A Genetic Programming Hyper-heuristic: Turning Features into Heuristics for Constraint Satisfaction

José Carlos Ortiz-Bayliss*, Ender Özcan*, Andrew J. Parkes* and Hugo Terashima-Marín†
*Automated Scheduling, Optimisation and Planning (ASAP)
School of Computer Science, University of Nottingham, UK
Email: {Jose.Ortiz_Bayliss, Ender.Ozcan, Andrew.Parkes}@nottingham.ac.uk
†Tecnológico de Monterrey
Campus Monterrey, Mexico
Email: terashima@itesm.mx

***Abstract*—A constraint satisfaction problem (CSP) is a combinatorial optimisation problem with many real world applications. One of the key aspects to consider when solving a CSP is the order in which the variables are selected to be instantiated. In this study, we describe a genetic programming hyper-heuristic approach to automatically produce heuristics for CSPs. Human-designed 'standard' heuristics are used as components enabling the construction of new variable ordering heuristics which is achieved through the proposed approach. We present empirical evidence that the heuristics produced by our approach are competitive considering the cost of the search when compared to the standard heuristics which are used to obtain the components for the new heuristics. The proposed approach is able to produce specialized heuristics for specific classes of instances that outperform the best standard heuristics for the same instances.**

***Index Terms*—Constraint Satisfaction, Heuristics, Hyper-heuristics, Genetic Programming**

I. Introduction

In practice, many combinatorial optimisation problems require heuristic solutions. Considering the large size of the space of solutions, it is often infeasible to perform an exhaustive search. Thus, heuristics are used to guide the search to the promising areas where a solution to the problem is expected to be found. In this study, we analyse the constraint satisfaction problem (CSP) as one of the those combinatorial problems because of its many practical applications (see for example [1], [2], [3]).

A CSP is defined by a set of n variables with each variable having a domain of m possible values, and also a set of constraints with each constraint restricting the values the variables can take simultaneously. To solve a CSP requires either to find one possible assignment of values for all the variables and that satisfies all the constraints or else to prove that no such assignment exists. CSPs can be approached by local search algorithms, that do not guarantee to find a solution [4]; or by complete methods, that guarantee to find a solution if it exists [5]. In this research, we will focus on complete algorithms for CSPs, which explore the search tree of the possible values for each of the variables of the problem [5]. Complete algorithms start from an empty variable assignment that is extended until obtaining a complete assignment that satisfies all the constraints in the problem [6]. It is a common practice to use depth first search (DFS) to solve CSPs, where every variable represents a node in the search tree and the deeper we go in that tree, the larger the number of variables that have already been assigned a feasible value. When a value assigned to one variable breaks one or more constraints, the instantiation must be undone, and another value must be considered for that variable. If there are no more values available, the value of the previous instantiated variable must be changed; this technique is known as backtracking [7]. The order in which the variables are considered for instantiation determines the form of the search tree. Different orderings to instantiate the variables produce different search trees, and different search trees represent different costs. Regarding the cost, this research treats the CSP as a combinatorial problem, not as an optimization one. There is no cost associated to the solution itself, but the cost of the search may be different for two distinct solutions. Thus, when we refer to the cost, we mean the cost of reaching a solution or proving that none exists. In this investigation, we do not intend to measure how good or bad a solution is. We will use the number of consistency checks as an estimation of the cost of the search. Every time a constraint is revised to see whether it is satisfied or not, a consistency check occurs. The smaller the number of consistency checks, the better the performance of the algorithm.

This investigation provides evidence that it is possible to delegate, at least in some tasks, the generation of heuristics to an automatic process. We have selected features that are 'old heuristics' derived from the state of the variables and used a genetic programming hyper-heuristic to combine these features into 'new heuristics' for variable ordering within CSPs. The potential advantage of this re-use of existing heuristics is that they are likely to encapsulate significant human-derived expertise and yet such recombinations may still improve them. The approach is designed for problems where search is needed to find a solution (a similar approach for local search can be found in [8]). The approach is tested in the domain of constraint satisfaction, but it may be applied to other domains that also use heuristics to guide the search.

Hyper-heuristics are motivated with the goal of automating the design of heuristic methods to solve hard computational search problems [9]. Although 'hyper-heuristic' is a relatively

new term [10], the idea of automating the design/selection of heuristics can be traced back to the early 1960's, when Fisher and Thompson [11] and Crowston [12] suggested that combining priority dispatching rules would produce a superior performance than using any of the rules in isolation for the production scheduling problem. According to Burke et al. [13], hyper-heuristics can be divided into two main categories: methodologies that select from a fixed set of heuristics and methodologies that generate new heuristics. Regarding hyper-heuristics that select among existing heuristics, they produce a mapping between the states of the problem and a feasible heuristic. These methodologies maintain a set of heuristics and then, as the problem changes, decide which heuristic to apply. Examples of these methodologies (although not all of them use the term 'hyper-heuristic' to refer to their approaches) include dynamic algorithm portfolios like CP-Hydra [14], [15], ACE (Adaptive Constraint Engine) [16] and more recent studies on heuristic selection [17], [18]. On the other hand, hyper-heuristics that produce heuristics identify critical parts of existing heuristics to create new ones [19], [20]. This study focusses on a genetic programming approach as a generation hyper-heuristic which produces new constructive heuristics based on components of the existing heuristics [21].

