Improving the Performance of Vector Hyper-heuristics through Local Search

José Carlos Ortiz-Bayliss
Tecnológico de Monterrey
Monterrey, Mexico
jcobayliss@gmail.com

Hugo Terashima-Marín
Tecnológico de Monterrey
Monterrey, Mexico
terashima@itesm.mx

Santiago Enrique Conant-Pablos
Tecnológico de Monterrey
Monterrey, Mexico
sconant@itesm.mx

Ender Özcan
University of Nottingham
Nottingham, United Kingdom
Ender.Ozcan@nottingham.ac.uk

Andrew J. Parkes
University of Nottingham
Nottingham, United Kingdom
ajp@cs.nott.ac.uk

ABSTRACT

Hyper-heuristics are methodologies that allow us to selectively apply the most suitable heuristic given the properties of the problem at hand. They can be applied in CSP in different ways, but one way which has received attention in recent years is variable ordering by using hyper-heuristics. To select the next variable, a set of heuristics exist and the hyper-heuristic decides, considering the features that describe the instance at hand, which heuristic is more suitable to be applied at the moment. This paper explores a hyper-heuristic model for variable ordering within CSP based on vector hyper-heuristics. Each hyper-heuristic is represented as a set of vectors that maps instance features to heuristics. These vector hyper-heuristics are constructed by going into a local search method that modifies the hyper-heuristics. The results suggest that the approach is able to combine the strengths of different heuristics to perform well on a wide range of instances and compensate for their weaknesses on specific instances, resulting in an improvement in the performance of the search compared against the heuristics applied in isolation.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods, Graph and tree search strategies*

General Terms

Algorithms

Keywords

Constraint Satisfaction, Variable Ordering, Hyper-heuristics

1. INTRODUCTION

A Constraint Satisfaction Problem (CSP) is defined by a set of n variables, each one with its respective domain of m possible values and a set of constraints, each one restricting the values some variables can take simultaneously. The task when solving a CSP instance is to find one possible assignment for the n variables that satisfies all the constraints. For this research we have only considered those CSP in which the domains are discrete, finite sets and the constraints involve only one or two variables (binary constraints).

CSP are usually solved by using systematic algorithms. Systematic algorithms explore the search tree which is based on the possible values for each of the variables of the problem [11]. These algorithms start from an empty variable assignment that is extended until obtaining a complete assignment that satisfies all the constraints in the problem [20]. For small instances, it is feasible to explore the complete search space and find one solution. For large instances it is not possible to run an exhaustive search due to the exponential growth in the search space with respect to the number of variables and the time needed to explore it. Thus, every CSP can be treated as a classic search problem over a search space and can be solved using any of the conventional search methods like Depth First Search (DFS) [28].

In the search tree, every time a variable is instantiated, the constraints in which that variable is involved must be checked to verify that none of them is violated; this is known as a consistency check. When an assignation violates one or more constraints, the instantiation must be undone, and another value must be considered for that variable. If there are not any more values available, the value of the previous instantiated variable must be changed; this technique is known as backtracking [4]. There are some improvements to this basic search method which try to reduce the number of revisions of the constraints (consistency checks) like constraint propagation [15] and backjumping [16]. The CSP solver used for this investigation incorporates constraint propagation by using the AC3 algorithm [21] and also implements backjumping.

The selection of the next variable to instantiate affects the search complexity and represents an opportunity to optimize the search. Different orderings to instantiate the variables produce different trees, and different trees have different search costs. Then, if we decide correctly, we can construct a search tree which is considerably cheaper than others and save computational effort. When trying to construct this ordering of variables, we know that there are some features that make one instance more suitable to a certain heuristic than others [23]. If these features can be identified, then the problems would be solved more efficiently. The isolated application of heuristics has not proved to be an efficient method for solving sets with a large diversity of instances of the problem. When presented with a problem instance, it may not be obvious which heuristic to apply, even though one of them might end up performing much better than others on a group of instances [2].

A technique that provides some advice about which heuristic is to be best for the current instance would be very useful [19]. A more efficient alternative to the isolated application of heuristics to the whole instance is to apply a different heuristic as the search progresses depending on the current problem state; this is the task of a hyper-heuristic. A hyper-heuristic is used to define a high-level heuristic which controls the application of the heuristics [8].

This paper is organized as follows: Section 2 presents a small survey of other hyper-heuristic works for CSP along with the variable ordering heuristics used in this investigation. Section 3 describes in detail the vector hyper-heuristic and the local search method developed in this research and the instances used. The experiments and main results are shown in Section 4. Finally, Section 5 presents the conclusions and future work.

