Noname manuscript No.
(will be inserted by the editor)

Grammar-based generation of variable-selection heuristics for constraint satisfaction problems

Alejandro Sosa-Ascencio · Gabriela Ochoa · Hugo Terashima-Marin · Santiago Enrique Conant-Pablos

Received: date / Accepted: date

Abstract We propose a grammar-based genetic programming framework that generates variable-selection heuristics for solving constraint satisfaction problems. This approach can be considered as a generation hyper-heuristic. A grammar to express heuristics is extracted from successful human-designed variable-selection heuristics. The search is performed on the derivation sequences of this grammar. The approach brings two innovations to grammar-based hyper-heuristics in this domain. First, the incorporation of decision functions in the function set, which allows generating heuristic closer to algorithmic procedures (not only mathematical expressions). Second, the implementation of overloaded functions that are capable of handling different input dimensionality, which brings greater flexibility. Moreover, the heuristic search space is explored using not only evolutionary search, but also two alternative simple strategies, namely, iterated local search, and parallel hill climbing. We tested our approach on synthetic and real-world instances. The newly generated heuristics have an overall good performance when compared against human-designed heuristics. We also study how the composition of the training set impacts the performance of the produced heuristics on unseen instances, and how heuristics evolved from synthetic instances perform on real-world data sets.

Keywords Constraint satisfaction problems · Hyper-heuristics · Genetic Programming · Variable ordering heuristics · Grammar-based framework

1 Introduction

Hyper-heuristics have been defined as search methods or learning mechanisms for selecting or generating heuristics to solve computational search problems [14,11]. They are related to the notions of meta-learning in machine learning [37], and autoconstructive evolution in genetic programming [42]. The main motivation is to develop search methodologies with a higher degree of generality than tailored

E-mail: a01061994@itesm.mx
Present address: F. Author if needed · S. Author
second address

metaheuristics and crafted heuristics. Hyper-heuristics differ from metaheuristics in that they explore the space of heuristics or heuristic components, rather than directly the space of solutions [11]. Two types of hyper-heuristics can be distinguished: heuristic selection (methodologies for choosing or selecting existing heuristics) and heuristic generation (methodologies for generating new heuristics from components of existing heuristics) [14]. This paper presents a methodology of the second type. Genetic programming is one of the most commonly used approaches to automatically generate heuristics, and the inclusion of grammars is a recent trend with promising results.

A constraint satisfaction problem (CSP) is defined by a set of variables X where each variable is associated to a set of values D (domain), and a set C of constraints between the variables. The objective is to either find a consistent assignment of values to variables in such a way that all constraints are satisfied, or to show that such consistent assignment does not exist. A wide range of problems and applications in various domains can be modeled and solved as CSPs [1, 23].

This article proposes a grammar-based approach for automatically generating new variable-selection heuristics for constraint satisfaction. A grammar is designed taking inspiration from successful human-designed variable-selection heuristics. The search space is therefore composed of valid sentences according to this grammar. The proposed approach differs from previous grammar-based hyper-heuristics developed for CSPs in two aspects. First, the function set contains conditional statements (`if-then-else`), arithmetic operators and logical expressions, which allow generating heuristics with a wider range of possibilities than previous approaches that are restricted to arithmetic operators. Second, the implementation of overloaded functions capable of handling different input dimensionality, which brings greater flexibility and a wider space of possible solutions. Three high-level strategies are considered for the generation of new heuristics: genetic programming (GP), iterated local search (ILS) and parallel hill-climbing (PHC).

Using both synthetic and real-world instances, we compare the performance of newly generated heuristics against several state-of-the-art human-designed heuristics. We also study the impact of the composition of the training set on the generality of the evolved heuristics, i.e, their performance across unseen instances. Finally, we consider how heuristics evolved by using sets of small synthetic instances generalize to both unseen larger synthetic instances, and real-world data sets.

The next section discusses related work on grammar-based hyper-heuristics and how they have been applied to CSPs. Section 3 describes CSPs and the variable-selection heuristics used to solve them. Section 4 explains the proposed hyper-heuristic approach including the components and structure of the grammar. Section 5 overviews the experimental setup and set of research questions addressed, while Section 6 shows the experimental results and their analysis. Finally, Section 7 summarizes our findings and suggest future research directions.

2 Related work

Genetic programming has been applied to generate heuristics in several domains such as production scheduling, cutting and packing, boolean satisfiability, timetabling

and scheduling [12,11]. We next overview previous work related to constraint satisfaction and grammar-based genetic programming hyper-heuristics.

2.1 Heuristic Generation for CSPs

An early approach to the automated generation of heuristics for constraint satisfaction was proposed by Minton [28]. This pioneering work presents a system for generating reusable heuristics by modifying given elements of algorithm 'schema', which are templates of generic algorithms. The general idea is to automatically synthesize problem-specific versions of constraint satisfaction algorithms.

Ortiz-Bayliss et al. [32,35] propose the generation of variable-selection heuristics by using a linear combination of descriptors of the variables and tuned weights. The descriptors extract information from the variables at each point in the exploration of the CSP instance, and the weights determine the importance of each descriptor. A genetic algorithm was used to tune the weights. The heuristics produced by their approach suffer from over-fitting, as in most cases they fail to generalize to unseen instances from different classes of instances.

Bain et al. [5,4] present the generation of new heuristics that compose local and complete search algorithms for solving CSPs. In Bain et al. [4] genetic programming is used to evolve a population of local search algorithms, and in a more recent work [5], the authors use beam search and random-generated heuristics and compare them against the genetic programming approach, with the latter producing better synergies.

Jafari and Mouhoub [30] develop a hybrid approach where two non-systematic algorithms are used to assign weights to constraints and variables. Prior to the backtracking process, hill climbing and ant colony optimization are used to adjust the weights assigned to variables. Once the weights have been assigned, they remain unchanged for the rest of the solving process.

2.2 Grammar-based Genetic Programming Hyper-heuristics

Bader-El-Den and Poli [2] introduce a grammar-based approach able to generate parsimonious and fast heuristics for satisfiability (SAT). The designed grammar expresses four human-created heuristics and allows flexibility to create brand new heuristics. The authors also propose a grammar-based framework for generating timetabling heuristics [3]. The grammar contains components of graph coloring and slot allocation heuristics. The results obtained are comparable with a range of human-created approaches in the literature.

Keller and Poli [21,22] devise a linear genetic programming hyper-heuristic for the traveling salesman problem (TSP). The idea is to evolve iterative programs that apply a number of simple local search operators. The programs are expressed as sentences of a grammar, which is made progressively more complex in successive papers including conditionals and loops.

Burke et al. [13] use grammatical evolution for evolving local search heuristics for one-dimensional bin packing. Grammatical evolution is a branch of grammar-based genetic programming that uses a linear representation. Their work shows

that the space of neighborhood move operators can be specified by a grammar, and high-quality operators can be evolved.

Sabar et al. [39] propose a grammatical evolution approach for generating local search heuristics to solve two combinatorial optimization problems: exam timetabling and vehicle routing. The proposed grammar contains three types of heuristic components: acceptance criteria, neighborhood structures or move operators, and neighborhood combinations/operators.

Ortiz-Bayliss et al. [34] proposed a simple grammar composed by arithmetic operators and five terminals (feature extractors), for generating variable-selection heuristics for CSP.

3 Constraint Satisfaction Problems

CSPs are an important topic of study in operational research because many combinatorial problems, such as scheduling, frequency assignment and micro-controller selection and pin assignment can be formulated as CSPs (see for example [8], [16], [20] and [7]). Several deterministic methods to solve CSPs exist (see for example [23]), and solutions are found by searching systematically through the possible assignments to variables, guided by heuristics.

3.1 Variable Ordering Heuristics

A solution to a CSP is constructed by selecting one variable at a time according to a given heuristic. In complete search methods for solving CSPs, heuristics are usually designed based on the *fail-first* principle [19] which is based on the idea of selecting the variable with the highest probability of failure. Previous studies have shown that a heuristic may work well for some classes of instances, but perform badly for others [33]. Nine human-designed variable ordering heuristics are used in this investigation and they are described below:

Minimum remaining values (MRV) heuristic. Selects the variable with the fewest available values in its domain [19,38]. The idea consists basically in taking the most restricted variable from those which have not been instantiated yet and by doing so, reducing the branching factor of the search.

