetic operation were not evaluated again to reduce the computation cost. For each problem instance we performed 10 independent runs.

In other experiments we used larger populations, with sizes ranging between 500 and 1,000 individuals. Each individual was executed three times to reduce stochastic effects on its output. Again the best performing individuals were run for an extra two times. Naturally these runs where substantially more computationally expensive and, so, we were only able to do between 1 and 5 runs for each problem instance.

6 Test problems and results

For a HH framework to be practical and useful, the framework should ideally find reasonably good heuristics, reasonably cheaply and reasonably quickly. Reasonably good means that the evolved heuristics should perform well on the problem instance(s) it was evolved for. Relatively cheaply means that the evolved heuristics should not be more expensive in terms of effort, time, or cost than manually finding and tuning a good heuristics for the problem instance. Reasonably fast refers to the requirement that the heuristic should be evolved within a few hours, or at most a day. As we will illustrate in this section, GPHH achieves all this in relation to the exam timetabling problem.

We tested GPHH by applying it to one of the most widely used benchmarks in exam timetabling, against which many state-of-the-art algorithms have been compared in the past. The benchmark was first presented in [20]. Its characteristics are shown in Table 1. The size of the problems varies from 81 to 682 exams and from 611 to 18,419 students. Table 1 reports also the density of the conflict matrix (a_{ij}). This is given by the ratio of the number of conflicting exams over the total number of possible exam pair combinations.

Table 2 shows the performance of the GPHH approach against other state-of-the-art algorithms. The table shows the cost of the best solution found by each algorithm for each problem instance in the benchmark set. The cost is calculated using Eq. (1). In the case of GPHH, we report results

Table 1 Characteristics of the benchmark exam timetabling problems that were used in our experiments

Instance name	Exams	Students	Slots	Max. reg.	Matrix density
car91	682	16,925	35	1,835	0.13
car92	543	18,419	32	1,566	0.14
ear83	190	1,125	24	232	0.27
hec92	81	2,823	18	634	0.42
kfu93	461	5,349	20	1,280	0.60
lse91	381	2,726	18	382	0.60
sta83	193	611	13	237	0.14
tre92	261	4,360	23	407	0.18
uta93	184	2,750	10	1,314	0.80
york83	181	941	21	175	0.29

Each row shows the total number of exams, the number of students registered in at least one exam, the maximum number of slots available, the maximum number of students registered in one exams, and the conflict's matrix density for the problem instance (see text)

Table 2 The best results obtained by GPHH on benchmark exam timetabling problems against the best results reported in the literature

	car91	car92	ear83	hec92	kfu93	lse91	sta83	tre92	uta93	york83
Constructive										
GPHH (small population)	**5.12**	**4.46**	37.10	11.78	14.72	11.11	158.70	**8.62**	3.47	**40.56**
GPHH (large population)	**5.19**	**4.15**	37.20	11.96	14.54	11.17	158.63	**8.63**	3.43	**40.05**
Tabu [15]	5.36	4.53	37.92	12.25	15.2	11.33	**158.19**	8.92	3.88	41.37
Fuzzy [4]	5.20	4.52	37.02	11.78	15.81	12.09	160.42	8.67	3.57	40.66
Carter et al. [20]	7.1	6.2	**36.4**	**10.8**	**14.0**	**10.5**	161.5	9.6	3.5	41.7
Tabu-Multi-stage [15]	5.4	4.74	38.84	13.11	15.99	12.43	159.9	9.02	**3.04**	43.51
Improvement methods										
Abdullah et al. [2]	5.21	4.36	34.87	10.28	**13.46**	10.24	150.28	**8.13**	3.63	**36.11**
Burke and Newall 2002 [16]	4.6	**4.0**	37.05	11.54	13.9	10.82	168.73	8.35	**3.2**	36.8
Burke et al. [9]	**4.2**	4.8	35.4	10.8	13.7	10.4	159.1	8.3	3.4	36.7
Caramia et al. [19]	6.6	6.0	**29.3**	**9.2**	13.8	**9.6**	158.2	9.4	3.5	36.2
Casey and Thompson [21]	5.4	4.4	34.8	10.8	14.1	14.7	**134.7**	8.7	na	37.5
Di Gapero and Schaerf [31]	6.2	5.2	45.7	12.4	18.0	15.5	160.8	10.0	4.2	41.0
Merlot et al. [37]	5.1	4.3	35.1	10.6	13.5	10.5	157.3	8.4	3.5	37.4