2. RELATED BACKGROUND

In this section we present a brief description of previous works on hyper-heuristics for CSP and the variable ordering heuristics used in this investigation.

2.1 Hyper-heuristics

The idea of combining algorithms or heuristics goes back to 1960s [14] and has been used in many investigations under different names [34, 29, 36, 12]. Hyper-heuristics are one alternative to combine the strengths of heuristics based on the current problem features.

Hyper-heuristics can be divided into two main classes: those which select from existing heuristics and those that generate new heuristics. A more detailed description about the classification of hyper-heuristics can be found in [9, 7]. In this investigation we will focus our attention on hyper-heuristics that select from existing heuristics.

With respect to CSP, one of the first attempts to systematically map CSP to algorithms and heuristics according to the features of the instances was presented by Tsang and Kwan in 1993 [37]. In that study, Tsang and Kwan presented a survey of algorithms and heuristics for solving CSP and proposed a relation between the formulation of the CSP and the most adequate solving method for that formulation. Also, algorithm portfolios for Constraint programming have been successfully studied before [13]. Petrovic and Eipstein [25] studied the idea of combining various heuristics to produce mixtures that work well on particular classes of instances. More recent studies about the dynamic combination of heuristics applied to CSP include the work done by Terashima-Marín et al. [35], who proposed an evolutionary framework to generate hyper-heuristics for variable ordering in CSP and the research developed by Bittle and Fox [5] who presented a hyper-heuristic approach for variable and value ordering for CSP based on a symbolic cognitive architecture augmented with case based reasoning as the machine learning mechanism for their hyper-heuristics. Ortiz-Bayliss et al [24] recently presented a study where they represent variable ordering hyper-heuristics as integer matrices and a genetic algorithm is used to evolve the structure of the matrices in order to generate hyper-heuristics.

2.2 Variable Ordering

Many researchers have proved the importance of the order of the variables and its impact in the cost of the solution search [26]. The search space grows exponentially with respect to the number of variables, and so does the time for finding the optimal ordering. Once a variable has been selected to be instantiated we need to decide which value, among all the feasible ones, will be used for instantiation. This ordering also has relevance to the search but the selection of value ordering heuristics is beyond the scope of this investigation.

Various heuristic and approximate approaches have been proposed that find good solutions for some instances of the problem. However, it has not been possible to find a reliable method to solve well all instances of CSP. In this study, we have included four variable ordering heuristics: Minimum Remaining Values (MRV) [18], Kappa (K) [17], Maximum Forward Degree (MFD) [38] and Brelaz (BZ) [6].

A solution to any given CSP is constructed selecting one variable at a time based on one of the four variable ordering heuristics used in this investigation. Each one of these heuristics reorders the variables to be instantiated dynamically at each step during the construction process. Later, a value must be selected and assigned to the chosen variable considering the constraints and using MNC as value ordering heuristic depending on the instance features. The ordering heuristics used in this investigation are briefly explained in the following lines:

MRV (also known as Fail First (FF) [30] and Dynamic Search Rearrangement (DSR) [27]) is one of the most simple and effective heuristics to determine which variable to instantiate next [27, 33]. This heuristic selects the variable with the smallest number of available values in its domain. 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.

K selects the variable in such a way that the resulting subproblem minimizes the kappa factor κ, which is defined as [17]:

$$\kappa = \frac{-\sum_{c_i \in C} \log_2(1-p_{c_i})}{\sum_{x_i \in X} \log_2(m_{x_i})} \quad (1)$$

where p_{c_i} is the fraction of forbidden pairs of values by constraint C_i and m_{x_i} is the domain size of variable x_i.

The constrainedness parameter κ represents a notion of how restricted a problem is. It is likely that problems with $\kappa \ll 1$ are less restricted and may have many solutions, while the problems with $\kappa \gg 1$ are likely to be highly restricted and may not be solved [17]. It has been found that hard problems take place when $\kappa \approx 1$. With this heuristic, we branch on a variable that is estimated to be the most constrained, giving the least constrained subproblem; this is, the subproblem with smallest κ.

MFD is a very simple heuristic that selects the variable which instantiation will produce a subproblem that minimizes the number of conflicts among the variables left to instantiate. Then, MFD selects the variable that is connected to the larger number of uninstantiated variables.