This paper is organized as follows. An overview of previous works related to this research is presented in Sec. II. Section III provides a description of the features used to characterize the variables and how those features are used by some well known variable ordering heuristics. In Sec. IV we describe the proposed model and how it is used to produce variable ordering heuristics. The experiments and main results are discussed in Sec. V. Finally, Sec. VI presents the conclusion and future directions of this research.

II. Background

Genetic programming [22] is an evolutionary algorithm-based methodology that borrows ideas from the theory of natural evolution to produce a program. These programs are represented by tree-based data structures. In the standard formulation, the methodology uses a specialized genetic algorithm that combines and modifies the programs through three genetic operators: selection, crossover and mutation. A fitness function is used to evaluate all the programs and then, the programs with a higher fitness are more likely to survive to future generations. Genetic programming has been widely used in investigations related to automated heuristic generation.

A. Hyper-heuristics for Heuristic Generation

Hyper-heuristics for heuristic generation have successfully been applied to various problem domains, for example: production scheduling [23], [24], cutting and packing [25], [26], satisfiability [27], [28], travelling salesman problem [29], vehicle routing problem [30] and timetabling and scheduling [31], [32]. In the domain of production scheduling, genetic programming has been used to evolve dispatching rules. Genetic programming has also been successfully applied to produce heuristics for one-dimensional bin packing [33], two-dimensional strip packing [34] and three-dimensional knapsack packing [35]. In the domain of satisfiability, Fukunaga describes CLASS (Composite Heuristic Learning Algorithm for SAT Search), an automated genetic programming heuristic discovery system for SAT. Fukunaga states that, because of the large number of ways in which the heuristic components can be combined, the task of combining them to produce effective heuristics is difficult to humans but suitable for an automated system.

Genetic programming is not the only methodology to produce hyper-heuristics for heuristic generation. Özcan and Parkes introduced a policy matrix hyper-heuristic for one-dimensional online bin packing [25]. In their approach, an offline genetic algorithm is used to evolve matrices that represent constructive heuristics.

Most of the work on hyper-heuristics for heuristic generation has explored the generation of constructive heuristics (heuristics that start with an empty assignment and try to iteratively construct a solution to the problem, one step at the time). Recent work by Burke et al. [36] investigated the generation local search heuristics by exploring the space of neighbourhood move operators that can be specified by a grammar and producing high quality operators through a grammatical evolution technique. Hong et al. [37] generated mutation operators based on different probability distributions for evolutionary programming to solve families of functions.

B. Automated Heuristic Generation for CSP

Related to CSPs, the first ideas on the automated generation of heuristics were presented in late 90's [38]. According to the authors, MULTI-TAC is described as "an expert system for operationalizing the generic heuristics". MULTI-TAC produced programs that represented heuristics designed for systematic algorithms. More recently, Bain et al. [8], [39] proposed the use of a genetic programming approach to generate new heuristics for CSPs. The authors proposed a representation that allows the generation of heuristics by combining individual functions and terminals that required some existing heuristics to be broken down into their component parts. Their approach can be used to both local search and complete methods but the experimental set was only conducted on local search algorithms. The main difference between their work and the one presented in this document is the set of functions and terminals used to construct the tree-based data structures. We consider our representation to be a much simpler one, which makes it easier to understand and interpret what the produced heuristics are doing. Also, there are significant differences in the way the heuristics are interpreted. For example, the approach described by Bain et al [8], [39] considers that each tree-based structure is by itself, a heuristic. Our approach provides a second level of generality, that considers the structures produced by the genetic algorithm, functions to be evaluated by a generic interpreter. Each of these functions can indeed be seen as a heuristic (because they define the strategy for variable ordering) but the generic interpreter that evaluates the functions and decides how to use

that evaluation allows more complex behaviours that may be useful for future studies.

III. Features and Heuristics

There are many heuristics for variable ordering within CSPs. In a general way, we can see these heuristics as procedures that receive a set of uninstantiated variables X and a heuristic function $f(x)$, and return the variable with the smallest (or largest) $f(x)$. As we will explain in the following lines, maximization or minimization can be represented by changing the sign of the value returned by $f(x)$. Thus, we can assume that the generic interpreter always deals with a minimization problem (see Algorithm 1). For example, the DOM heuristic [40] prefers the variable with the minimum domain size. Its function is $f(x) = dom(x)$. Thus, DOM will select the variable that minimizes $f(x)$. If we decide to select the variable that maximizes $f(x)$ we obtain a different heuristic (the inverse of DOM); but based on the same feature, the domain size. By using the generic interpreter described in Algorithm 1, the inverse of DOM would be represented by the function $f(x) = -dom(x)$. Similar functions can be defined for any variable ordering heuristic by considering additional features.

Algorithm 1 Generic interpreter

Require: $X = \{x_0, x_1, \ldots, x_n\}$, $f(x)$
$[value, index] \leftarrow min(f(x_0), f(x_1), \ldots, f(x_n)))$
return x_{index}

The features considered to characterize the variables within the CSP instances are described in the following lines.