Figures represent the cost (computed with Eq. 1) associated to the best evolved timetable. The upper part of the table represent systems based on constructive heuristics only. The lower part of the table represents algorithms which use constructive heuristics followed by improvement heuristics. Figures in bold face represent the best result in each part (on a per-column basis)

Table 3 Performance statistics for GPHH with small populations

	car91	car92	ear83	hec92	kfu93	lse91	sta83	tre92	uta93	york83
Mean cost of timetable	5.15	4.48	37.62	12.04	14.82	11.28	159.03	8.69	3.48	41.09
Standard deviation	0.03	0.02	0.47	0.19	0.04	0.10	0.37	0.05	0.01	0.30
Best cost across runs	5.12	4.46	37.10	11.78	14.72	11.11	158.70	8.62	3.47	40.56
Worst cost across runs	5.18	4.51	38.50	12.48	14.85	11.50	159.77	8.78	3.50	41.50

for both small populations (50 individuals run for 50 generations) and large populations (either 500 or 1,000 individuals, run for between 50 and 100 generations). For GPHH with small populations figures represent the best results obtained within 10 independent runs (further statistical information about our runs will be provided below), while for the large populations they represent the best result obtained in between 1 and 5 independent runs.

The table is divided into two parts. The upper part contains GPHH and other algorithms that *only use construction heuristics* (the first two methods, other than GPHH, in this group use constructions heuristics only; the other two use construction with backtracking). The bottom part of Table 2 shows a group of algorithms that use *constructive heuristics followed by improvement heuristics*. Note that GPHH uses constructive methods only. So, clearly, it is unfair to compare GPHH with this second group of systems. However, we show the comparison here so that readers can see how close GPHH can get to the performance of improvement-based algorithms.

If we first focus on GPHH with small populations, we see that it outperforms the four algorithms (other than GPHH) in the first group on 40% of the instances (`car91`, `car92`, `tre92` and `york83`) and it is second best in 50%. In other words, when compared with algorithms using the same elementary heuristics, GPHH with small populations resulted best or second best in 90% of the test instances, thus showing a remarkable robustness. The situation is similar for GPHH with large populations.

GPHH has been able to outperform many of the algorithms in the second group too, despite it not using improvement heuristics. For example, on `car92` GPHH with large populations is the second best across all algorithms, while on `uta93` it is the fourth best overall.[2] Amazingly, GPHH with small populations is second best overall on `car91`.

While the results in Table 2 are very encouraging, since we report best results over multiple runs, one might wonder what sort of level of performance we could expect to obtain from GPHH in typical runs. In Table 3 we report statistics for the cost associated to the best-of-run timetables evolved by GPHH with small populations across the problems in our benchmark set. As one can see standard deviations are extremely small, indicating that one can get excellent timetables with GPHH even if the system is run only once or very few times on a problem.

We should note that certain problems in our benchmark set are hard, in the sense that only relatively high-cost solutions can be found, by all algorithms. For example, on `sta83`, `york83` and `ear83` all algorithms find solutions with costs which are on average one or more orders of magnitude bigger than for other problems. To understand why this is the case one needs to take a closer look at the characteristics of the problems in the benchmark set. Table 4 shows histograms of the number of students registered for 1, 2, etc. exams as well as the average number of exams per student in each problem instance. Noting how widely the number of exams per student varies from instance to instance one can easily understand why `sta83` is so difficult. It this problem, with only one exception, all students attend a very large number of exams. So, the problem is badly over-constrained and all heuristics exhibit relatively poor results. Note that `york83` and `ear83` have the second (7.21) and third (5.95) highest mean number of exams per student across all problems, respectively. It is then not surprising that no solver can find low-cost solutions for these instances.