BZ (also known as *dom/deg* [3, 32]) was originally developed for graph colouring and has since been applied to CSP. BZ chooses a variable with the smallest remaining domain, but in case of a tie, chooses from these, the variable with the largest future degree, that is the one which constraints the largest number of unassigned variables. It can be observed that BZ is then a composed heuristic; it is essentially MRV but tie breaking on the variable with most constraints acting into a future subproblem [6]. BZ addresses a flaw of the MRV heuristic, that it takes no account of the constraint graph.

In all cases, the heuristics break ties by using the lexical ordering criterion. The variable ordering heuristics are combined with the MNC value ordering heuristic to improve the search. MNC is one simple and commonly used value ordering heuristic for CSP. MNC prefers the value (from the selected variable) that is involved in the minimum number of conflicts [22]. MNC tries to leave the maximum flexibility for subsequent variable assignments.

2.3 CSP Instances and The Problem State Representation

Various studies have been developed to randomly generate instances of binary CSP (see for example [1, 26]), and those studies have shown that random binary CSP have very interesting properties which make them an important topic of study. Binary CSP present two important properties that are used in this research: the constraint density (p_1) and the constraint tightness (p_2). 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) represents a proportion of the conflicts within the constraints. A conflict is a pair of values $\langle x, y\rangle$ that is not allowed for two variables at the same time. The higher the number of conflicts, the more unlikely an instance has a solution. Of course, there are many CSP features that can be used to represent the problem state (for example, κ [17]). We have reduced the problem representation only to p_1 and p_2 because they are the most common properties of CSP instances.

The binary CSP instances included in this research were randomly generated by using the standard model B [31]. In the first stage, a constraint graph G with n nodes is randomly constructed and then, in the second stage, the incompatibility graph C is formed by randomly selecting a set of edges (incompatible pairs of values) for each edge (constraint) in G. The parameter p_1 determines how many

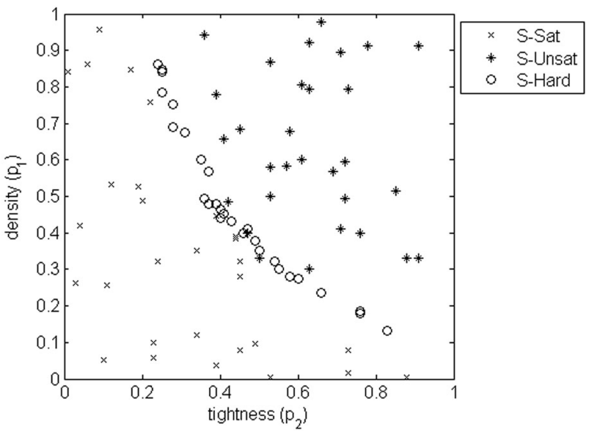

Figure 1: The distribution of the instances contained in S-Sat, S-Unsat and S-Hard in the space $p_1 \times p_2$

constraints exist in the CSP instance, whereas p_2 determines how restrictive the constraints are. In model B, there should be exactly $p_1 n(n-1)/2$ constraints (rounded to the nearest integer), and for each pair of constrained variables, the number of inconsistent pairs of values should be exactly $m^2 p_2$ (where m is the uniform domain size of the variables). To determine the hardness of a given instance, it has been widely accepted to use the κ as a measure of the constrainedness of combinatorial problems [17]. For small values of κ, the instances tend to have many solutions, in contrary to larger values of κ where the instances tend to be unsolvable. For values of $\kappa \approx 1$, a phase transition in solvability takes place [10], a region where the instances change from having many solutions to being unsolvable. In this region, a peak in the median search cost occurs and we can expect that the instances inside this region are hard to solve with respect to their size.

Every time a variable is assigned a new value and the infeasible values are removed from domains of the remaining uninstantiated variables, the values of p_1 and p_2 change and a sub-problem with new features appears. This is the reason why we decided to use the constraint density and tightness to represent the problem state and guide the selection of the heuristics. Our idea is that these two features can be used to describe a CSP instance and to create a relation between instances and heuristics.

The collection of instances used for this investigation includes 90 instances used both for training and testing, and 300 instances used only for testing. The first 90 instances are divided into three small sets: a set of 30 satisfiable instances, a set of 30 unsatisfiable instances and a set of 30 hard instances ($\kappa \approx 1$). From now on we will refer to these sets as S-Sat, S-Unsat and S-Hard, respectively. Each set contains randomly generated instances, with different values of p_1 and p_2. Figure 1 presents the distribution of the instances in S-Sat and S-Unsat in the space $p_1 \times p_2$.

The additional 300 instances used only for testing purposes are contained in Set S-Sat/Unsat. This set includes 150 satisfiable and 150 unsatisfiable instances. These instances were also randomly generated with model B and are randomly distributed in the space $p_1 \times p_2$.

