Automatic Generation of Heuristics for Constraint Satisfaction Problems

José Carlos Ortiz-Bayliss[1], Jorge Humberto Moreno-Scott[2] and Hugo Terashima-Marín[2]

[1] Automated Scheduling, Optimisation and Planning (ASAP)
School of Computer Science, University of Nottingham, UK
Jose.Ortiz_Bayliss@nottingham.ac.uk
[2] Tecnológico de Monterrey, Campus Monterrey, Mexico
Jorge.Moreno@arrisi.com, terashima@itesm.mx

Abstract. The constraint satisfaction problem (CSP) is a generic problem with many applications in different areas of artificial intelligence and operational research. When solving a CSP, the order in which the variables are selected to be instantiated has a tremendous impact in the cost of finding a solution. In this paper we explore a novel type of heuristic that combines different features that describe the current state of the instance to decide which variable to instantiate next. A generational genetic algorithm is used to automatically tune the parameters used by these new heuristics. This paper contributes to the development of new heuristics that can be either very specialized to one class of instances, or general enough to deal with different classes of instances with an acceptable performance.

Keywords: Constraint Satisfaction, Heuristics, Genetic Algorithms

1 Introduction

A CSP is defined by a set of variables X, where each variable is associated a domain D of values subject to a set of constraints C [25]. The goal is to find a consistent assignment of values to variables in such a way that all constraints are satisfied, or to show that a consistent assignment does not exist. There is a wide range of theoretical and practical applications of CSPs that include scheduling, frequency assignment, micro-controller selection and pin assignment, among others (see for example [9] and [4]).

In CSPs, the selection of the next variable to instantiate determines the way the solution space is explored. Different orderings for instantiation of the variables produce different exploration patterns, and different patterns have different costs [1]. Then, if we decide correctly, we can find a solution which cost

[1] In this research, we refer to the cost of finding a solution, not the cost of the solution itself. All the solutions to one instance are equally valid because the problem is treated as a combinatorial one. The cost of the search can be measured in terms of time, expanded nodes or consistency checks, to mention some.

is smaller than the others. The general idea in this investigation is to describe a methodology to automatically produce variable ordering heuristics based on the combination of the criteria of existing ones. The approach produces a linear combination of the features that describe the variables within a CSP instance and uses that combination to rank each variable and decide the next one to be instantiated.

This paper is organized as follows. Section 2 presents a brief survey of works relevant to this research. In Sec. 3 some important heuristics and the features they use to order the variables are discussed. The methodology proposed is described in detail in Sec. 4. The experimental set up and the main results are presented in Sec. 5. Finally, the conclusion and future work of this investigation are discussed in Sec. 6.

2 Background

In the last years we have witnessed a rapid growth of developments oriented to improve how heuristics work. Two trends are clearly identified: methods that optimize the use of existing heuristics and methods that construct new heuristics. Regarding the methods that optimize the use of 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 include dynamic algorithm portfolios like CP-Hydra [18, 20] and ACE [10] and selective hyper-heuristics [8, 19, 23]. On the other hand, methodologies that produce new heuristics identify critical parts of existing heuristics to produce new ones [6, 7]. In this paper we will explore the second trend, the one that produces new heuristics based on some components of existing ones.

Our approach is related to the automated parameter tuning problem, which consists in adjusting the parameters of an algorithm without the intervention of the user [22]. As we will see later, the approach proposed represents heuristics as functions. These functions are very similar to the ones used by linear regression (in linear regression these functions are known as hypotheses). Although the representation of the functions is similar, the mechanisms to adjust the parameters are completely different. In the case of linear regression, as a supervised learning mechanism, it requires training examples. In our approach, we do not need any training examples, because our model is unsupervised. To perform the tuning of the parameters, we use a generational genetic algorithm [8, 13].

In the domain of CSPs, one of the first ideas about automatic heuristic generation was proposed by Minton et al. [16]. Their system, MULTI-TAC, produced programs that represented heuristics designed for systematic algorithms. More recently, Bain et al. [2, 3] proposed the use of genetic programming to generate 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. The main difference between their work and the one presented in this document is the set of features used to construct the heuristics. Bain et al. [2, 3] use features that obtain information from the whole instance, while we use information of each individual variable. Although both approaches rely on evolutionary algorithms to produce heuristics, the representation of the heuristics produced is completely different for both approaches.