- Degree, $deg(x)$. The degree of a variable is defined as the number of constraints with uninstantiated variables in which the variable participates. If we select the variable with the largest degree we obtain the DEG heuristic [41].
- Number of conflicts, $conflicts(x)$. A conflict is a pair of values $\langle a, b\rangle$ that is not allowed for two variables at the same time. The higher the number of conflicts among the constraints the variable is involved, the more unlikely that a variable can be instantiated without breaking any constraints. Thus, the heuristic that instantiates first the variable with the largest number of conflicts will be referred to as the MXC heuristic.
- Domain size, $dom(x)$. As mentioned before, selecting the variable that minimizes $dom(x)$ gives place to the DOM heuristic [40].
- Kappa, $\kappa(x)$. Inspired in the κ factor that estimates how restricted a problem is [42] we propose a similar measure to be used for each variable. $\kappa(x)$ is calculated as:

$$\frac{-\sum_{c_j \in C_x} \log_2(1 - p_{c_j})}{\log_2(dom(x))} \quad (1)$$

  where c_j is a constraint where x is involved and prohibits a fraction p_{c_j} of tuples in the constraint. If we prefer the variable that maximizes the value of $\kappa(x)$ we obtain the KAPPA heuristic [42].

Thus, four standard variable ordering heuristics are used in this investigation: DEG [41], MXC, DOM [40] and KAPPA [42]. In addition to these variable ordering heuristics, the values are ordered according to the min-conflicts heuristic (MNC) [43]. Once a variable is selected for instantiation, the first value to be tried is the one most likely to success, the one that participates in the fewest conflicts (forbidden pairs of values between two variables). In all cases, lexical ordering is used to break ties.

IV. Generation of New Heuristics for CSPs

This study proposes a method for automatically producing heuristics for variable ordering within CSPs. The method is composed by two stages. In the first stage, the main components of existing heuristics are identified. In the next stage, a genetic algorithm evolves combinations of the components obtained in the first stage (in functions represented by tree-based data structures) to obtain new heuristics.

The main component of any variable ordering heuristic is the way it decides which variable should be tried before the others. This component consists of an evaluation of the variables according to some metrics to rank the variables. For the evaluation of the variables we require the use of features that are derived from the context of each variable with respect to the other variables (see Sec. III).

A genetic algorithm is used to produce functions that can be used as inputs of the generic interpreter shown in Algorithm 1. These functions combine the features that describe the variables and allow the heuristics to order them. It is important to mention that the functions produced by the genetic algorithm cannot directly be mapped to the cost of the search for a given instance. The functions are used by the generic interpreter to order the variables and that order is what will determine the cost of the search. Then, the functions produced by the genetic algorithm indirectly determine the cost of the search. Because the functions produced by the genetic algorithm represent the core of the new heuristics, we will refer to them only as heuristics, even though we know that they are always evaluated by a generic interpreter like the one described in Algorithm 1.

A. Functions and Terminals

The genetic programming approach requires that we define a set of functions and a set of terminals. The functions provide the operations allowed in the heuristic function while the terminals represent the parameters for such functions.

The set of functions includes only four operations: addition (+), subtraction (-), multiplication (*) and protected division (%) (if the divisor is zero no division is performed and the function returns 1). All these functions receive two arguments. At the moment, these four simple operations have been sufficient to produce competitive heuristics. We consider adding more functions as part of the future work of this investigation.

One of the most interesting and challenging parts in this research, and where preliminary investigation was needed, was the definition of the set of terminals. The approach requires

$\langle S \rangle \rightarrow$ E
| E $\rightarrow$ Op F Op
| Op $\rightarrow$ (E) | T
| F $\rightarrow$ + | - | * | %
| T $\rightarrow$ $deg(x)$ | $conflicts(x)$ | $dom(x)$ | $\kappa(x)$ | r

Fig. 1. Grammar proposed for this investigation

a set of terminals that is capable of describing the current problem state for the heuristic to provide good advice. Because we are proposing an approach for generating variable ordering heuristics, it seemed reasonable for us to focus on features that describe such variables. The set of terminals contains all the features described in Sec. III. All these features are normalized in the range $[0, 1]$ to avoid that some features contribute more than others when evaluating the functions. The reader may have noticed that even though we refer to these features as terminals, they are also functions that receive a variable and return an evaluation of the variable with respect to a specific feature. Along with the features that describe the variables, we have also included a terminal r that randomly produces a real number in the range $[-1, 1]$ the first time it is invoked by a heuristic function. Because r may appear in different functions, its values are randomly chosen for each function the first time it is called and kept the same for the rest of the run.

We have proposed a grammar to define all the valid structures that can be formed by using the functions and terminals provided. This grammar is shown in Fig. 1. All the structures created during the initialization of the genetic algorithm are randomly generated by following these rules. As it is defined now, the grammar does not allow simple structures formed by only one terminal. This was done to prevent new heuristics from behaving exactly as any of the standard heuristics.

Figure 2 provides a simple example to show how the tree-based structures are defined and how they should be interpreted. The structure represents the function $((r - deg(x)) + dom(x)) * \kappa(x)$. The first time the function is evaluated, the value of r is randomly defined as explained before. Thus, it will represent a constant for the rest of the evaluations of this function. Each time the function is evaluated, the values of the features $deg(x)$, $dom(x)$ and $\kappa(x)$ change for each variable, what produces different outputs of the evaluation of the function for different variables. As we can observe, a tree-based data structure can represent a wide variety of valid functions as long as we respect the grammar defined before.