3. THE VECTOR HYPER-HEURISTIC AND THE LOCAL SEARCH METHOD

Any hyper-heuristic that selects among heuristics can be represented as a function that maps from problem features to one variable ordering heuristic. We propose to use a vector representation, where any vector hyper-heuristic will be represented as a set of pairs:

$$HH = \{(\vec{f_0}, h_0), (\vec{f_1}, h_1), \dots (\vec{f_l}, h_k)\} \qquad (2)$$

where $\vec{f_i}$ represents a vector of features, and h_j is the variable ordering heuristic to apply when the instance has a vector of features $\vec{I}$ that is close enough to $\vec{f_i}$. In our representation, all the vectors start at the origin (their magnitude is equal to the distance from the end of the vector to the origin). In this investigation we decided to use the euclidean distance to measure the distance between the input vector $\vec{I}$ (the vector formed by the features of the instance at hand) and $\vec{f_i}$. Then, only one of the vectors contained in HH will be selected given an input vector. The pair that minimizes distance$(\vec{I}, \vec{f_i})$ is selected and the heuristic in the pair is applied.

Each vector hyper-heuristic contains all the information needed to decide which heuristic to apply given an input vector of features $\vec{I}$. For example, let us imagine a vector hyper-heuristic HH_x with only two vectors of features $\vec{f_1} = (0.2, 0.6)$ and $\vec{f_2} = (0.7, 0.3)$ (the values of p_1 and p_2, respectively), and two possible variable ordering heuristics h_1 and h_2. Imagine that $HH_x = \{(\vec{f_1}, h_1), (\vec{f_2}, h_2)\}$. Given an input vector of features $\vec{I} = (0.4, 0.5)$, the heuristic to apply would be h_1 because distance$(\vec{f_1}, \vec{I}) <$ distance$(\vec{f_2}, \vec{I})$.

3.1 The Local Search Method

We define a run of the local search method by the 4-tuple $\langle h_0, n, \delta_d, T\rangle$, where h_0 indicates the default heuristic, n the number of cycles, δ_d the maximum allowed distance between vectors and T the set of instances used for generating the hyper-heuristic. We will describe these parameters in detail in the following lines.

At the beginning of the process, one vector hyper-heuristic is created. The hyper-heuristic contains only one random vector with the default heuristic h_0 attached to it. Every time the hyper-heuristic explores a node of the search tree, a different pair of values (p_1, p_2) is explored. Then, for each instance in T, the hyper-heuristic will try to find one vector f which is close enough to the input vector $\vec{I}$. The hyper-heuristic calculates the distance $d_i =$ distance$(\vec{f_i}, \vec{I})$ for each vector in the hyper-heuristic. The vector with the minimum distance d_i and $d_i < \delta_d$ (where δ_d is the maximum allowed distance to $\vec{I}$), will provide the heuristic to use. If no vector that fulfils the requirements is found, a new vector with the same components of $\vec{I}$ is added to the hyper-heuristic, attaching the default heuristic to the new vector.

After solving all the instances in T, we will have a hyper-heuristic with more than one vector, all attached to the default heuristic. These vectors correspond to the values of p_1 and p_2 of the nodes explored, it is, all the points visited during the search for all the instances in T. Because each vector is attached the same heuristic in the first cycle (the default one), at this point the behaviour of the hyper-heuristic will not be different from the default heuristic. The local search method works based on the idea of changing the heuristics attached to the vectors in the hyper-heuristic, only one change per cycle. Our model changes the vectors with larger magnitudes first. Then, at each cycle of the local search process, the heuristic attached to the vector with the largest magnitude, which has not achieved the maximum number of allowed changes, will be changed. If k heuristics are being used, the maximum number of allowed changes per heuristic will be $k - 1$. Because one heuristic is used to initialize the vectors, $k - 1$ changes are enough to try all the possible heuristics for a given vector. After the change of the heuristic, the hyper-heuristic is used to solve the whole training set again. If the change reduces the average consistency checks required to solve all the instances in the set, the change is kept, otherwise the change is undone and the heuristic returns to the previous configuration. After the change has been accepted or rejected, a new cycle occurs where another vector (it could also be the same vector) will have its heuristic changed. If after the $k - 1$ changes no improvement has been achieved, the vector will stay attached to the default heuristic for the rest of the process. At the end of the n cycles, the hyper-heuristic will contain the set of pairs $(\vec{f_i}, h_j)$ that minimizes the average consistency checks required to solve all the instances in the training set.