3 Descriptors and Variable Ordering Heuristics

Although heuristics that decide the next variable to instantiate are referred to as "variable ordering heuristics" in the literature, we do not think this is the correct term to describe what these heuristics currently do. The term comes from the first works on CSPs where the variables were ordered before the search and the order was kept until the search was over. Nowadays, this 'ordering' is performed via a dynamic fashion, where the heuristics decide, at each stage of the search, which variable will be instantiated next. Then, if a heuristic orders all the variables at a certain point of the search, it will only use the first one of the list, because it has no guarantee that once that variable is instantiated, the ordering at the next stage of the search will remain as it was in the previous one. The properties of the instances change as the search progresses and so the decisions made by the heuristics. For this reason, and to help explaining some concepts in this investigation, we will assume that all the variable ordering heuristics described in this document return only one variable at the time, the next one to be instantiated. This assumption simplifies our analysis without loss of generality.

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

Fig. 1. Generic heuristic model

In a general way, we can see any variable ordering heuristic as a procedure that receives a set of uninstantiated variables X and a heuristic function $f(x)$; and returns the variable to instantiate. Thus, each specific heuristic ranks the variables in X according to the values returned by $f(x)$, $\forall x \in X$. Depending on how the heuristic is designed, the heuristic will prefer variables with large values of $f(x)$ over smaller ones, or vice versa. By changing the sign of the values returned by $f(x)$ we can automatically invert the preference of the heuristic. With this idea on mind, we propose the generic heuristic model shown in Fig. 1. Given the proper function $f(x)$, this generic heuristic interpreter is able to represent any specialized heuristic. For example, the min-domain heuristic [5] prefers the variable with the minimum domain size. In our generic heuristic model, the heuristic function for min-domain is $f(x) = dom(x)$, where $dom(x)$ returns the domain size of variable x. Thus, min-domain will instantiate first the variable that minimizes $f(x)$ among all the uninstantiated variables. If we decide to select the variable that maximizes $f(x)$, we obtain a different heuristic known as max-domain but, based also on the domain size. By using the generic heuristic interpreter described in Fig. 1, max-domain should be implemented by the function $f(x) = -dom(x)$. Similar functions can be defined for other variable ordering heuristics by using different features. This simple example shows how we can represent two simple heuristics by using our generic heuristic model and the proper function $f(x)$. In the case of min-domain and max-domain, $f(x)$ only considers one feature of the variables, the domain size. To produce more complex heuristics, more information needs to be obtained from the variables. Each one of these pieces of information is gathered by what we call 'descriptors' that extract information from the variables at a certain point of the search. In the next lines we will present the descriptors proposed for this investigation.

Constraint Density, $p_1(x)$. The constraint density of a variable is defined as the proportion of constraints with other variables over the maximum number of bidirectional constraints the variable can participate in. Given a CSP instance with n variables, the maximum number of bidirectional constraints for a variable is $n-1$. No unary constraints are considered for this calculation. Then, $p_1(x)$ is calculated as the degree of the variable, $deg(x)$ (the number of constraints with other uninstantiated variables) over the maximum number of possible constraints: $p_1(x) = deg(x)/(n-1)$. If we select the variable with the largest constraint density we obtain the max-density heuristic (also known as deg [24]), which prefers the variable involved in the largest number of constraints.

Constraint Tightness, $p_2(x)$. The constraint tightness indicates the proportion of conflicts within the constraints in which the variable is involved. A conflict is a pair of values $\langle a, b \rangle$ that is not allowed for two variables at the same time. Should we prefer the variable with the largest constraint tightness we would obtain the max-tightness heuristic.

Domain size, $\hat{dom}(x)$. As mentioned before, selecting the variable that minimizes $dom(x)$ gives place to the min-domain heuristic [5]. To restrict the range to the interval $[0, 1]$, we use $\hat{dom}(x)$ instead of $dom(x)$, where $\hat{dom}(x)$ is defined as the domain size of variable x divided by the maximum domain size among all the currently uninstantiated variables. This normalization does not modify the behaviour of the heuristic but improves the automatic learning process.

Kappa, $\hat{\kappa}(x)$. Inspired in the κ factor that estimates how restricted a problem is [12], we propose a similar measurement to be used as a descriptor for each variable. $\kappa(x)$ is calculated as:

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

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 max-kappa heuristic [12]. To normalize the values of $\kappa(x)$

we proposed that, for values of $\kappa(x) \leq 5$, the normalized value $\hat{\kappa}(x) = \kappa(x)/5$. Larger values of $\kappa(x)$ will produce $\hat{\kappa}(x) = 1$. This normalization was inspired in results obtained from preliminary studies with the descriptors.