Solution density (RHO) heuristic. Uses the approximated calculation of the solution density ρ [18]. The idea is to select a variable that takes the search into the subproblem that contains the largest fraction of solution states. That is, the subproblem with the largest solution density. RHO will prefer the variable that, once assigned a value, maximizes:

$$\rho = \prod_{c \in C} (1 - p_c) \tag{1}$$

where p_c is the fraction of forbidden tuples in constraint c.

Expected number of solutions (ENS) heuristic. Selects the variable that produces the subproblem maximizing the expected number of solutions, defined as:

$$E[S] = \prod_{x \in X} |m_x| \times \rho \tag{2}$$

where $|m_x|$ is the domain size of variable x.
The ENS heuristic maximizes the size of the subproblem so as the solution density. The selection criterion of this heuristic is a combination of the MRV and RHO heuristics [18].

Kappa (K) heuristic. Orders the variables based on the value of the kappa factor, κ [18]. κ represents a notion of how restricted a combinatorial problem is. Problems with $\kappa \ll 1$ are less restricted and likely to have many solutions, while the problems with $\kappa \gg 1$ are highly restricted and likely to be unsatisfiable [18]. K will select first the variable that leads the search into the subproblem that maximizes the value of κ:

$$\kappa = \frac{-\sum_{c \in C} \log_2(1 - p_c)}{\sum_{x \in X} \log_2(|m_x|)} \tag{3}$$

Maximum backward degree (MBD) heuristic. Selects the first variable arbitrarily. Then, at each stage it prefers the variable that is connected to the largest group among the variables already instantiated [45,15].

Maximum forward degree (MFD) heuristic. Prefers the variables connected to the maximum number of uninstantiated variables, that is, the variables involved in the largest number of constraints (edges between nodes) [45] .

Backward Brelaz (BBZ) heuristic. Inspired by the heuristic for graph coloring proposed by Brelaz [9], it selects the variable that minimizes:

$$bbz(x_i) = \begin{cases} \frac{|m_x|}{bdeg(x)} & \text{if } bdeg(x) > 0 \\ |m_x| & \text{otherwise} \end{cases} \tag{4}$$

where $bdeg(x)$ is the backward degree of variable x.

Forward Brelaz (FBZ) heuristic. Instantiates first the variable that minimizes the quotient of the domain size over the forward degree of the variable:

$$fbz(x_i) = \begin{cases} \frac{|m_x|}{fdeg(x)} & \text{if } fdeg(x) > 0 \\ |m_x| & \text{otherwise} \end{cases} \tag{5}$$

where $fdeg(x)$ is the forward degree of variable x.

Max conflicts (MXC) heuristic. Selects the variables according to the number of conflicts they are involved in. Therefore, it prefers the variables involved in the largest number of conflicts (which must not be confused with the number of constraints).

4 The Proposed Approach

We propose a grammar-based genetic programming system based on Backus-Naur form (BNF) for evolving variable-selection heuristics for solving CSPs. Grammars bring a number of benefits for genetic programming where the most important

```
<start>          → instanciate_var(<var>)
<var>            → select_var_from(<var_value>)
                    | if-then-else(<boolean>,<var>,<var>)
<var_value>      → smallest(<var_values[]>,csp) | largest(<var_values[]>,csp)
                    | if-then-else(<boolean>,<var_value>,<var_value>)
<var_values[]>   → conflicts(csp) | constraints_involved(csp) | fdeg(csp)
                    | bdeg(csp) | bbz(csp) | fbz(csp) | mxi(csp)
                    | plus(<double>,<var_values[]>)
                    | plus(<var_values[]>,<var_values[]>)
                    | plus(var_values[][])
                    | multiplication(<double>,<var_values[]>)
                    | multiplication(<var_values[]>,<var_values[]>)
                    | multiplication(variables_values[][])
                    | minus(<double>,<var_values[]>)
                    | minus(<var_values[]>,<var_values[]>)
                    | division(<double>,<var_values[]>)
                    | division(<var_values[]>,<var_values[]>)
                    | negative(<var_values[]>)
                    | log2(<var_values[]>)
                    | if-then-else(<boolean>,<var_values[]>,<var_values[]>)
<var_values[][]> → pxi(csp) | plus(<var_values[][]>,<double>)
                    | plus(<var_values[][]>,<var_values[]>)
                    | multiplication(<var_values[][]>,<double>)
                    | multiplication(<var_values[][]>,<var_values[]>)
                    | minus(<var_values[][]>,<double>)
                    | minus(<var_values[][]>,<var_values[]>)
                    | negative(<var_values[][]>)
                    | if-then-else(<boolean>,<var_values[][]>,<var_values[][]>)
<boolean>        → greater_or_equal_than(<var_value>,<var_value>)
                    | smaller_or_equal_than(<var_value>,<var_value>)
                    | if-then-else(<boolean>,<boolean>,<boolean>)
<double>         → plus(<var_values[]>) | multiplication(<var_values[]>) | one
                    | if-then-else(<boolean>,<double>,<double>)
```

Fig. 1: Description of the grammar proposed to construct variable-selection heuristics for CSPs

one is to restrict the search space to ensure the construction of valid individuals. To generate an individual, our implementation follows the procedure proposed by Whigham [46]. A random tree is generated up to a depth bound, according to an estimation of the minimum tree-depth required by a function to reach all its terminal nodes. But unlike usual grammar-based systems, where the genotype is first decoded into a derivation tree before being transformed into an expression tree [27], the genotype in our system is directly represented as a tree structure that is recursively constructed when the individual is evaluated. This brings a significant reduction in the computational time. The components and structure of the grammar are presented in Fig. 1. The grammar module works over a strongly-typed system. The possible types that a function can receive are described in Table 1. Every component in the grammar was defined as a Java method from two main classes: simple functions and special functions.

Table 1: Type values handled by the grammar

`var`	Uninstantiated variable.
`var_value`	Double value associated to a specific variable.
`var_values[]`	One-dimensional array of length equal to the number of uninstantiated variables, each array element contain a double value associated to a variable.
`var_values[][]`	Two-dimensional matrix of dimension $[n \times t]$ where n is the number of uninstantiated variables and t is the number of constraints; each element in the matrix is a double value that relates a variable with a constraint.
`boolean`	True or false boolean value.
`double`	Double value.

Table 2: Terminal functions

Terminal	Return type	Description
`constraints`	`var_values[]`	Number of constraints in which a variable is involved.
`conflicts`	`var_values[]`	Corresponds to the number of conflicts that results to select a variable for instantiation. A *conflict* is an invalid pair of values between two variables at the same time.
`fdeg`	`var_values[]`	Is the *forward degree* used by heuristic FBZ. Calculates the number of constraints with uninstantiated variables in which the variable participated.
`bdeg`	`var_values[]`	Is the *backward degree* used by heuristic MBD. Returns the number of constraints with instantiated variables in which the variable participates.
`fbz`	`var_values[]`	Is the *forward brelaz* value used by the FBZ heuristic, and is calculated as is presented in equation 5.
`bbz`	`var_values[]`	Is the *backward brelaz* value used by the BBZ heuristic, and is presented in equation 4.
`mxi`	`var_values[]`	Number of available values for uninstantiated variable x_i.
`pxi`	`var_values[][]`	Is the fraction of infeasible pairs of values for C_{x_i} over the total number of possible assignments. Where C_{x_i} is the set of constraints in which variable x_i is involved.
`one`	`double`	Constant value of 1.0.

The terminal functions, based on elements extracted from the human-designed heuristics, are described in Table 2. The non-terminal functions are divided into three main types: arithmetical, logical and decision. Table 3 describes the arithmetic operators and their corresponding overloading.

We defined two logical functions: greater or equal than ($\geq$) and smaller or equal than ($\leq$). Both functions receive two `var_values` objects as input arguments, and return a `boolean` object.