It is interesting to study how the performance of GPHH varies in relation to other solvers as the difficulty of the problem changes. Figure 8 shows a plot of the rank of GPHH with small populations in relation to the other exam timetabling algorithms listed in Table 2 *versus* the mean number of exams per student across all problem instances in our benchmark set. In this figure lower ordinate values are better. For example, the data point at coordinates (3.36, 3) means that GPHH was third best overall on a problem characterised by a mean number of exams per student of 3.36.

As one can see from the figure, GPHH performs particularly well on problems with a relatively low number of exams per student. This is not surprising since the algorithm does not use timetable refinement procedures, which are particularly beneficial in the more heavily constrained problems. This suggests that in future extensions of this work we could obtain further significant performance improvements by extending the GPHH's grammar to include refinement heuristics.

[2] We should note that for many of the results in Table 2, cost values were published with only one decimal figure. If we assume that such values were obtained via rounding, and we round all values in the table to one decimal place, we find that on `uta93` GPHH with large populations is actually third best overall.

Table 4 The number of students attending 1, 2, 3, ... exams for each exam timetabling instance in our benchmark set

Number of exams	car91	car92	ear83	hec92	kfu93	lse91	sta83	tre92	uta93	york83
1	3,409	3,969	1	321	276	99	0	667	6,180	1
2	2,145	3,330	1	315	234	80	0	524	3,866	20
3	2,107	3,436	1	351	277	302	0	744	3,717	93
4	4,098	4,168	20	659	765	1,638	0	1,191	4,026	4
5	4,569	3,212	72	1,071	2,515	474	1	1,214	3,073	44
6	565	275	201	105	1,082	103	0	20	381	174
7	27	29	302	1	189	27	0	0	23	372
8	4	0	409	0	11	3	162	0	0	0
9	1	0	109	0	0	0	239	0	0	0
10	0	0	9	0	0	0	0	0	0	0
11	0	0	0	0	0	0	209	0	0	0
12	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	1
Mean number of exams per student	3.36	3.01	7.21	3.77	4.69	4.00	9.41	3.42	2.77	5.95

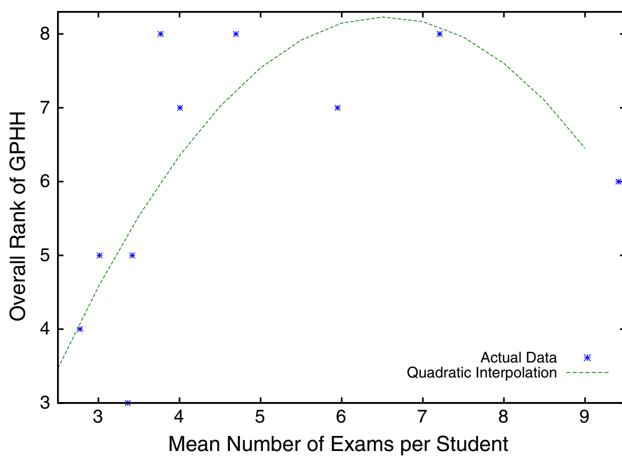

Fig. 8 Plot of the rank of GPHH with small populations in relation to the other exam timetabling algorithms listed in Table 2 *versus* the mean number of exams per student across all problem instances in our benchmark set. The *solid line* represents a quadratic polynomial interpolating the raw data

While the intrinsic difficulty of finding good (low-cost) timetable is by and large determined by the nature and number of the constraints imposed on the solutions, the computational load of fitness evaluations is effectively determined by the number of students and exams to be taken by each. So, relatively weakly constrained problems such as `car91` and `car92` are, in fact, one or more orders of magnitude computationally more demanding than problems such as `sta83` or `york83`. On the latter two problems runs of GPHH with small populations lasted less than 10 min each on an ordinary PC, while runs of `car91` and `car92` took around 4 h each. Other problems had run times in between these two extremes.[3]

If we look at these timing results together with the high performance levels obtained by GPHH with small populations and the reproducibility of the quality of its results across runs, we can see that the approach is entirely viable as GPHH is able to find reasonably good heuristics, reasonably cheaply and reasonably quickly. There is, therefore, hope that a future parallelisation of GPHH (e.g., through the use of GPUs) together with the use of a more extensive grammar should produce an even more effective system.