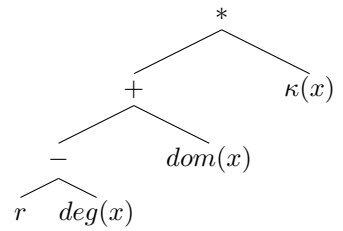

Fig. 2. An example of a tree-based data structure that represents the function $((r - deg(x)) + dom(x)) * \kappa(x)$

B. The Genetic Programming Hyper-heuristic

Our genetic programming hyper-heuristic requires a genetic algorithm to run to construct tree-based data structures that represent new heuristics. The genetic algorithm used is a generational one with memory. The memory guarantees that if an exceptional good individual is found and later lost by the evolutionary process, we still can recall it and return it as the best heuristic function, even if it is not present in the last population. The parameters for the genetic algorithm for all the experiments were set as follows. A population size of 30 was used, and a maximum depth of four was used for the generation of the initial trees. For initialization, the grow method was used as described in [22]. Each run of the genetic algorithm consisted of 50 generations. Tournament selection of size two was used to select to parents for crossover. Once the parents have been selected, there is a probability of 0.9 that they are combined. If crossover takes place, the parents are combined by using a standard one-point crossover operator for genetic programming and the new individuals are included in the new population. If crossover does not occur, the parents are incorporated to the new population without any changes. The mutation operator selects one node in the tree-based data structure at random (each node in the tree has the same probability of being chosen) and it produces a new subtree with root on the node to be mutated. Because this operator is more disruptive than the crossover operator, it is applied with a probability of 0.05, which is significantly smaller than the probability of crossover.

To estimate the fitness of the solutions, we use the following algorithm. Each of the m instances in the training set is solved with the n individuals in the population (heuristic functions). The quality of each individual i on instance j is estimated as:

$$q(i, j) = \frac{1}{cost(i, j)} \tag{2}$$

where $cost(i, j)$ is the cost of the search required by individual i on instance j. The cost of the search is estimated by the number of consistency checks required by the search.

Then, for each instance j in the training set, we normalize its quality by dividing the quality of each individual on instance j over the maximum quality among all the individuals for such instance:

$$\hat{q}(i, j) = \frac{q(i, j)}{max\left(q(1, j), q(2, j), \ldots, q(n, j)\right)} \tag{3}$$

Thus, the best individual for each instance will receive a normalized quality of 1.0. Finally, the fitness of each individual i is calculated as the sum of all the normalized qualities among the m instances within the training set:

$$f(i) = \sum_{j=1}^{m} \hat{q}(i, j) \tag{4}$$

This fitness function is the result of preliminary studies on similar experiments. One simple approach for estimating the fitness of an individual is to sum the qualities (before

normalisation), but this leads to over-fitted heuristics, which are rarely useful for unseen instances. Another drawback of such approach is that if one instance results in a very expensive search with respect to other instances in the training set, one heuristic that performs well on that instance will receive a high evaluation, regardless of a poor performance on the rest of the instances in the training set. The fitness function in this investigation estimates the performance of each heuristic with respect to the performance of the others. If one instance is extremely hard, the performance of one heuristic in such instance will always lie in the fixed range $[0, 1]$, regardless of how expensive the search is. In general, the fitness function used in this investigation guides the evolutionary process to heuristics that are good on many instances of the training set. In other words, the genetic algorithm is looking for general high quality heuristics instead of very specialized ones.

V. Experiments and Results

In this section, we describe the instances used in this investigation, the experiments conducted and the results obtained. As an overview, the first set of experiments produces and analyses heuristics on seen and unseen instances of the same class that was used for training. The second experiment tests the heuristics obtained in the first experiment on a set of real instances to prove their generality. The final part of this section presents the analysis of two of the heuristics obtained through the proposed approach.

The CSP solver used for this investigation incorporates constraint propagation by using the AC3 algorithm [44] and also implements backjumping [45]. All the experiments were conducted on a 3.6 Ghz. 8-core processor Windows 7 machine with 16 GB of RAM.

A. Description of the Instances Used

For this study, we have only considered those CSPs in which the domains are discrete, finite sets and the constraints involve only one or two variables (binary constraints). Rossi et al. [46] proved that for every general CSP there is an equivalent binary CSP. Thus, all general CSPs can be reduced into a binary CSP. All the random instances used in this investigation were produced with model F. In model F [47], we select uniformly, independently and with repetitions, a proportion $p_1 \times p_2$ conflicts out of the $m^2 n(n-1)/2$ possible. We then generate a constraint between each pair of connected variables in the graph until we have exactly $p_1 \times n \times (n-1)/2$ edges and throw out any conflicts that are not between connected variables in this graph. Model F has proved to be one of the most robust random CSPs generators because it is a generalization of the well studied model E [48]. In model F, p_1 and p_2 determine the constraint density and tightness of the instance generated. The constraint density is a measure of the proportion of constraints within the instance; the closer the value of p_1 to 1, the larger the number of constraints within the instance. The constraint tightness (p_2) indicates the proportion of the conflicts within the constraints.

With model F, we produced two simple classes of random instances based on the constraint density (sparse or dense) and tightness (high or low) of the whole instance. All the instances have 25 variables and 10 values in their domains. The values of p_1 and p_2 were randomly selected according to one of the following: class A (sparse constraints, low tightness) $p_1 = [0.2, 0.3]$, $p_2 = [0.2, 0.3]$ and class B (dense constraints, high tightness) $p_1 = [0.7, 0.8]$, $p_2 = [0.7, 0.8]$. Class A contains only satisfiable instances, while class B contains only unsatisfiable ones. For each class we produced two sets, one for training the new heuristics and the other used exclusively for testing purposes. Each set is named according to the class of instances it contains. Thus, we produced four instance sets: training sets A and B; and test sets A and B. Each training set contains 25 instances while the test sets contain 500 instances each. Thus, a total of 1050 random instances were generated for this research.