The grammar also has three decision functions, two for selecting a variable (`smallest` and `largest`) with either the smallest or largest value contained in the `var_value[]` object, and the conditional function `if-then-else`. This is a function intended to add flexibility to the design, since it receives as argument a wide range of different object types. Also, it provides one of the main components determining the structure of the new heuristics, usually produced in the form of decision trees. The function receives three arguments, the first will always be a `boolean` object, the second and third argument must share the same object type, but this could be any of the types handled by the functions (`double, boolean,`

Table 3: Description of arithmetic operators with their corresponding overloading, i and j are the index for `var_values[]` and `var_values[][]`, where $i = 1, \ldots, n$, and n is the number of uninstantiated variables, and $j = 1, \ldots, c$, where c is the number of constraints.

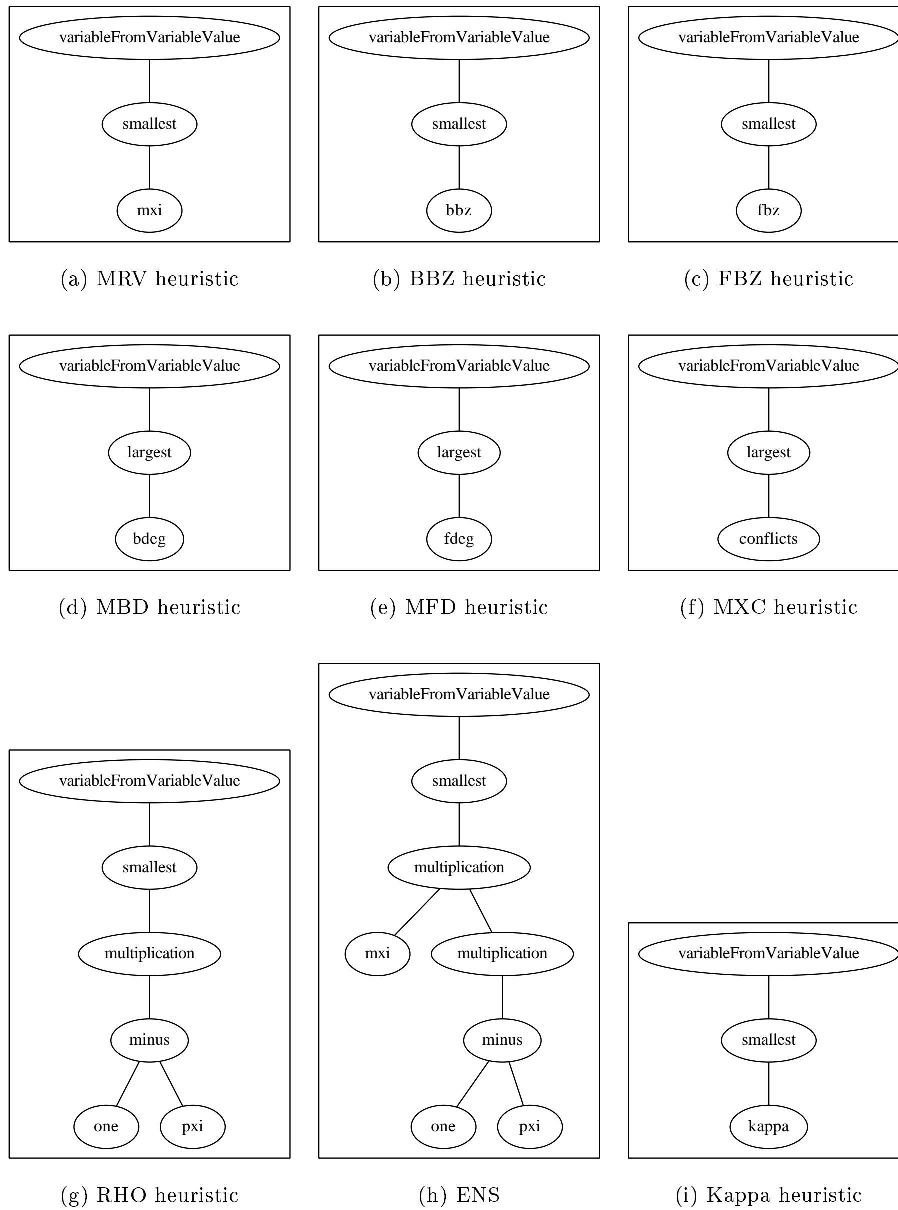
Operator	Input1 (x)	Input2 (y)	Output (z)	Description
addition	`var_values[]`		`double`	$z = \sum_{i=1}^{n} x_i$
	`var_values[]`	`var_values[]`	`var_values[]`	$z_i = x_i + y_i$
	`double`	`var_values[]`	`var_values[]`	$z_i = x + y_i$
	`var_values[][]`		`var_values[]`	$z_i = \sum_{j=1}^{c} x_{j,i}$
	`var_values[][]`	`double`	`var_values[][]`	$z_{j,i} = x_{j,i} + y$
	`var_values[][]`	`var_values[]`	`var_values[][]`	$z_{i,j} = x_{j,i} + y_j$
subtraction	`var_values[]`	`var_values[]`	`var_values[]`	$z_i = x_i - y_i$
	`double`	`var_values[]`	`var_values[]`	$z_i = x - y_i$
	`var_values[][]`	`double`	`var_values[][]`	$z_{j,i} = x_{j,i} - y$
	`var_values[][]`	`var_values[]`	`var_values[][]`	$z_{i,j} = x_{j,i} - y_j$
multiplication	`var_values[]`		`double`	$z = \prod_{i=1}^{n} x_i$
	`var_values[]`	`var_values[]`	`var_values[]`	$z = x_i \cdot y_i$
	`double`	`var_values[]`	`var_values[]`	$z_i = x \cdot y_i$
	`var_values[][]`		`var_values[]`	$z_i = \prod_{j=1}^{c} x$
	`var_values[][]`	`double`	`var_values[][]`	$z_{j,i} = x_{j,i} \cdot y$
	`var_values[][]`	`var_values[]`	`var_values[][]`	$z_{i,j} = x_{j,i} \cdot y_j$
division	`double`	`var_values[]`	`var_values[]`	$z_i = \begin{cases} y_i/x & \text{if } x \neq 0 \\ y & \text{if } x \equiv 0. \end{cases}$
	`var_values[]`	`var_values[]`	`var_values[]`	$z_i = \begin{cases} y_i/x_i & \text{if } x_i \neq 0 \\ y & \text{if } x_i \equiv 0. \end{cases}$
negative	`var_values[]`		`var_values[]`	$z_i = -1 \cdot x_i$
	`var_values[][]`		`var_values[][]`	$z_{j,i} = -1 \cdot x_{j,i}$
$\log_2$	`var_values[]`		`var_values[]`	$z_i = \log_2(x_i)$

`var_values[]`, ...). The output will be determined in terms of the value of the boolean argument. If the value is `true`, the output will be the second argument, otherwise the function returns the third argument.

During the random initialization of new heuristics, the terminal and non-terminal functions are randomly chosen by using a uniform distribution. In the future, other distributions may be applied based on domain-specific knowledge.

The grammar components were extracted from an analysis performed over the human-designed heuristics mentioned in Sec. 3. Figure 2 shows how these heuristics are represented by the grammar components. We observe that the human-designed heuristics are usually simple structures selecting either the largest or smaller feature value returned by one of the terminal functions.

Figure 3 gives an example of an automatically generated heuristic, where a more elaborated combination of features is made through arithmetical and decision functions. Algorithm 1 shows the pseudo-code of this heuristic.