Additionally to these heuristics, we have also included the standard heuristic min-domain/max-density (referred to as dom/deg in [5]). This heuristic became popular because of its simplicity and because it combines two features by using a quotient, $f(x) = \hat{dom}(x)/p_2(x)$. We have decided also to include this heuristic for the experimental phase because of the fact that it is indeed, a heuristic that exploits the use of more than one descriptor to discriminate among variables. Thus, a total of five variable ordering heuristics have been considered for this investigation: max-density, max-tightness, min-domain, max-kappa and min-domain/max-density. In addition to these variable ordering heuristics, the values are ordered according to the min-conflicts heuristic [17]. Once a variable is selected for instantiation, the first value to be tried is the one more likely to success, the one that participates in the fewer conflicts (forbidden pairs of values between two variables). In all cases, lexical ordering is used to break ties.

4 Automatic Generation of Heuristics

We have observed that different descriptors and their combinations allow us to produce different heuristics. Nevertheless, it is not clear which descriptor is more suitable to describe the current problem state and then, make a good decision about the next variable to instantiate. In this paper we propose a new heuristic representation that uses the linear combination of the descriptors presented in Sec. 3 to decide which variable to try next. All the descriptors are used to obtain information about the variables, but a vector of weights determines the importance of the descriptors to make the decision. Let $\boldsymbol{s(x)}$ be the vector that contains all the values of the descriptors for variable x at a certain point of the search, and $\boldsymbol{w}$ a vector of weights (the tuned parameters of the heuristic). Thus, we define the heuristic function as: $f(\boldsymbol{w}, \boldsymbol{s(x)}) = \boldsymbol{w} \cdot \boldsymbol{s(x)}$. As we can observe, the values of the vector of weights $\boldsymbol{w}$ are the same for all the variables, regardless of the instance. What changes at each step of the search are the values of the vector of descriptors $\boldsymbol{s(x)}$. Each heuristic makes decisions based on its internal heuristic function and the heuristic generic interpreter that decides how to deal with the values of the heuristic function. Depending on how the heuristic is designed, it may prefer large or small values of $f(x)$. The representation proposed for the heuristic function allows both preference schemes. Let us assume that our generic heuristic model prefers the variables with small values of $f(\boldsymbol{w}, \boldsymbol{s(x)})$. Then, min-domain (which prefers the variable with the smallest domain size) will be defined by $\boldsymbol{w} = (1)$ and $\boldsymbol{s(x)} = (\hat{dom}(x))$. When $f(\boldsymbol{w}, \boldsymbol{s(x)})$ is calculated, variables with small domain sizes will return small values, and because the heuristic prefers small values of $f(\boldsymbol{w}, \boldsymbol{s})$, it will behave exactly as min-domain. On the contrary, if $\boldsymbol{w} = (-1)$, variables with large domain sizes will obtain large negative values from the heuristic function $f(\boldsymbol{w}, \boldsymbol{s(x)})$ and the heuristic will behave as max-domain.

All the weights in $\boldsymbol{w}$ lie in the range $[-1,1]$. Also, all the components of the vector of descriptors $\boldsymbol{s(x)}$ lie in the range, $[0,1)$. Because of this, given k descriptors to define the state of the variables, the values $f(\boldsymbol{w},\boldsymbol{s(x)})$ can produce are always in the range $(-k,k)$.

4.1 The Genetic Algorithm

To generate new heuristics, we propose the use of a genetic algorithm to adjust the values in $\boldsymbol{w}$ according to the set of descriptors provided. In our implementation, a generational genetic algorithm with memory was used, but other implementations may be considered in the future (for example a steady state genetic algorithm or a messy one). The memory is implemented as a mechanism to always keep the best configuration so far. In this way, if the evolutionary process removes a good individual from the population, we can always use the memory to recall that the best configuration was found before but it is no longer part of the population.