7 Conclusion

In this paper we have introduced a Grammar-based Genetic Programming Hyper-heuristic framework for evolving constructive heuristics for timetabling. The framework was tested on one of the most widely used benchmarks in the field of exam timetabling and compared with the best state-of-the-art approaches. Results show that the framework is very competitive with other constructive techniques, effectively outperforming other HH frameworks based on constructive heuristics and also some more sophisticated frameworks where constructive heuristics are combined with refinement ones. In future work, we will look at extending the grammar used in GPHH with further heuristics (e.g., to allow the evolution of heuristics that can refine as well

[3] Runs of GPHH with large populations took approximately 1 to 1.5 orders of magnitude longer.

as construct timetables) and to the possibility of reusing some of the heuristics evolved in this study as building blocks.

References

1. Abbass HA (2001) MBO: marriage in honey bees optimization—a haplometrosis polygynous swarming approach. In: Proceedings of the 2001 IEEE congress on evolutionary computation. IEEE Press, Seoul, Korea, pp 207–214
2. Abdullah S, Ahmadi S, Burke EK, Dror M (2007) Investigating ahuja-borlins large neighbourhood search approach for examination timetabling. OR Spectr 29(2):351–372
3. Ahmadi S, Barrone R, Chen P, Cowling PI, McCollum B (2003) Perturbation based variable neighbourhood search in heuristic space for examination timetabling problem. In: Selected papers from MISTA 2003, pp 155–171
4. Asmuni H, Burke EK, Garibaldi JM, McCollum B (2004) Fuzzy multiple heuristic orderings for examination timetabling. In: PATAT'04: The 4th international conference for the practice and pheory of automated timetabling, pp 334–353
5. Bader-El-Den MB, Poli R (2007) Generating sat local-search heuristics using a gp hyper-heuristic framework. In: Monmarché N et al (eds) Artificial evolution, vol 4926, pp 37–49
6. Bilgin B, Özcan E, Korkmaz EE (2006) An experimental study on hyper-heuristics and exam timetabling. In: PATAT'06: The 6th international conference for the practice and theory of automated timetabling, pp 394–412
7. Brélaz D (1979) New methods to color the vertices of a graph. Commun ACM 22(4):251–256
8. Broder S (1964) Final examination scheduling. Commun ACM 7(8):494–498
9. Burke EK, Bykov Y, Newall J, Petrovic S (2004) A time-predefined local search approach to exam timetabling problems. IIE Trans 36(6):509–528
10. Burke EK, Dror M, Petrovic S, Qu R (2005) Hybrid graph heuristics in hyper-heuristics applied to exam timetabling problems. In: Golden BL et al (eds) The next wave in computing, optimization, and decision technologies. Springer, Maryland, pp 79–91
11. Burke EK, Hyde MR, Kendall G (2006) Evolving bin packing heuristics with genetic programming. In: Runarsson et al (eds) Parallel problem solving from nature—PPSN IX, LNCS, vol 4193. Springer, Reykjavik, pp 860–869
12. Burke EK, Hyde MR, Kendall G, Woodward J (2007) Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In: GECCO '07: Proceedings of the 9th annual conference on genetic and evolutionary computation. ACM, New York, pp 1559–1565
13. Burke EK, Kendall G, Silva DL, O'Brien R, Soubeiga E (2005) An ant algorithm hyperheuristic for the project presentation scheduling problem. In: 2005 IEEE congress on evolutionary computation. IEEE press, pp 2263–2270
14. Burke EK, Kendall G, Soubeiga E (2003) A tabu-search hyperheuristic for timetabling and rostering. J Heuristics 9(6):451–470
15. Burke EK, McCollum B, Meisels A, Petrovic S, Qu R (2007) A graph-based hyper-heuristic for educational timetabling problems. Eur J Oper Res 176(1):177–192