(a) MRV heuristic (b) BBZ heuristic (c) FBZ heuristic

(d) MBD heuristic (e) MFD heuristic (f) MXC heuristic

(g) RHO heuristic (h) ENS (i) Kappa heuristic

Fig. 2: Human-designed heuristics constructed with the grammar proposed

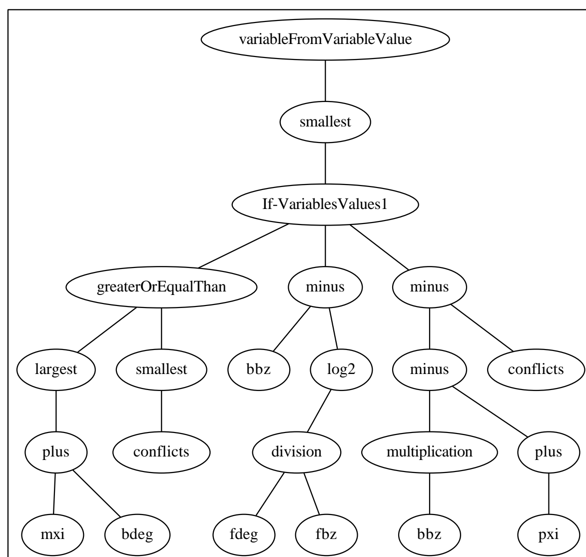


Fig. 3: Example of a heuristic generated with the grammar proposed and GP as the high-level strategy. The `if` nodes specify the object type that is received as second and third arguments

Algorithm 1 Pseudo-code for the heuristic depicted in Fig. 3

function NEW HEURISTIC(csp)
  `var_values`[ ] $r = null$
  `var` $variable = null$
  //The condition of the if statement corresponds to the left subtree
  if $largest(csp.m_{x_i} + csp.bdeg) \geq smallest(csp.conflicts)$ **then**
    $r = csp.bbz - \log_2 \left(\frac{csp.fdeg}{csp.fbz} \right)$ //Middle subtree below if-node
  else//Right subtree below if-node
    double $c = \prod csp.bbz$
    int $n = csp.uninstantiated_variables$
    int $t = csp.number_of_constraints$
    `var_values`[ ][ ] $pxi = csp.p_{x_i}$
    for $i := 1, n$ **do** //Overloading of addition operator over feature pxi
      $r_i = \sum pxi_i$
    end for
    $r = c - r$
    $r = c - csp.conflicts$
  end if
  $variable = variableFromVariableValue(smallest(r))$
  return $variable$
end function

4.1 Fitness Function

The cost to solve a CSP instance is measured by the number of consistency checks required to find the first solution or to prove than none exists –a consistency check occurs every time a constraint is to be revised. Then, in general, easy instances require less consistency checks than more difficult instances. Since the training set may contain instances of different difficulty, we gave equal weight to each instance in the training set, because otherwise, the evolutionary process could drift into generating specialized heuristics for solving hard instances that would fail to generalize. The evolutionary process is guided by the maximization of the fitness function given by Equation 6,

$$f\left(h\right)=\sum_{i=0}^{T}\frac{b_i}{c_{i,h}} \tag{6}$$

where the fitness of the heuristic h is computed as a sum of coefficients, each being the quotient between b_i (lowest number of consistency checks for instance i provided by a heuristic) and $c_{i,h}$ (number of consistency checks produced by heuristic h when solving instance i), for the training set of size T. Before the search process begins, a vector $\mathbf{b}$ is initialized with the lowest number of consistency checks obtained using the human-designed heuristics. The value of b_i changes dynamically as the search progresses and new heuristics are generated. Every time a smaller count of consistency checks is found, b_i is updated accordingly, causing the coefficient to have a maximum value of 1, since in this case $b_i = c_{i,h}$. During this process, the maximum fitness value that a heuristic can receive is T.

4.2 Search Operators

When a heuristic is created or modified, the system guarantees that the returned value for the function in the root node will be a variable/value tuple. The depth of a tree is always controlled by having in each terminal or non-terminal function what we call a *maximum expansion value* (MEV), which is the maximum-allowed depth from a specific node to the leaves. Every time a node is added to the tree, the maximum depth allowed from this node's level is verified, and only the grammar components that meet the MEV restriction are used to build the next level.

To search the space of possible heuristics, three operators are used:

- **Crossover**. Receives as input two parent heuristics, we call them $parent_1$ and $parent_2$. It randomly selects a node (except the root) from $parent_1$ and enumerates all nodes in $parent_2$ that share the same expected return type as the selected node from $parent_1$. If no node with these features is found, a new random node will be selected from $parent_1$. Once the system finds two nodes with the same return type, it verifies that after the interchange of nodes, both trees fall within the maximum depth allowed. If this is not met, another node from $parent_2$ will be considered until a valid assignment is found or the list is empty. In this case, a new random node will be selected from $parent_1$.
- **Normal mutation**. Randomly selects a node, which will be replaced by a new sub-tree, following the same rules used in the generation of new trees.

– **Small mutation**. Selects only a leaf node at a time and replaces it with a different terminal function of the same type.

4.3 High-level Strategies

To guide the construction of new heuristics, we implement three different search methods as hyper-heuristics to explore the space of possible combinations of grammar components. We used genetic programming (GP), iterated local search (ILS), and parallel random hill climbing (PHC), to generate heuristics in the way of decision trees.

4.3.1 Genetic Programming Hyper-heuristic

We used a generational approach with population size of 50, elitism of 5 individuals, 30 generations and tournament selection of size 2. Once the parents have been selected for crossover, a probability of 0.9 was used for actually crossing the parents, and normal mutation probability of 0.05. All these parameters were settled down from empirical analysis performed in a previous work [41], where those parameters showed to be adequate considering performance and expressiveness.

4.3.2 Iterated Local Search Hyper-heuristic

Iterated local search (ILS) is a relative simple but effective algorithm, which has been rediscovered several times [26,6,24]. The key idea is to generate a sequence of solutions by using two basic operators: local search and perturbation. The local search operator is in charge of improving solutions, while the perturbation operator moves to other regions in the search space to escape from local optimal solutions.

One of the motivations behind including ILS in this investigation as an alternative search to evolutionary search, was its simple implementation and promising results provided by hyper-heuristic implementations in other domains [44,10,40].

Our implementation operates by generating an initial solution (a variable-selection heuristic) that is equivalent to an individual of the GP population. The local search heuristic uses the small mutation, while the perturbation step, the normal mutation. The acceptance criterion simply accepts improvements. The perturbation operator is applied at a rate of 1/10 times with respect to the calls to the local search operator. The pseudo-code for ILS is shown in Algorithm 2.

4.3.3 Parallel Hill Climbing Hyper-heuristic

Parallel hill climbing (PHC) [36] is based on the same principle that general hill climbing algorithms: start from a random initial solution and, by using a neighborhood function, visit a certain number of neighbors until a better solution is found or the stopping criteria is reached. PHC differs from traditional hill climbing by having multiple initial search points, so that multiple hills are climbed in parallel. Our implementation of PHC uses the GP module to randomly generate an initial population of 50 heuristics. Then, each heuristic is used to solve a set of training instances and its fitness value is calculated according to Equation 6. We

Algorithm 2 ILS Algorithm

s_0 ←Generate initial solution
n ←Number of times to apply the restricted mutation
for $i \leftarrow 1, n$ **do** //Local Search
    $s^\star$ ←Restricted Mutation(s_0)
    $s^\star$ ←Acceptance Criterion (s_0)
end for
repeat
    s' ←Perturbation($s^\star$)
    for $i \leftarrow 1, n$ **do**
        $s^{\star\prime}$ ←Restricted Mutation(s')
        $s^\star$ ←Acceptance Criterion($s^{\star\prime}$)
    end for
until termination criterion is met

then generate an alternative population from the previous one by using the normal mutation operator as neighborhood function. Each individual in the original population is mutated and the offspring become part of the alternate population with their corresponding fitness function. If the fitness of the new individual is better than its predecessor, the value is updated. This process is repeated for 30 iterations to complete the 1500 objective function evaluations, which was fixed as the termination criterion. The pseudo-code for PHC is shown in Algorithm 3.

We only performed 31 runs for PHC, using training set ABCDE to guide the search. We decided to exclude experiments with the other four training sets, because our implementation of PHC takes considerably more time than GP and ILS. This is because PHC explores neighborhoods by applying traditional mutation, which follows the same principle of generating new parse trees from scratch. This process is computationally more expensive than the crossover used by GP and the mutation of terminal nodes applied by ILS. although GP and ILS use traditional mutation, the occurrence of this operator is around one tenth of which occurs in PHC.