In our genetic algorithm, each individual encodes the values of the vector of weights $\boldsymbol{w}$. Then, each individual determines the way in which the descriptors are to be considered by the heuristic function $f(x)$ and how the heuristic will behave. The individuals are coded by using binary strings, as in standard genetic algorithms. This representation allows us to use standard crossover and mutation operators. Each weight in $\boldsymbol{w}$ is given 10 bits of the individual. As we can see, the length of the individuals is fixed to $10k$ given the number of descriptors, k (where $k=4$ in this investigation). Because there are 10 bits to represent each weight of the descriptor, there are 1024 possible values that can be represented. We divided the range $[-1,1)$ in 1024 uniform steps and (each one of 0.001953125) and according to the value coded in the individual for that weight, the individual is interpreted. For example, if the string corresponding to one component in $\boldsymbol{w}$ is 1100110110 (822 in base 10), the decimal value of that weight is calculated as $-1+(2/1024\times 822)$, which results in -0.6055. When the initial population is created, all the individuals are randomly initialized ('0' and '1' have the same probability of occurring in the string). The fitness of the individuals is calculated as the inverse of the cost of using the vector of weights $\boldsymbol{w}$ coded in that individual to solve all the instances in a training set. The cost is given in terms of consistency checks (the number of revisions of the constraints). Thus, the best individual should minimise the number of consistency checks to solve the whole set. Three genetic operators are used in this investigation. Tournament selection of size two is 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 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. For mutation, all the bits in the strings have the same probability of being affected, 0.0001. When this is the case, the value of the bit is changed to its complement.

5 Experiments and Results

In this section we present the instances used and a detailed description of the experiments conducted. In all cases, the CSP solver used is one implemented in Java by the authors. For all the experiments, the constraint propagation method used was AC3 [15], while backjumping [11] was always used as the strategy for backtracking. All the experiments were conducted on an Intel 8 Core Windows machine with 16 GB of memory.

5.1 Description of the Instances Used

For this research 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. [21] 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 [14], 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 [1]. The values of p_1 and p_2 used for the generation of the instances should not be confused with the descriptors $p_1(x)$ and $p_2(x)$. In model F, p_1 and p_2 determine the constraint density and tightness of the instance generated while our descriptors provide information regarding each variable.

With model F we produced three 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) $p_1 = [0.2, 0.3]$, $p_2 = [0.2, 0.3]$; class B (dense constraints, low tightness) $p_1 = [0.7, 0.8]$, $p_2 = [0.2, 0.3]$ and class C (dense constraints, high tightness) $p_1 = [0.7, 0.8]$, $p_2 = [0.7, 0.8]$. Sets A and B both contain instances with low tightness while classes B and C contain instances with dense constraints. 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. Then, we produced six instance sets: training sets A, B and C; and test sets A, B and C. Each training set contains 25 instances while the test sets contain 500 instances each. Thus, a total of 1575 instances were generated and analysed in this research.

5.2 Generating New Heuristics

For the first experiment we produced three heuristics for each set of instances. Each run of the genetic algorithm produces a heuristic. Thus, nine runs of the genetic algorithm were conducted to obtain the nine heuristics analysed in the first experiment. For each run, the genetic algorithm ran for 50 generations with a population size of 30 individuals. The heuristics produced were trained with instances from one specific class. Table 1 shows the results obtained from this experiment. Each cell in the table indicates the percentage of consistency checks saved or added by using each heuristic produced with respect to the best standard heuristic for each set. For example, H_{A1}, which is the first heuristic produced by our approach when set A was for training, reduced the number of consistency checks required by the best heuristic for class A in 5.08% (the max-density heuristic obtained the best results on class A). The same heuristic, H_{A1}, when applied to test set A achieved a reduction of 4.52% with respect to the best heuristic. Negative numbers indicate a reduction (the heuristic produced was better than the best standard heuristic) and positive numbers indicate additional consistency checks with respect to the best standard heuristic. The best results from the heuristics produced by our approach are marked in bold. The best heuristic for class A was max-density, both on the training and test set. Training set B was best solved by using min-domain/max-density, but max-kappa showed the best performance on test set B. Max-kappa obtained the best results for class C among all the standard heuristics.

Table 1. Performance of each heuristic produced against the best standard heuristic on each set (positive numbers indicate the percentage of consistency checks saved with respect to the best heuristic and negative numbers indicate the percentage of additional consistency checks with respect to the best heuristic).