Along with the sets of random instances, a small set of real instances was also analysed in this investigation. These instances are considered for this investigation to confirm that the heuristics produced can also be applied on structured instances with acceptable performance. The set of real instances corresponds to the set 'driver' that can be downloaded from http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/driver.tgz. The seven instances are satisfiable and are considerably larger than the random instances generated in this investigation by using model F.

TABLE I
Consistency checks saved by the heuristics produced for class A when compared against the best heuristic for this class and the median cost of the four standard heuristics.

Heuristic	Training set		Test set	
	BEST	MEDIAN	BEST	MEDIAN
H_{A01}	4.56%	13.73%	4.58%	11.73%
H_{A02}	2.41%	11.79%	1.73%	9.09%
H_{A03}	2.41%	11.79%	1.73%	9.09%
H_{A04}	2.41%	11.79%	1.73%	9.09%
H_{A05}	3.30%	12.60%	1.89%	9.24%
H_{A06}	1.35%	10.83%	2.08%	9.42%
H_{A07}	3.73%	12.99%	4.73%	11.87%
H_{A08}	1.67%	11.12%	2.46%	9.77%
H_{A09}	1.95%	11.37%	3.80%	11.01%
H_{A10}	2.93%	12.27%	4.67%	11.81%

B. Generation of New Heuristics

A new heuristic is the result of each run of the genetic programming hyper-heuristic. The heuristic function (coded in a tree-based data structure) with the largest fitness obtained during the evolutionary process is returned as the resulting heuristic. For each class of instances we produced 10 new heuristics. Thus, 20 runs of the genetic algorithm were conducted and 20 different heuristics were obtained. In this first experiment we tested the quality of the new heuristics on the sets corresponding to the class they were trained for. Tables I and II present the results of this experiment.

We can observe that all the heuristics produced by our methodology reduce the cost obtained by the best standard heuristic for each class. Although there is no statistical evidence that suggests that the real means of the new heuristics and the best standard heuristics for each set are different (with

TABLE II
CONSISTENCY CHECKS SAVED BY THE HEURISTICS PRODUCED FOR CLASS B WHEN COMPARED AGAINST THE BEST HEURISTIC FOR THIS CLASS AND THE MEDIAN COST OF THE FOUR STANDARD HEURISTICS.

Heuristic	Training set		Test set	
	BEST	MEDIAN	BEST	MEDIAN
H_{B01}	5.90%	11.17%	4.33%	9.64%
H_{B02}	1.09%	6.64%	0.12%	5.66%
H_{B03}	5.90%	11.17%	4.26%	9.57%
H_{B04}	5.90%	11.17%	4.23%	9.55%
H_{B05}	6.12%	11.38%	4.30%	9.61%
H_{B06}	6.13%	11.40%	4.20%	9.52%
H_{B07}	5.90%	11.17%	4.44%	9.74%
H_{B08}	4.31%	9.67%	3.92%	9.25%
H_{B09}	4.80%	10.14%	4.39%	9.69%
H_{B10}	5.90%	11.17%	4.31%	9.62%

5% of significance), the savings obtained are indeed, significant in practice. The heuristics produced are able to reduce the cost of the search compared to the best standard heuristics on each set, which provides evidence that the approach is able to evolve heuristics that specialize for the classes provided as input during the training process. We also compared the heuristics produced by our hyper-heuristic against the median cost of the four heuristics. In practice, we may not be interested in finding a heuristic that is always better than a specialized method (the cost of producing such method may not be feasible), but a general heuristic that performs acceptably on most of the instances. When the new heuristics are compared against the median cost of the four standard heuristics, important savings of more than 9% consistency checks are achieved by the new heuristics in almost all the cases (the performance of H_{B02} will be discussed later).

Regarding how well these heuristics generalize to unseen instances, for both classes of instances, competitive heuristics produced for the training sets are also competitive on the test sets (see for example H_{A01}, H_{A07}, H_{B05} and H_{B06}). Thus, the evidence suggests that the approach produces heuristics which are specialized for the class of instances used for training, but not over-fitted for such training set. The heuristics produced by using the genetic programming hyper-heuristic are capable of being used on unseen instances of the same class used for training and still be competitive with respect to the best standard heuristic for that class of instances.

C. Testing the Heuristics on Real Instances

We tested the 20 heuristics produced in Sec. V-B on a set containing real instances. In general, the heuristics perform well when compared against the median result of the four standard heuristics. In all cases, the heuristics produced save more than 73% of the consistency checks with respect to the median cost of the standard heuristics. Nevertheless, the results are not that encouraging when the heuristics produced are compared against the best heuristic for the set; in this case, KAPPA. Only two of the heuristics produced with our methodology were able to reduce the cost produced by KAPPA on the set of real instances. H_{A09} and H_{B02} proved to be very competitive heuristics for this set, saving 33% and 71.31% consistency checks with respect to KAPPA, respectively. Although the large reduction in the number of consistency checks achieved by H_{B02}, the statistical evidence suggests, with 5% of significance, that both H_{B02} and KAPPA may not be different. It is important to remark that all the instances in this set are satisfiable. Table I shows that H_{A09} achieved a high performance on instances from class A (that also contains only satisfiable instances). H_{A09} was trained for satisfiable instances, then it is not surprising that it is able to generalize well on the instances in the set of real satisfiable instances. On the contrary, the performance of H_{B02} on instances from class B is below the one of the other heuristics trained for unsatisfiable instances (class B contains only unsatisfiable instances). H_{B02} is not as competitive as other heuristics on instances from class B but it results in the best heuristic for the set of real instances. It is not as specialized for instances from class B as others, but it generalizes well to a completely different type of instances.