Algorithm 3 PHC Algorithm

$best := 0$
Generate initial population S
Evaluate initial population $f(s_i)\,\forall s_i \in S$
repeat
    for each population element $s_i \in S$ **do**
        Visit neighbor $s'_i \leftarrow Mutation(s_i)$
        Evaluate neighbor $f(s'_i)$
        if $(f(s_i) < f(s'_i))$ **then**
            $s_i \leftarrow s'_i$
        end if
    end for
until termination criteria met
for each population element $s_i \in S$ **do**
    if $best < f(s_i)$ **then**
        $best \leftarrow f(s_i)$
        $s^\star \leftarrow s_i$
    end if
end for
return $s^\star$

5 Experimental Setup

This section describes the constraint satisfaction solver used, the problem instances employed, and the approach used for training and testing the hyper-heuristic system.

The CSP solver used was fully implemented in Java by Ortiz-Bayliss et al. [31, 35,43]. The solver includes the AC3 constraint propagation method [25], together with backjumping [17] as backtracking strategy. The Min Conflicts (MNC) heuristic [29] is used for value selection. This heuristic selects the value with the minimum number of conflicts with the previous assigned values.

The maximum depth of trees in the grammar-based GP system was set to 6 levels. This followed preliminary experiments, where depths of 8 and 10 levels produced similar performance with a higher resource consumption.

5.1 Problem Instances

Our study considers both synthetic instances produced with random generation models, and real-world data sets, as described below.

5.1.1 Synthetic Instances

When using CSP random generation models, it is common to select instances from a region of relative difficulty, known as the phase transition [48,47]. This occurs at certain critic connectivity value, when the feasibility of instances changes abruptly from having a solution to being unsatisfiable. Instances with parameters below this threshold are under-constrained and thus easy to solve. Instances above the threshold are over-constrained and it is easy to determine the absence of solutions. It is around the phase transition, where determining if a problem has or not a solution takes a higher computational effort.

Our experiments use synthetic instances produced with model RB [48]. In this model the domain size of each variable increases polynomially with the number of variables. The domain size is uniform over all the variables and calculated by $m = n^{\alpha}$, where n is the number of variables and α is a constant greater than 0. All constraints have at least 2 variables ($a \geq 2$). This model guarantees to present the phase transition phenomena even when the number of variables approaches to infinity. The generation of an instance in Model RB proceeds as follow:

1. Generate $t{=}rn\ln(n)$ constraints, where n is the number of variables and r is a constant determining the growth of the number of constraints. Each constraint is made by selecting without repetition a of n variables.
2. Uniformly select without repetition pm^n disallowed tuples of values for each constraint.

We defined five classes of synthetic CSP instances with 20 variables and 10 values in their domains and fixed values of constraint density p_1 and tightness p_2. The classes of synthetic instances considered for this investigation are characterized as follows:

– Class A (easy satisfiable instances): $\langle 20, 10, 0.20, 0.30\rangle$ - Low constraint density and low tightness.

Table 4: Name and composition of the training sets

Set name	Composition
ABCDE	4 instances of each class A, B, C, D and E
AB	10 instances of class A and 10 of class B
AD	10 instances of class A and 10 of class D
ED	10 instances of class E and 10 of class D
BCE	7 instances of class B, 7 of class C and 6 of class E

- Class B: $\langle 20, 10, 0.20, 0.80 \rangle$ - Low constraint density and high constraint tightness, generate hard instances within the phase transition.
- Class C: $\langle 20, 10, 0.45, 0.50 \rangle$ - Medium constraint density and medium tightness, generate hard instances within the phase transition.
- Class D (easy unsatisfiable instances): $\langle 20, 10, 0.75, 0.80 \rangle$ - High constraint density and high tightness.
- Class E: $\langle 20, 10, 0.75, 0.20 \rangle$ - High constraint density and low constraint tightness, produce hard instances within the phase transition.

Initially, a few experiments were conducted with instances of 30 variables and domain sizes of 20, but the computational run-time grew considerably while the behavior of the experiments remained similar. Instances with 20 variables and 10 values in their domains proved to be enough to show the transition in the difficulty level between different regions in the search space.

Once the classes of synthetic instances were defined, we produced specific instances from those classes to be used either for producing new heuristics or for testing them. To produce new heuristics, five training sets of 20 instances were generated as described in Table 4.

For testing purposes five sets were also produced, each containing 40 unseen instances from each class (totaling 200).

5.1.2 Real-world instances

Two sets of benchmark problems are considered [1]. First, the radio frequency assignment problem, where the objective is to assign frequencies to a number of radio links, satisfying a large number of constraints using as few frequencies as possible. This set contains 8 satisfiable and 6 unsatisfiable instances. Second, the job-shop problem domain, containing ten satisfiable instances. More details of these instances are reported in Table 8.

Additionally to the 200 synthetic instances from the testing set defined in the previous section, both the 14 radio frequency assignment instances and the 10 job-shop ones were also exclusively used for testing purposes.

6 Results

Three sets of experiments were conducted in order to assess:

[1] The real-world benchmark problems used can be found at `http://www.cril.univ-artois.fr/lecoutre/benchmarks.html`, under the names ‘RLFAP-graphs’ and ‘jobShop-e0ddr1’

1. Whether heuristics generated with our system outperform human-designed heuristics, and a comparative performance of the three proposed high-level search strategies (GP, ILS, PHC) used as hyper-heuristics.
2. The impact of the composition of the training set upon the generality (or specificity) of the produced heuristics.
3. Whether heuristics trained on synthetic instances can generalize to solve real-worlds instances.

The following subsections summarize the results obtained in each case.

6.1 Comparing evolved Heuristics against Human-designed Heuristics

In order to compare the performance of heuristics generated by the three high-level strategies (GP, ILS, PHC) among themselves and against the human-designed heuristics, 31 heuristics were generated with each algorithm, using training set ABCDE and tested on each one of the available testing sets of synthetic instances. Figure 4 illustrates the results separating the instances by testing set. The horizontal line indicates the best possible result obtained by using any of the human-designed heuristics described in Sec. 3.1 for each particular instance. The values of the fitness function (Equation 6) presented in the box-plots have a value of b_i equal to the count of consistency checks of the best human-designed heuristic for instance i.

The box-plots in Fig. 4 clearly indicate that for instances from testing sets A, B and C, the automatically generated heuristics for the three high-level strategies, outperform the best-performing choice of human-designed heuristic per instance. For testing sets D and E, the median value of the evolved heuristics is below the best human-designed heuristic. However, the box-plots show that around 25% of the heuristics have a better performance than the best choice of human-designed heuristic per instance. One interesting result is that for instances from class A and B, usually is difficult to have a good gain against human-designed heuristics, because they are relatively easy to solve and the backtracking process does most of the work, but we can see in Fig. 4 that independently of the approach (GP, ILS, and PHC) the median value of the results of the automatically generated heuristics have an improvement of 20% on the fitness value over human-designed heuristics for class A, and for class B is around double.

These results suggest that the automatically generated heuristics in our system have an overall similar behaviour, independently of the search method (GP, ILS, and PHC), and presenting a decrease in performance when solving instances with high constraint density.

To determine which of the three algorithms (GP, ILS, PHC) produces the best-performing heuristics, we applied a Friedman test over the results obtained by each heuristic across the complete test set. The results are presented in Table 5, where each row corresponds to the Friedman ranking of each algorithm on each instance class (lesser value indicates better performance). The results indicate that for classes A, B and E, the heuristics generated with PHC outperform those generated by GP and ILS. For classes C and D, GP has a better performance. The p-value indicates statistical significance supporting that at least two of the algorithms have a different median performance. The ranking of the best approach