	Training			Test		
Heuristic	Set A	Set B	Set C	Set A	Set B	Set C
H_{A1}	**-5.08%**	1056.17%	-2.04%	**-4.52%**	381.61%	-1.03%
H_{A2}	**-5.09%**	283.01%	-2.04%	**-4.25%**	166.83%	-2.18%
H_{A3}	**-5.20%**	309.27%	-2.04%	**-4.28%**	173.00%	-2.14%
H_{B1}	7.34%	**-8.79%**	5.46%	6.36%	**-20.16%**	2.06%
H_{B2}	7.83%	**-10.95%**	3.98%	6.51%	**-19.72%**	2.03%
H_{B3}	7.24%	**-10.28%**	0.58%	6.50%	**-24.33%**	1.63%
H_{C1}	19.08%	22.41%	**-7.40%**	15.86%	11.41%	**-4.70%**
H_{C2}	20.76%	21.65%	**-7.34%**	15.75%	16.12%	**-5.23%**
H_{C3}	15.72%	18.09%	**-7.65%**	12.89%	-1.52%	**-4.86%**

It is not surprising that heuristics trained for one specific class obtain the best results on that class in most of the cases, both on training and test sets. The approach proposed is able to produce very competent heuristics for specific classes of instances, but fails (in general) to produce heuristics that can be applied to different classes of instances. These heuristics seem to deal correctly with unseen instances of the same class that was used for training, but tend to perform poorly when tested on instances from other classes. In general, the approach produces very specialized heuristics, suitable for instances of the same class used for training. This is useful if we have, in advance, some idea about the classes of instances that are required to be solved. Thus, we have found evidence that supports the idea that the approach can be used to train on a small subset of instances from a specific class and then, use the specialized heuristic to solve the rest of the instances.

5.3 Generalizing Heuristics

In the previous experiment we found evidence that supports the idea that the approach produces competent heuristics for specific classes of instances. Now, the question is how to make them more general and reusable for different classes of instances.

Heuristics produced by using only one class during the training phase suffer from being over-specialized for such class. They have problems to generalize and being reusable for other classes of instances. In this experiment, we deal with this problem by using combination of classes during the training. In this way, six composed sets of instances were constructed with the instances defined in Sec. 5.1. Training sets AB, BC, and ABC; and test sets AB, BC and ABC were constructed to be used in this new experiment. The sets are formed by the instances used in the previous experiment. For example, test set BC contains the instances from training sets B and C. As we mentioned before, some classes share some properties (for example, classes A and B contain instances with low tightness constraints). We want to observe if the heuristics trained with these new sets are better adapted for dealing with instances of different classes. Also, training set ABC and test set ABC were generated for this experiment. Sets ABC, contain all the instances used so far. We have the belief that heuristics trained on this set would be less specialized (less reduction with respect to the best heuristic, but very consistent among the three classes of instances). As in the previous experiment, three heuristics were produced for each composed set.

Table 2. Performance of each heuristic produced against the best standard heuristic on the composed sets (positive numbers indicate the percentage of consistency checks saved with respect to the best heuristic and negative numbers indicate the percentage of additional consistency checks with respect to the best heuristic)

	Training			Test		
Heuristic	Set AB	Set BC	Set ABC	Set AB	Set BC	Set ABC
H_{AB1}	**-6.49%**	-4.75%	-4.86%	**-3.15%**	-2.82%	-2.77%
H_{AB2}	**-6.68%**	-5.01%	-4.96%	**-9.85%**	-7.23%	-6.73%
H_{AB3}	**-9.96%**	-6.32%	-6.66%	9.81%	6.15%	5.25%
H_{BC1}	**-7.12%**	-5.24%	-5.19%	**-10.05%**	-7.38%	-6.94%
H_{BC2}	**-8.23%**	-5.31%	-5.76%	0.90%	0.33%	**-0.09%**
H_{BC3}	**-9.71%**	-5.62%	-5.28%	**-20.86%**	-13.65%	-12.58%
H_{ABC1}	**-8.85%**	-5.73%	-6.08%	**-4.39%**	-3.24%	-3.36%
H_{ABC2}	**-8.87%**	-5.82%	-6.09%	**-4.97%**	-3.66%	-3.72%
H_{ABC3}	**-9.63%**	-5.57%	-5.23%	**-17.87%**	-11.57%	-10.63%

The results suggest that sets AB (both training and test) are very easy to solve by most of the heuristics produced when the genetic algorithm uses the composed training sets as inputs. Only H_{AB3} and H_{BC2} fail to generalize on test set AB, where they do not achieve reductions in the cost when compared against the best heuristic for the same set.