Trying to understand what these heuristics are doing, we present the heuristic functions obtained for H_{A09} and H_{B02}:

$$\begin{aligned} H_{A09} &= (0.2592 - dom(x))deg(x) + conflicts(x) + 0.1637 \\ H_{B02} &= 2.1127 - \kappa(x) \end{aligned} \tag{5}$$

The reader must recall that, because of the generic interpreter used, the variables that minimise the output of the heuristic function are instantiated first. Also, all the features that describe the variables are normalized in the range $[0, 1]$. The analysis of H_{B02} is straight forward: the new heuristic has defined a threshold on when to rely on the feature $\kappa(x)$. For $\kappa(x) > 2.1127$, the evaluation of the heuristic function will return negative numbers, and because the generic interpreter selects first the variable that minimises the value of the heuristic function, the variable with the largest $\kappa(x)$ will be tried first (just as the standard KAPPA heuristic). On the other hand, when $\kappa(x) \leq 2.1127$, the evaluation of the heuristic function will return only positive numbers and then, it will prefer the variable with the smallest $\kappa(x)$. The function will always try first those variables with large values of $\kappa(x)$, because any $\kappa(x) \geq 2.1127$ will produce a result which is always smaller than any other produced by the evaluation of the heuristic function when $\kappa(x) \leq 2.1127$. Then, H_{B02} is a revised version of KAPPA that only prefers variables with small values of $\kappa(x)$ when the maximum $\kappa(x)$ is less or equal than 2.1127 among all the remaining variables. This small change produces important savings in the number of consistency checks in the set of real instances.

The case of H_{A09} is not as direct to interpret. H_{A09} is a combination of three features that describe the variables: $dom(x)$, $deg(x)$ and $conflicts(x)$. Only variables with small domain sizes will make $0.2592 - dom(x) > 0$. For variables with large domain sizes the result will be a negative number that, when multiplied by $deg(x)$, will produce a larger negative number. Thus, if a variable has a large domain size, the first part of the heuristic function will produce only negative numbers. Variables with small domains (below 0.2592) will

guarantee that $(0.2592 - dom(x))deg(x) > 0$, and the smaller the value of $deg(x)$, the more likely that the generic interpreter prefers such variable. On the contrary, variables with large domains will always produce negative numbers and the larger the value of $deg(x)$, the more likely it is that such variable is selected first. In H_{A09}, small values of $conflicts(x)$ are always preferred and the constant 0.1637 works as a bias, in the same way we analysed H_{B02}.

At this point we consider important to mention that the proposed approach sometimes produces functions that can be simplified into trivial expressions. For example, Fig. 3 shows heuristic H_{B07}. As we can observe, H_{B07} apparently considers two features: the domain size of the variable and its degree. But, the heuristic function described by the tree-based data structure $dom(x)(deg(x) - deg(x)) - deg(x)$ will be reduced to $-deg(x)$. In fact, H_{B07} is a trivial heuristic but not a bad one for instances from class B.

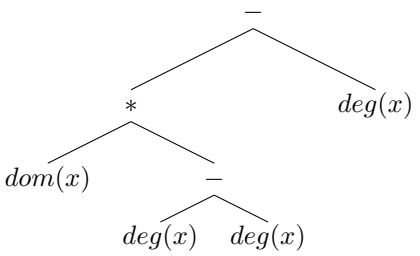

Fig. 3. A heuristic function that can be reduced into a trivial expression

The same problem can occur in other cases. For example, it may be the case that one operation in the heuristic function involves two constants. In that case, it is clear that it should be useful to reduce that operation to a single node with the result of the operation between the two constants. We are aware that as part of the future research, we need to implement a module to optimise and reduce the expressions coded in the tree-based data structures to avoid cases like the ones described in these examples.

VI. Conclusion and Future Work

We have presented an approach for the automated generation of new heuristics for CSPs. Different heuristics use different features to rank the variables and decide which one will be instantiated next. Our approach assumes that a good heuristic requires more than one feature to make its decisions. The problem is that, there are too many ways in which the different features can be combined. Our genetic programming hyper-heuristic explores the space of combinations of features and finds functions, represented as tree-based data structures, that correctly rank the variables. The orderings produced by the new heuristics represent important savings in the cost of the search with respect to the best standard heuristic in each class.

The use of a generic interpreter provides a second level of generality. Similar functions produced by our genetic programming hyper-heuristic may be improved by just changing the interpreter. For example, a change from minimization to maximization inside the interpreter produces the inverse of the original heuristic. The exploration of the benefits of the interpreter is a whole new area of research. For example, we can make any decision with the rankings returned by the evaluation of the function inside the interpreter. What would be the impact in the cost if we change from selecting the minimum or the maximum $f(x)$ and select the second minimum or the second maximum? How can the performance of a heuristic be affected if the interpreter prefers the variable which $f(x)$ is close to the mode or the median of all the values in $f(x)$? We expect to extend our research into this direction in the future.