16. Burke EK, Newall JP (2002) Enhancing timetable solutions with local search methods. In: PATAT'02: The 3rd international conference for the practice and theory of automated timetabling, pp 195–206
17. Burke EK, Petrovic S, Qu R (2006) Case-based heuristic selection for timetabling problems. J Sched 9(2):115–132
18. Burke KE, Kendall G, Newall J, Hart E, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. In: Glover et al (eds) Handbook of metaheuristics. Kluwer, Dordrecht pp 457–474
19. Caramia M, Dell'Olmo P, Italiano GF (2001) New algorithms for examination timetabling. In: WAE '00: Proceedings of the 4th International workshop on algorithm engineering. Springer, London, pp 230–242
20. Carter MW, Laporte G, Lee SY (1996) Examination timetabling: algorithmic strategies and. J Oper Res Soc 47:73–83
21. Casey S, Thompson J (2002) Grasping the examination scheduling problem. In: PATAT'02: The 4th international conference for the practice and theory of automated timetabling, pp 232–246
22. Corne D, Ross P, lan Fang H (1994) Evolutionary timetabling: practice, prospects and work in progress. In: Proceedings of the UK planning and scheduling SIG workshop
23. Corr PH, McCollum B, McGreevy MAJ, McMullan PJP (2006) A new neural network based construction heuristic for the examination timetabling problem. In: Procedings of 9th international conference parallel problem solving from nature—PPSN IX, pp 392–401
24. Côté P, Wong T, Sabourin R (2004) A hybrid multi-objective evolutionary algorithm for the uncapacitated exam proximity problem. In: PATAT'04: The 4th international conference for the practice and theory of automated timetabling, pp 294–312
25. Cowling PI, Kendall G, Han L (2002) An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. In: Proceedings of the 2002 IEEE congress on evolutionary computation. IEEE press, Washington, DC, pp 1185–1190
26. Di Gaspero L, Schaerf A (2003) Easylocal++: an object-oriented framework for the flexible design of local-search algorithms. Softw Pract Exp 33(8):733–765
27. Du J, Korkmaz E, Alhajj R, Barker K (2004) Novel clustering approach that employs genetic algorithm with new representation scheme and multiple objectives. In: DaWaK'04: Proceedings of the 6th international conference on data warehousing and knowledge discovery, vol 3181/2004. Springer, Heidelberg
28. Erben W (2001) A grouping genetic algorithm for graph colouring and exam timetabling. In: PATAT '00: The 3rd international conference for the practice and theory of automated timetabling. Springer, London, pp 132–158
29. Ersoy E, Ozcan E, Etaner-Uyar AS (2007) Memetic algorithms and hyperhill-climbers. In: Baptiste J et al (eds) MISTA'07: The 3rd multidisciplinary international scheduling conference: theory and applications, pp 159–166
30. Fukunaga AS (2002) Automated discovery of composite sat variable-selection heuristics. In: Proceedings of the18th national conference on artificial intelligence. AAAI, pp 641–648
31. Gaspero LD, Schaerf A (2000) Tabu search techniques for examination timetabling. In: PATAT'06: The 3rd international conference for the practice and theory of automated timetabling, pp 104–117
32. Gutin G, Karapetyan D (2009) A selection of useful theoretical tools for the design and analysis of optimization heuristics. Memet Comput 1(1):25–34
33. Kendall G, Hussin NM (2004) An investigation of a tabu search based hyper-heuristic for examination timetabling. In: Kendall G, Burke EK, Petrovic S (eds) Selected papers from MISTA 2003. Kluwer, Dordrecht
34. Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge
35. Krasnogor N (2002) Studies on the theory and design space of memetic algorithms. Ph.D. thesis, University of the West of England
36. Krasnogor N, Gustafson S (2004) A study on the use of "self-generation" in memetic algorithms. Nat Comput 3(1):53–76