With the change in the training sets, the approach produces heuristics that, in general, reduce the cost of the search on all the instances used for training, regardless of the set used. Nevertheless, when analysed on the test sets, these heuristics are not always as competent as they were on the training sets (see for example H_{AB1}, H_{AB2} and H_{AB3}) The problem with these heuristics is that they are very specialized for some hard instances in the training sets. For example, if one heuristic gets specialized for the hardest instances in one of the sets, it is likely that that heuristic also performs well on a composed set that contains those instances. But, once we present the heuristic new instances, the heuristic may be so specialized for the hard instances it has already solved that fails to solve the new instances.

In order to understand how these heuristics perform, we present a brief analysis of H_{BC3} and H_{ABC3}. These heuristics obtained the most relevant results on the composed sets.

$$\begin{aligned} H_{BC3} &= -(0.877)p_1(x) + (0.410)p_2(x) + (0.283)\hat{dom}(x) - (0.193)\hat{\kappa}(x) \\ H_{ABC3} &= -(0.994)p_1(x) + (0.547)p_2(x) + (0.326)\hat{dom}(x) - (0.232)\hat{\kappa}(x) \end{aligned} \tag{2}$$

These heuristics have very similar vectors of weights and then, we expect their decisions to be similar. H_{BC3} and H_{ABC3} should be interpreted in the following way. First, recall that the general heuristic model used in this investigation is using a minimisation approach. Because it has the largest coefficient, $p_1(x)$ is the most relevant descriptor, followed by $p_2(x)$. Because of the differences in the values of the two most important coefficients (a proportion of 2 to 1), most of the decisions will be mainly based on the values of $p_1(x)$ of the remaining variables. Nevertheless, the combination of the values of the remaining descriptors makes these heuristics to behave in a different way than the max-density heuristic. In a very abstract way (and based only on the signs of the coefficients), these heuristics will instantiate first the variable involved in the largest number of constraints, with the fewer conflicts among the constraints where it is involved, with a small domain and with a large value of $\hat{\kappa}(x)$. Because these heuristics are combining the criteria of the single heuristics we expect the combination to be somehow similar to the decisions made by the single heuristics (see Sec. 3 for more details). It is interesting to notice that the way to consider the constraint tightness is inverse to the recommendation of the single and accepted heuristic (max-tightness prefers the variables with the largest values of this descriptor). Also, we are presenting a very simple and straight forward interpretation of the heuristic. The exact interactions between the descriptors –defined by the values of the weights and not only their signs– is what gives the new heuristics their strength.

6 Conclusion and Future Work

This paper describes a methodology to produce heuristics for variable ordering within CSPs by using components of standard heuristics taken from the literature. The components of these standard heuristics are the descriptors of the variables. The approach is able to generate specialized heuristics that reduce the cost of the search when compared against good quality standard heuristics on the same instances, but more research is still needed to prove the real advantages and limits of the approach. The genetic algorithm produces heuristics that perform well on unseen instances of the same classes used for training. We observed that including more diverse instances in the training sets is, in general, a way to produce more flexible heuristics that are capable of performing well on different classes of instances.

We have represented the heuristics as general procedures that take two inputs: the set of uninstantiated variables and a heuristic function. In this investigation, we used a generic heuristic interpreter for a minimisation problem. One important question derives from the use of this interpreter for the heuristic function. How would the use of a more complex interpreter affect the performance of the heuristic? Let us assume that the generic heuristic model prefers large values of $f(x)$ at the beginning of the search and small ones for the last stages. This slightly modified version of the generic heuristic interpreter can produce an algorithm that switches from one heuristic to the opposite one without any other change to the model. We consider this flexibility of the approach another important contribution of this investigation.

By analysing the results, we have identified a potential drawback in the current implementation of the genetic algorithm to update the vector of weights. The fitness function, as it is currently designed, aims to reduce the cost of a heuristic on the instances in the training set. This seemed to be the 'natural' way to measure the quality of the heuristics. But, this approach tends to produce over-fitted heuristics for the instances in the training set. The reason is simple: attempting to reduce the overall cost, the heuristic that best solves the hardest instances in the training set receives the highest values from the fitness function. Thus, if there is one hard instance in the training set, the fitness function will reward competent heuristics for such instance, regardless of their performance on other instances from the training set that may be easier to solve. The problem with the fitness function is even more important when we deal with composed training sets. In general, the use of composed sets facilitates the generalization of heuristics. But, if there is the case that a small subset of really hard instances are contained in two or more of the sets that form the composed one, and the evolutionary process specializes the heuristics for such instances, then we can expect those heuristics to perform well on the training sets but to fail to generalize to unseen instances. In this case, the heuristics only learn how to solve well a very small subset of specific hard instances. We consider to change the design of the fitness function as part of the future work.