As part of the future work we plan to include new operations to the set of functions. Also, more features will be included in the set of terminals. We have the belief that if we include more descriptive features, the approach will be able to provide better results. Of course, it may be the case that some of the features are redundant or useless, in which case their incorporation to the model will only produce noise. A deeper analysis about the features used to characterize the variables is to be conducted as part of the future work.

References

[1] E. C. Freuder and A. K. Mackworth, *Constraint-Based Reasoning*. Cambridge: MIT/Elsevier, 1994.

[2] N. Dunkin and S. Allen, "Frequency assignment problems: Representations and solutions," University of London, Tech. Rep. CSD-TR-97-14, 1997.

[3] J. Berlier and J. McCollum, "A constraint satisfaction algorithm for microcontroller selection and pin assignment," in *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, march 2010, pp. 348–351.

[4] S. Minton, A. Phillips, and P. Laird, "Solving large-scale CSP and scheduling problems using a heuristic repair method," in *Proceedings of the 8th AAAI Conference*, 1990, pp. 17–24.

[5] W. W. Chu and P. Ngai, "A dynamic constraint-directed ordered search algorithm for solving constraint satisfaction problems," in *Proceedings of the 1st international conference on Industrial and engineering applications of artificial intelligence and expert systems (IEA/AIE'88)*, vol. 1. ACM Press, 1988, pp. 116–125.

[6] N. Jussien and O. Lhomme, "Local search with constraint propagation and conflict-based heuristics," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press / The MIT Press, 2000, pp. 169–174.

[7] J. R. Bitner and E. M. Reingold, "Backtrack programming techniques," *commun. of the ACM*, vol. 18, no. 11, pp. 651–656, November 1975.

[8] S. Bain, J. Thornton, and A. Sattar, "Evolving algorithms for constraint satisfaction," in *Congress on Evolutionary Computation 2004 (CEC2004)*, vol. 1, june 2004, pp. 265–272.

[9] E. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, and S. Shulenburg, "Hyper-heuristics: an emerging direction in modern research technology," in *Handbook of metaheuristics*. Kluwer Academic Publishers, 2003, pp. 457–474.

[10] P. Cowling, G. Kendall, and E. Soubeiga, "A hyperheuristic approach to scheduling a sales summit," in *Practice and Theory of Automated Timetabling III : Third International Conference (PATAT'00)*, ser. Lecture Notes in Computer Science, E. K. Burke and W. Erben, Eds., vol. 2079. Springer-Verlag, August 2000, pp. 176–190.

[11] H. Fisher and G. L. Thompson, "Probabilistic learning combinations of local job-shop scheduling rules," in *Factory Scheduling Conference*. Carnegie Institute of Technology, 1961.

[12] W. B. Crowston, F. Glover, G. L. Thompson, and J. D. Trawick, "Probabilistic and parametric learning combinations of local job shop scheduling rules," *Office of Naval Research: Research memorandum*, p. 117, 1963.

[13] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "A classification of hyper-heuristic approaches," in *Handbook of Metaheuristics*, ser. International Series in Operations Research & Management Science, M. Gendreau and J.-Y. Potvin, Eds. Springer US, 2010, vol. 146, pp. 449–468.

[14] S. Petrovic and R. Qu, "Case-based reasoning as a heuristic selector in a hyper-heuristic for course timetabling problems," in *Proceedings of the 6th International Conference on Knowledge-Based Intelligent Information Engineering Systems and Applied Technologies (KES'02)*, vol. 82, September 2002, pp. 336–340.

[15] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan, "Using case-based reasoning in an algorithm portfolio for constraint solving," *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[16] S. L. Epstein, E. C. Freuder, R. Wallace, A. Morozov, and B. Samuels, "The adaptive constraint engine," in *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, ser. CP '02. London, UK, UK: Springer-Verlag, 2002, pp. 525–542.

[17] B. Crawford, R. Soto, C. Castro, and E. Monfroy, "A hyperheuristic approach for dynamic enumeration strategy selection in constraint satisfaction," in *Proceedings of the 4th international conference on Interplay between natural and artificial computation: new challenges on bioinspired applications - Volume Part II*, ser. IWINAC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 295–304. [Online]. Available: http://dl.acm.org/citation.cfm?id=2009542.2009574

[18] J. C. Ortiz-Bayliss, H. Terashima-Marín, and S. E. Conant-Pablos, "Learning vector quantization for variable ordering in constraint satisfaction problems," *Pattern Recogn. Lett.*, vol. 34, no. 4, pp. 423–432, Mar. 2013.

[19] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward, "Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1559–1565.

[20] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "Exploring hyper-heuristic methodologies with genetic programming," in *Computational Intelligence*, ser. Intelligent Systems Reference Library, C. Mumford and L. Jain, Eds. Springer Berlin Heidelberg, 2009, vol. 1, pp. 177–201.

[21] E. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, J. A. Vazquez-Rodriguez, and M. Gendreau, "Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms," in *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC'10)*, July 2010, pp. 1–8.

[22] J. R. Koza, *Genetic Programming*. MIT Press, 1992.

[23] D. Jakobovic, L. Jelenkovic, and L. Budin, "Genetic programming heuristics for multiple machine scheduling," in *Proceedings of the 10th European conference on Genetic programming*, ser. EuroGP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 321–330.