37. Merlot LTG, Boland N, Hughes BD, Stuckey PJ (2002) A hybrid algorithm for the examination timetabling problem. In: PATAT'02: The 3rd international conference for the practice and theory of automated timetabling, pp 207–231
38. Montana DJ (1995) Strongly typed genetic programming. Evol Comput 3(2):199–230
39. Moscato P (1989) On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. Technical Report 826, Caltech concurrent computation program
40. Nguyen QH, Ong YS, Lim MH (2008) Non-genetic transmission of memes by diffusion. In: GECCO'08: Proceedings of the 10th annual conference on Genetic and evolutionary computation. ACM, New York, pp 1017–1024
41. Ong YS, Kean AJ (2004) Meta-lamarckian learning in memetic algorithms. IEEE Trans Evol Comput 8(2):99–110
42. Ong YS, Lim MH, Zhu N, Wong KW (2006) Classification of adaptive memetic algorithms: a comparative study. IEEE Trans Syst Man Cybern B 36(1):141–152
43. Pappa G, Freitas A (2009) Evolving rule induction algorithms with multi-objective grammar-based genetic programming. Knowl Inf Syst 19(3):283–309
44. Poli R, Graff M (2009) There is a free lunch for hyper-heuristics, genetic programming and computer scientists. In: Vanneschi L et al (eds) EuroGP'09: Proceedings of the 12th european conference on genetic programming, vol 5481. Springer, pp 195–207
45. Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk
46. Poli R, Woodward J, Burke EK (2007) A histogram-matching approach to the evolution of bin-packing strategies. In: 2007 IEEE congress on evolutionary computation. IEEE Press, Singapore, pp 3500–3507
47. Ross P, Corne D, Terashima-Marín H (1996) The phase-transition niche for evolutionary algorithms in timetabling. In: PATAT'95: The 1st international conference for the practice and theory of automated timetabling. Springer, London, pp 309–324
48. Ross P, Hart E, Corne D (1998) Some observations about ga-based exam timetabling. In: PATAT'97: The 2nd international conference for the practice and theory of automated timetabling. Springer, London, pp 115–129
49. Ross P, Hart E, Corne D (2003) Genetic algorithms and timetabling. Springer, New York 755–771
50. Schumacher C, Vose MD, Whitley LD (2001) The no free lunch and problem description length. In: Spector L et al (eds) GECCO'01: Proceedings of the 3rd genetic and evolutionary computation conference. Morgan Kaufmann, CA, USA, pp 565–570
51. Smith JE (2007) Coevolving memetic algorithms: a review and progress report. IEEE Trans Syst Man Cybern B 37(1):6–17
52. Terashima-Marin H, Ross P, Valenzuela-Rendon M (1999) Evolution of constraint satisfaction strategies in examination timetabling. In: Banzhaf W et al (eds) Proceedings of the genetic and evolutionary computation conference, vol 1. Morgan Kaufmann, Florida, pp 635–642
53. Ülker Ö, Özcan E, Korkmaz EE (2006) Linear linkage encoding in grouping problems: applications on graph coloring and timetabling. In: Burke EK, Rudová H (eds) PATAT'06: The 5th international conference for the practice and theory of automated timetabling, Lecture Notes in Computer Science, vol 3867. Springer, pp 347–363
54. Voudouris C, Dorne R, Lesaint D, Liret A (2001) iopt: a software toolkit for heuristic search methods. In: Walsh T (ed) Proceedings of the 7th international conference on principles and practice of constraint programming, vol 2239. Springer, pp 716–719
55. Welsh D, Powell M (1967) An upper bound for the chromatic number of a graph and its application to timetabling problems. Comput J 10(1):85–87
56. Whigham PA (1995) Grammatically-based genetic programming. In: Rosca JP (ed) Proceedings of the workshop on genetic programming: from theory to real-world applications. Tahoe City, CA, pp 33–41
57. Whigham PA (1997) Evolving a program defined by a formal grammar. In: The 4th international conference on neural information processing—The annual conference of the Asian Pacific neural network assembly (ICONIP'97). Dunedin, New Zealand
58. Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 1(1):67–82
59. Zeleny M (1974) A concept of compromise solutions and the method of the displaced ideal. Comput Oper Res 1(3):479–496