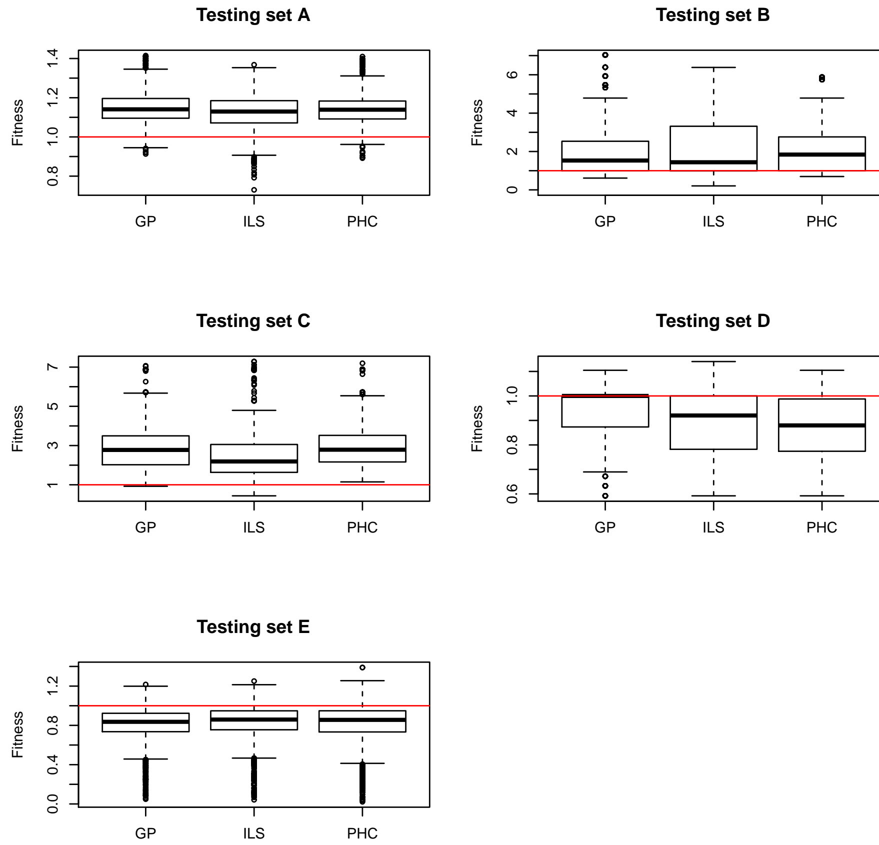


Fig. 4: Performance of heuristics generated with GP, ILS, and PHC (horizontal axis) using the training set ABCDE. Results are presented for each testing set.

over a specific class is highlighted in bold font. Although, none of the high-level strategies dominates in all classes, it is surprising to note that the simpler parallel hill climber (PCH), is competitive and indeed produced the best performance in 3 out of 5 testing instance classes. This indicates that the structure of the search space induced by the proposed grammar is more important than the high-level strategy employed to explore the space. The random generation of a sufficiently large number of heuristics is likely to produce good results. To support this insight, Fig. 5 illustrates the average best and mean fitness over time for the 31 heuristics evolved by GP. Fitness is measured as the heuristic performance over the 20 instances in the training set being used. It is worth mentioning that the GP population is of size 50. Therefore, the initial generation consists of 50 randomly generated heuristics. Since 31 runs are considered, a total of $50 \times 31 = 1,550$ heuristics were randomly generated as part of the population initialization process. As we can observe from Fig. 5, the first GP generation already produced

Table 5: Friedman ranking of the three hyper-heuristic methods on the five testing sets

Testing set	GP	ILS	PHC	p-value
A	1.9870	2.2040	**1.8088**	7.0908E-11
B	2.0217	2.1544	**1.8237**	5.0007E-11
C	**1.7661**	2.4286	1.8052	1.8432E-10
D	**1.8302**	2.1205	2.0491	3.4050E-11
E	2.0939	1.9572	**1.9487**	2.6620E-4

individuals with high fitness values, which slightly improve across the run. The mean fitness gradually increases, with most of the improvement occurring during the first 10 generations.

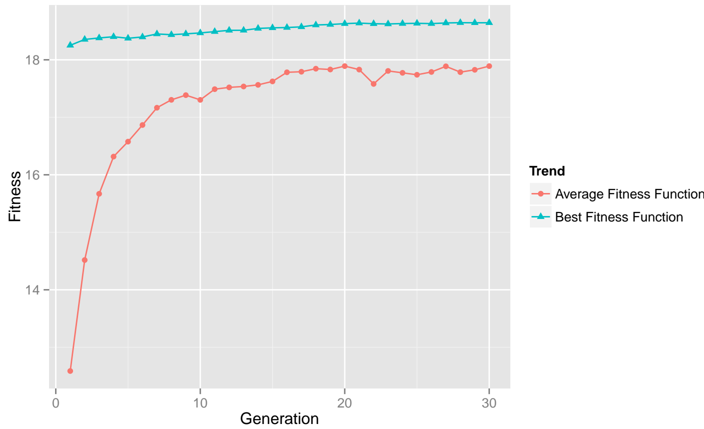


Fig. 5: Average of the best and mean fitness values from the 31 runs of the GP hyper-heuristic.

6.2 Impact of the Training Set on the Heuristic Generality

This subsection explores the impact of the composition of the training set upon the generality or specificity of the evolved heuristics. A single high-level strategy namely, GP is considered. To analyse the results we use both the Friedman and Aligned Friedman tests, as the latter allows comparability among data sets and it is desirable when the number of methods to compare is small. Table 6 shows the average rankings of the 155 heuristics generated (31 heuristics for each of the 5 training sets). The evolved heuristics are tested using 200 testing instances coming from all instance classes (20 from each of the 5 classes). The values in the table are arranged in descending order, from the best ranked approach to the worst, according to the Aligned Friedman value. Results suggest that the best

performing (i.e more general) heuristics tend to be those with more variety of classes in their training set.

Table 6: Average Friedman and Aligned Friedman rankings of the training sets used with GP over the five testing sets (A, B, C, D, and E)

Training Set	Friedman	Aligned Friedman
ABCDE	5.0795	27056.2877
BCE	5.2703	27359.1380
ED	4.8844	29804.9437
AD	5.1109	30341.5815
AB	5.5263	31482.8831

Figure 6 illustrates the performance of heuristics evolved with different training sets (one box per each set) as indicated in the horizontal axis. Each sub-figure illustrates the performance on a separate test set consisting of instances of a single class (indicated in the plot title). The horizontal line indicates the performance of the best human-designed heuristic for each test instance class.

For classes A, B, and E the distribution of performance and median values suggest that all the evolved heuristics have similar behaviour. That is, the composition of the training set does not impact on the performance of the evolved heuristics. On the other hand, for testing sets C and D, the composition of the training set is a determining factor. In particular, the presence of instances in class C in the training set (i.e training sets ABCDE and BCE) , improves the performance of heuristics when tested with class C instances. Similarly, the training sets containing instances from class D (i.e. ABCDE, AD and ED), produce improved performance when tested with class D instances. It is worth noticing that in all cases, there are evolved heuristics that outperform the best human-designed heuristic.

6.3 Performance of evolved Heuristics on Real-world Sets

Tables 7 and 8 report the ratio of the number of consistency checks by the best performing human-designed heuristic over the number of consistency checks required by the best performing evolved heuristics, when solving two sets of real-world instances. Values higher than one indicate a proportional improvement of the generated heuristics over the number of consistency checks of the best human-designed heuristic. The best heuristic produced by each high-level strategy is considered. Values in bold font highlight the fitness of the best automatically generated heuristics by each approach (GP, ILS, and PHC) that outperform the best human-designed heuristic.

Table 7 shows the results for the radio frequency assignment problem, where the best heuristics are generated by ILS and GP, with ILS slightly outperforming GP. In most cases, the best human-designed heuristic is outperformed by an evolved heuristic, with some exceptions like instances ‘graph3’, ‘graph4’ and ‘graph12’. It is interesting to observe that the improvement, in some cases, reaches over 500 times. We could remark at this point, that the comparison presented is against the result