[24] N. B. Ho and J. C. Tay, "Evolving dispatching rules for solving the flexible job-shop problem," in *IEEE Congress on Evolutionary Computation*, vol. 3, 2005, pp. 2848–2855.

[25] E. Özcan and A. J. Parkes, "Policy matrix evolution for generation of heuristics," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 2011–2018.

[26] K. Sim, E. Hart, and B. Paechter, "A hyper-heuristic classifier for one dimensional bin packing problems: improving classification accuracy by attribute evolution," in *Proceedings of the 12th international conference on Parallel Problem Solving from Nature - Volume Part II*, ser. PPSN'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 348–357.

[27] A. Lokketangen and R. Olsson, "Generating meta-heuristic optimization code using ADATE," *Journal of Heuristics*, vol. 16, no. 6, pp. 911–930, 2010.

[28] A. S. Fukunaga, "Automated discovery of local search heuristics for satisfiability testing," *Evolutionary Computation*, vol. 16, no. 1, pp. 31–61, March 2008.

[29] A. Runka, "Evolving an edge selection formula for ant colony optimization," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 1075–1082.

[30] J. Drake, N. Kililis, and E. Özcan, "Generation of VNS components with grammatical evolution for vehicle routing," in *Genetic Programming*, ser. Lecture Notes in Computer Science, K. Krawiec, A. Moraglio, T. Hu, A. S. Etaner-Uyar, and B. Hu, Eds. Springer Berlin Heidelberg, 2013, vol. 7831, pp. 25–36.

[31] N. Pillay, "Evolving hyper-heuristics for the uncapacitated examination timetabling problem," *Journal of Operational Research Society*, vol. 63, no. 1, pp. 47–58, 2012.

[32] M. Bader-El-Den, R. Poli, and S. Fatima, "Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework," *Memetic Computing*, vol. 1, no. 3, pp. 205–219, 2009.

[33] R. Poli, J. Woodward, and E. Burke, "A histogram-matching approach to the evolution of bin-packing strategies," in *IEEE Congress on Evolutionary Computation (CEC 2007)*, 2007, pp. 3500–3507.

[34] E. Burke, M. Hyde, G. Kendall, and J. Woodward, "A genetic programming hyper-heuristic approach for evolving 2-d strip packing heuristics," *IEEE Transactions on Evolutionary Computation*, vol. 14, no. 6, pp. 942–958, 2010.

[35] S. Allen, E. K. Burke, M. Hyde, and G. Kendall, "Evolving reusable 3d packing heuristics with genetic programming," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 931–938.

[36] E. Burke, M. Hyde, and G. Kendall, "Grammatical evolution of local search heuristics," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 3, pp. 406–417, 2012.

[37] L. Hong, J. Woodward, J. Li, and E. Özcan, "Automated design of probability distributions as mutation operators for evolutionary programming using genetic programming," in *Genetic Programming*, ser. Lecture Notes in Computer Science, K. Krawiec, A. Moraglio, T. Hu, A. S. Etaner-Uyar, and B. Hu, Eds. Springer Berlin Heidelberg, 2013, vol. 7831, pp. 85–96.

[38] S. Minton, "An analytic learning system for specializing heuristics," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*. Morgan Kaufmann, 1993, pp. 922–929.

[39] S. Bain, J. Thornton, and A. Sattar, "Methods of automatic algorithm generation," in *PRICAI 2004: Trends in Artificial Intelligence*, ser. Lecture Notes in Computer Science, C. Zhang, H. W. Guesgen, and W.-K. Yeap, Eds. Springer Berlin Heidelberg, 2004, vol. 3157, pp. 144–153.

[40] C. Bessière and J. C. Régin, "MAC and combined heuristics: Two reasons to forsake FC (and CBJ) on hard problems," in *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, 1996, pp. 61–75.

[41] R. Wallace, "Analysis of heuristic synergies," in *Recent Advances in Constraints*, ser. Lecture Notes in Computer Science, B. Hnich, M. Carlsson, F. Fages, and F. Rossi, Eds. Springer Berlin / Heidelberg, 2006, vol. 3978, pp. 73–87.

[42] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and 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)*, 1996, pp. 179–193.

[43] S. Minton, M. D. Johnston, A. Phillips, and P. Laird, "Minimizing conflicts: A heuristic repair method for CSP and scheduling problems," *Artif. Intell.*, vol. 58, pp. 161–205, 1992.

[44] A. K. Mackworth, "Consistency in networks of relations," *Artif. Intell.*, vol. 8, no. 1, pp. 99–118, 1977.

[45] J. Gaschnig, "Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems," in *Proceedings of the Canadian Artificial Intelligence Conference*, 1978, pp. 268–277.

[46] F. Rossi, C. Petrie, and V. Dhar, "On the equivalence of constraint satisfaction problems," in *Proceedings of the 9th European Conference on Artificial Intelligence*, 1990, pp. 550–556.

[47] E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh, "Random constraint satisfaction: Theory meets practice," in *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP 98)*. Springer-Verlag, 1998, pp. 325–339.

[48] D. Achlioptas, M. S. O. Molloy, L. M. Kirousis, Y. C. Stamatiou, E. Kranakis, and D. Krizanc, "Random constraint satisfaction: A more accurate picture," *Constraints*, vol. 6, no. 4, pp. 329–344, 2001.