Finally, we are interested in extending our results to include real instances to prove the effectiveness of the model to solve structured instances.

7 Acknowledgments

This research was supported in part by ITESM Strategic Project PRY075 "Intelligent Learning for Pattern Recognition and its Application in Medicine and Logistics" and the CONACyT Project under grant 99695.

Bibliography

[1] Achlioptas D, Molloy MSO, Kirousis LM, Stamatiou YC, Kranakis E, Krizanc D (2001) Random constraint satisfaction: A more accurate picture. Constraints 6(4):329–344

[2] Bain S, Thornton J, Sattar A (2004) Evolving algorithms for constraint satisfaction. In: Congress on Evolutionary Computation 2004 (CEC2004), vol 1, pp 265–272

[3] Bain S, Thornton J, Sattar A (2004) Methods of automatic algorithm generation. In: Zhang C, W Guesgen H, Yeap WK (eds) PRICAI 2004: Trends in Artificial Intelligence, Lecture Notes in Computer Science, vol 3157, Springer Berlin Heidelberg, pp 144–153

[4] Berlier J, McCollum J (2010) A constraint satisfaction algorithm for microcontroller selection and pin assignment. In: Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon), pp 348–351

[5] Bessière C, Régin JC (1996) 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, pp 61–75

[6] Burke E, Hyde M, Kendall G, Ochoa G, Ozcan E, Woodward JR (2009) Exploring hyper-heuristic methodologies with genetic programming. In: Mumford C, Jain L (eds) Computational Intelligence, Intelligent Systems Reference Library, vol 1, Springer Berlin Heidelberg, pp 177–201

[7] Burke EK, Hyde MR, Kendall G, Woodward J (2007) Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, New York, NY, USA, GECCO '07, pp 1559–1565

[8] Crawford B, Soto R, Castro C, Monfroy E (2011) 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, Springer-Verlag, Berlin, Heidelberg, IWINAC'11, pp 295–304

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

[10] Epstein SL, Freuder EC, Wallace R, Morozov A, Samuels B (2002) The adaptive constraint engine. In: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, Springer-Verlag, London, UK, UK, CP '02, pp 525–542

[11] Gaschnig J (1978) Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In: Proceedings of the Canadian Artificial Intelligence Conference, pp 268–277

[12] Gent I, MacIntyre E, Prosser P, Smith B, TWalsh (1996) 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

[13] Holland J (1975) Adaptation in Natural and Artificial Systems. The University of Michigan Press

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

[15] Mackworth AK (1977) Consistency in networks of relations. Artificial Intelligence 8(1):99–118

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

[17] Minton S, Johnston MD, Phillips A, Laird P (1992) Minimizing conflicts: A heuristic repair method for CSP and scheduling problems. Artificial Intelligence 58:161–205

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

[19] Ortiz-Bayliss JC, Terashima-Marín H, Conant-Pablos SE (2013) Learning vector quantization for variable ordering in constraint satisfaction problems. Pattern Recognition Letters 34(4):423–432

[20] Petrovic S, Qu R (2002) 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, pp 336–340

[21] Rossi F, Petrie C, Dhar V (1990) On the equivalence of constraint satisfaction problems. In: Proceedings of the 9th European Conference on Artificial Intelligence, pp 550–556

[22] Schwartz S, Wah B (1992) Automated parameter tuning in stereo vision under time constraints. In: Tools with Artificial Intelligence, 1992. TAI '92, Proceedings., Fourth International Conference on, pp 162–169

[23] Soto R, Crawford B, Monfroy E, Bustos V (2012) Using autonomous search for generating good enumeration strategy blends in constraint programming. In: ICCSA (3)'12, pp 607–617

[24] Wallace R (2006) Analysis of heuristic synergies. In: Hnich B, Carlsson M, Fages F, Rossi F (eds) Recent Advances in Constraints, Lecture Notes in Computer Science, vol 3978, Springer Berlin / Heidelberg, pp 73–87

[25] Williams CP, Hogg T (1992) Using deep structure to locate hard problems. In: Proceedings of AAAI'92, pp 472–477