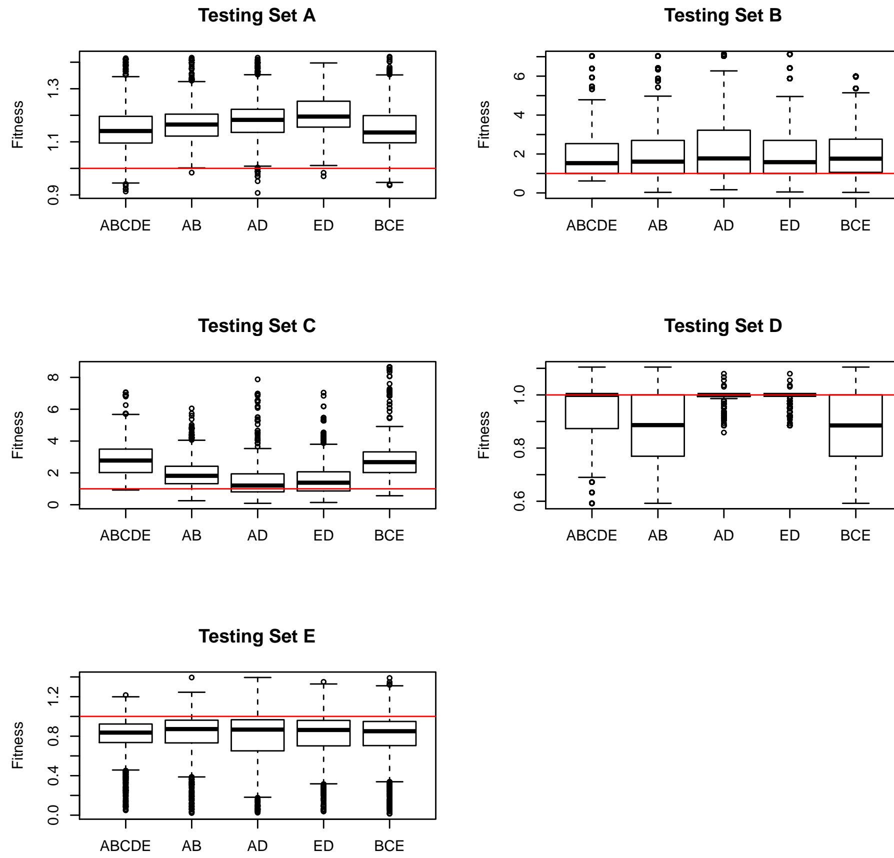


Fig. 6: Performance of heuristics generated with GP using the five different training sets (ABCDE, AB, AD, ED, BCE in the horizontal axis). Results are presented over the complete testing set, grouping testing instances by class (sub-figures with titles).

of the best human-designed heuristic for each instance, and the instances in the real set are different in size (variables, constraints) to those employed for training the high-level strategies. This is a good indication that the scheme is capable of producing good heuristics able to generalize over a wider range of problems.

Table 8 shows the performance of the three best heuristics generated with ILS, PHC, and GP, respectively, when solving 10 job shop instances, modeled as CSPs. All instances in this set contain 50 variables and 265 constraints. The evolved heuristics show an improvement for 5 out of 10 instances, reaching a magnitude up to 298 times for the instance 'jobShop2'. However, the evolved heuristics did not perform as well for the rest of the instances. This opens up an opportunity for further research in order to determine the causes of this behavior that may guide to redesign the process for generating heuristics and obtain a better performance.

Table 7: Performance of the three best heuristics found by each hyper-heuristic approach (GP, ILS, and PHC) on the radio frequency assignment problem

Instance	Variables	Constraints	Size of domains	GP	ILS	PHC
graph1	200	1134	24,44,6,22,42,36	**1.0099**	0.9798	**1.0004**
graph2	400	2245	44,42,36,6,22	0.9593	**1.0068**	**1.0791**
graph3	200	1134	42,24,44,22,36,6	0.7862	0.9300	0.9999
graph4	400	2244	24,36,42,44,22	0.0757	0.0759	0.0893
graph5	200	1134	42,44,22,6,36,24	**189.9457**	**127.7952**	0.9216
graph6	400	2170	42,44,24,36,22,6	**525.4997**	**525.4997**	0.9512
graph7	400	2170	44,36,42,22,24,6	**1.9618**	**299.3141**	**1.1157**
graph8	680	3757	22,42,44,6,36,24	**1.1799**	**1.2035**	0.9988
graph9	916	5246	44,36,42,22,6,24	**1.3311**	**1.2702**	0.7211
graph10	680	3907	24,42,22,36,44	**1.0143**	**1.0202**	0.0038
graph11	680	3757	36,22,42,44,6,24	**521.7835**	**304.3901**	**161.0860**
graph12	680	4017	24,36,22,44,6,42	0.1218	0.1378	0.0301
graph13	916	5273	24,22,44,36,42,6	**129.4874**	**129.4874**	0.6915
graph14	916	4638	44,24,22,36,42	**1.0210**	**1.0350**	0.9568

Table 8: Performance of the three best heuristics found by each hyper-heuristic approach (GP, ILS, and PHC) on jobshop benchmark problems

Instance	Variables	Constraints	Size of domains	GP	ILS	PHC
jobShop1	50	265	107,106,103,100,115, 105,113,108,114	0.0037	0.0037	0.0037
jobShop2	50	265	110,103,118,102, 109,104,111	**298.4025**	**289.0380**	**290.8993**
jobShop3	50	256	102,111,114,104, 113,106,110	0.0893	0.0894	0.0893
jobShop4	50	265	93,111,106,113, 100,115,116,104	0.1155	0.0134	0.0032
jobShop5	50	265	120,113,111,116, 117,102,95	0.9657	0.8972	0.9369
jobShop6	50	265	107,109,124,113, 101,120,108,117	**1.1657**	**1.1055**	**1.1521**
jobShop7	50	265	105,107,115,103, 99,111,104,118	**242.4871**	**260.2672**	**280.3285**
jobShop8	50	265	106,102,121,110, 107,109,105,112	0.0034	0.0034	0.0033
jobShop9	50	265	115,110,122,112, 102,113,106,119,121	0.9629	**1.0218**	0.8261
jobShop10	50	265	119,111,109,112, 114,108,102	**1.0017**	**1.0003**	0.9986

7 Discussion and Conclusions

This article presents the design and empirical validation of a grammar-based hype-heuristic framework for generating variable selection heuristics in constraint satisfaction. The grammar incorporates components of successful human-designed heuristics. Three high-level search strategies were used to explore the search space.

Our results suggest that the constrained search space imposed by the proposed grammar is the main player in the generation of good heuristics. Good solutions can be found when a large enough sample of heuristics is randomly produced. Indeed, it was easy to find at least one heuristic that outperformed most human-designed heuristics on a small set of instances. However, in order to generate a general heuristic, capable of outperforming human-designed heuristics over a bigger set of instances, it was necessary to further refine the randomly generated heuristics. Even small improvements on fitness over a small training set, produced significant improvements on the generality of the evolved heuristics across larger unseen testing sets.

To generate competitive and more general heuristics, the composition of the training set and the search methodology used played an important role. Our results suggest that increasing the variability of the training set improves the generality of the evolved heuristics. Among the three high-level strategies used, GP produces a more consistent performance over unseen random-generated instances, although the differences are not large.

The evolved heuristics had an overall improved performance when compared against the human-designed heuristics. However, they do not completely dominate. We can make the following two observations. First, there is a huge improvement over human-designed heuristics on unseen random-generated instances with low and medium constraint tightness, regardless of the constraint density. Second, when the constraint tightness is high, the average performance of the evolved heuristics is low, and only in few cases they are competitive against human-designed heuristics.

It was interesting to observe that the best heuristics evolved from synthetic training sets, have an improved performance as compared to the human-designed heuristics, on one set of real-world instances (radio frequency assignment) and a competitive performance on the second real-world data set (job-shop problems).

Future work will both explore the performance of an alternative grammar to tackle the same problem, and adapt the framework to tackle an additional problem in the domain of cutting and packing.

Acknowledgements This research was supported in part by Tecnológico de Monterrey under the strategic project PRY075 and the CONACyT Project under Grant 99695.

References

1. Achlioptas, D., Kirousis, L.M.: Random constraint satisfaction: A more accurate picture. Practice of Constraint pp. 329–344 (1997)
2. Bader-El-Den, M., Poli, R.: A gp-based hyper-heuristic framework for evolving 3-sat heuristics. Proceedings of the 9th annual conference on ... p. 1749 (2007)
3. Bader-El-Den, M., Poli, R., Fatima, S.: Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. Memetic Computing **1**(3), 205–219 (2009)
4. Bain, S., Thornton, J., Sattar, A.: Evolving algorithms for constraint satisfaction. Congress on Evolutionary Computation pp. 265–272 (2004)
5. Bain, S., Thornton, J., Sattar, A.: Methods of automatic algorithm generation. PRICAI 2004: Trends in Artificial Intelligence pp. 1–10 (2004)
6. Baum, E.B.: Iterated descent: A better algorithm for local search in combinatorial optimization problems. Tech. rep., Caltech, Pasadena, CA (1986)

7. Berlier, J., McCollum, J.: A constraint satisfaction algorithm for microcontroller selection and pin assignment. In: Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon), pp. 348–351 (2010)
8. Brailsford, S.C., Potts, C.N., Smith, B.M.: Constraint satisfaction problems: Algorithms and applications. European Journal of Operational Research **119**(3), 557–581 (1999)
9. Brelaz, D.: New methods to colour the vertices of a graph. Communications of the ACM **22** (1979)
10. Burke, E.K., Curtois, T., Hyde, M., Kendall, G., Ochoa, G.a., Petrovic, S., Vázquez-Rodríguez, J.A., Gendreau, M.: Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms. In: Congress on Evolutionary Computation. IEEE, Barcelona (2010)
11. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. Journal of the Operational Research Society (JORS) **64**(12), 1695–1724 (2013). DOI 10.1057/jors.2013.71
12. Burke, E.K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.: Exploring hyper-heuristic methodologies with genetic programming. In: C. Mumford, L. Jain (eds.) Computational Intelligence: Collaboration, Fusion and Emergence, Intelligent Systems Reference Library, pp. 177–201. Springer (2009)
13. Burke, E.K., Hyde, M.R., Kendall, G., Member, S.: Grammatical evolution of local search heuristics. Transactions on Evolutionary Computation **16**(3), 406–417 (2012)
14. Burke, E.K., Hyde, M.R., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.: A Classification of Hyper-heuristic Approaches, *International Series in Operations Research & Management Science*, vol. 146, pp. 449–468. Springer US (2010). DOI 10.1007/978-1-4419-1665-5_15
15. Dechter, R., Meiri, I.: Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. Artificial Intelligence **38**(2), 211–242 (1994)
16. Dunkin, N., Allen, S.: Frequency assignment problems: Representations and solutions. Tech. Rep. CSD-TR-97-14, University of London (1997)
17. Gaschnig, J.: Experimental case studies of backtrack versus waltz-type versus new algorithms for satisfying assignment problems. In: C.I.P. Society (ed.) Second Biennial Conference of the Canadian Society for Computational Studies of Intelligence. Toronto (1978)
18. Gent, I., MacIntyre, E., Prosser, P., Smith, B., T.Walsh: An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP'96), pp. 179–193 (1996)
19. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14**(3), 263–313 (1980)
20. Hell, P., Nesetril, J.: Colouring, constraint satisfaction, and complexity. Computer Science Review **2**(3), 143–163 (2008)
21. Keller, R., Poli, R.: Linear genetic programming of parsimonious metaheuristics. In: Evolutionary Computation, 2007. CEC 2007. IEEE Congress on, pp. 4508–4515 (2007)
22. Keller, R.E., Poli, R.: Self-adaptive hyperheuristic and greedy search. 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence) pp. 3801–3808 (2008)
23. Kumar, V.: Algorithms for constraint satisfaction problems: A survey. AI MAGAZINE **13**(1), 32–44 (1992)
24. Lourenco, H.R., Martin, O., Stutzle, T.: Iterated local search. Tech. rep., Kluwer Academic Publishers, Norwell, MA (2002)
25. Mackworth, A.: Consistency in networks of relations. Artificial intelligence **8**(1), 99–118 (1977)
26. Martin, O., Otto, S.W., Felten, E.W.: Large-step markov chains for the traveling salesman problem. Complex Systems **5**(3), 299–326 (1991)
27. McKay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O'Neill, M.: Grammar-based genetic programming: a survey. Genetic Programming and Evolvable Machines **11**(3-4), 365–396 (2010)
28. Minton, S.: An analytic learning system for specializing heuristics. IJCAI pp. 922–928 (1993)
29. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. Artificial Intelligence **58**, 161—-205 (1992)

30. Mouhoub, M., Jafari, B.: Heuristic techniques for variable and value ordering in csps. Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11 p. 457 (2011)
31. Ortiz Bayliss, J.C.: Exploring hyper-heuristic approaches for solving constraint satisfaction problems. Ph.D. thesis, Tecnológico de Monterrey (2011)
32. Ortiz-Bayliss, J.C., Moreno-Scott, J.H., Terashima-Marín, H.: Automatic generation of heuristics for constraint satisfaction problems. In: International Workshop on Nature Inspired Cooperative Strategies for Optimization (NICSO 2014), pp. 1–14 (2014)
33. Ortiz-Bayliss, J.C., Özcan, E., Parkes, A.J., Terashima-Marin, H.: Mapping the performance of heuristics for constraint satisfaction. In: CEC'10: Proceedings of the Congress on Evolutionary Computation. IEEE (2010)
34. Ortiz-Bayliss, J.C., Parkes, A.J., Terashima-Marín, H.: A genetic programming hyper-heuristic: Turning features into heuristics for constraint satisfaction. In: Workshop on Computational Intelligence (2013)
35. Ortiz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: Learning vector quantization for variable ordering in constraint satisfaction problems. Pattern Recognition Letters **34**(4), 423–432 (2013)
36. Ovalle-Martínez, F.J., Solano-González, J., Stojmenovic, I.: A parallel hill climbing algorithm for pushing dependent data in clients-providers-servers systems. Mobile Networks and Applications **9**, 257–264 (2004)
37. Pappa, G., Ochoa, G., Hyde, M., Freitas, A., Woodward, J., Swan, J.: Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. Genetic Programming and Evolvable Machines **15**(1), 3–35 (2014). DOI 10.1007/s10710-013-9186-9. URL `http://dx.doi.org/10.1007/s10710-013-9186-9`
38. Purdom, P.W.: Search rearrangement backtracking and polynomial average time. Artificial Intelligence **21**, 117–133 (1983)
39. Sabar, N., Ayob, M., Kendall, G., Qu, R.: Grammatical evolution hyper-heuristic for combinatorial optimization problems. Evolutionary Computation, IEEE Transactions on **17**(6), 840–861 (2013). DOI 10.1109/TEVC.2013.2281527
40. Soria-Alcaraz, J.A., Ochoa, G., Swan, J., Carpio, M., Puga, H., Burke, E.K.: Effective learning hyper-heuristics for the course timetabling problem. European Journal of Operational Research **238**(1), 77 – 86 (2014)
41. Sosa-Ascencio, A., Terashima-Marín, H., Valenzuela-Rendón, M.: Grammar-based genetic programming for evolving variable ordering heuristics. In: Congress on Evolutionary Computation, pp. 1154–1161. IEEE (2013)
42. Spector, L.: Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems, vol. 8, pp. 17–33. Springer (2010)
43. Terashima-Marín, H., Ortiz-Bayliss, J.C., Ross, P., Valenzuela-Rendón, M.: Using hyper-heuristics for the dynamic variable ordering in binary constraint satisfaction problems. In: A. Gelbukh, E. Morales (eds.) 7th Mexican International Conference on Artificial Intelligence, *Lecture Notes in Computer Science*, vol. 5317, pp. 407–417. Springer Berlin Heidelberg (2008)
44. Walker, J.D., Ochoa, G., Gendreau, M., Burke, E.K.: Vehicle routing and adaptive iterated local search within the hyflex hyper-heuristic framework. In: Y. Hamadi, M. Schoenauer (eds.) Learning and Intelligent Optimization, Lecture Notes in Computer Science, pp. 265–276. Springer Berlin Heidelberg (2012)
45. Wallace, R.: Analysis of heuristic synergies. In: B. Hnich, M. Carlsson, F. Fages, F. Rossi (eds.) Recent Advances in Constraints, *Lecture Notes in Computer Science*, vol. 3978, pp. 73–87. Springer (2006)
46. Whigham, P.: Grammatically-based genetic programming (1995)
47. Williams, C.P., Gogg, T.: Using deep structure to locate hard problems. In: Proceedings if AAAI, pp. 472–477 (1992)
48. Xu, K., Li, W.: Exact phase transitions in random constraint satisfaction problems. Journal of Artificial Intelligence Research **12**, 93–103 (2000)