The local search method can be summarized in the following steps:

1. Initialize the vector hyper-heuristic HH with one vector attached the default heuristic h_0.

2. Solve the instances in the T with the HH. For each node visited during the search. If no vector that fulfils $d_i < \delta_d$ exists, create a new vector with the values of p_1 and p_2 of the node under exploration and attach the default heuristic.

3. Obtain the average consistency checks for the whole set of instances $avg_0(HH)$ (the result of the solver running with single heuristics).

4. Change the heuristic attached to one of the vectors. Only the heuristic from one vector is changed according to the criteria already described.

5. Solve the instances in the T with the changed hyper-heuristic and obtain the average consistency checks for the set $(avg(HH))$. If $avg(HH) < avg_0(HH)$, make $avg_0(HH) = avg(HH)$ and accept the change. Otherwise, cancel the change and return the vector hyper-heuristic to the previous configuration.

6. Repeat from step 4 until the maximum number of cycles is reached.

3.1.1 *Time Analysis*

The extra time needed to generate a hyper-heuristic is one of the main criticisms to the hyper-heuristic approach. It is true that an additional time is needed to produce one hyper-heuristic and that for a specific instance it may be better to use all the heuristics available and keep the best result. Unfortunately, this is not feasible when we are dealing with a large set of instances. As the number of instances grows, more expensive it will be to solve such instances with different heuristics to obtain the best result. Also, if we could do that, we would observe that the best results are not always achieved by the same heuristic.

To produce one vector hyper-heuristic with the proposed approach, our model requires to solve the instances in T, n times, one per change to the hyper-heuristic. If the time to solve the instances in T with the k heuristics is τ, it can be assumed that the time to solve the instances in T with a given hyper-heuristic is $\frac{\tau}{k}$. Then, the cost of producing one hyper-heuristic with the methodology described is around $n\frac{\tau}{k}$.

Once the vector hyper-heuristic is produced, using it requires a few extra operations compared to the application of the single heuristics. Different heuristics require a different number of operations to decide the next variable to instantiate. For example, K requires more computational effort than MRV to produce a decision, but the difference in practice is not significant. With a hyper-heuristic, extra operations are needed to decide which heuristic to apply and later, the operations of the selected heuristic will be performed. If m nodes are expanded during the search and we use a vector variable ordering hyper-heuristic of l vectors, the extra operations required to apply the hyper-heuristic will be ml. Thus, the extra cost of applying the hyper-heuristic depends both on the number of nodes expanded (number of decisions made) and the number of vectors within the hyper-heuristic. Even though we have identified these differences in the number of operations for heuristics and hyper-heuristics, we have not identified a significant difference in practice with regard to the execution time.

4. EXPERIMENTS AND RESULTS

We conducted three different experiments in this investigation. The first experiment explores the local search method, producing hyper-heuristics that are applied on the same sets that were used for training. The second experiment tries to demonstrate the change in the performance of the heuristics when the instance features change and unseen instances are included. Also, the second experiment tries to show the flexibility of the hyper-heuristics produced in the first experiment and how they can be applied to unseen instances without losing quality. Finally, the third experiment explores the implications of changing the set of heuristics to be used for the hyper-heuristics, and how this affects the performance of the hyper-heuristics produced.

4.1 Experiment I

In this first set of experiments we produced 12 hyper-heuristics. Because each time we create a hyper-heuristic with the proposed approach we use a default ordering heuristic for initialization, four hyper-heuristics can be produced out of the four heuristics. Also, three sets are defined, what makes it possible to produce four distinct hyper-heuristics per set, giving a total of 12 hyper-heuristics. In all cases, the local search process ran for 100 cycles ($n = 100$) and the maximum distance between vectors δ_d was set to 0.2.

Table 1 presents the results of hyper-heuristics HH01-HH04. The columns must be interpreted in the following way: HH indicates the hyper-heuristic ID, obtained in the order the hyper-heuristics were generated, h_0 indicates the default heuristic used for each run of the local search method, Avg_0 is the average number of consistency checks of the hyper-heuristic at the beginning of the process (which is equivalent to the average consistency checks achieved by h_0 on the same set), Avg_n is the average consistency checks required by the hyper-heuristic on the set at the end of the process, and the column 'Reduction' represents the percentage of consistency checks reduced (with respect to h_0 of each run) once the process has finished. Hyper-heuristics HH01-04 were trained by using the instances contained in S-Sat. We can observe that MRV seems to be the best variable ordering heuristic for S-Sat because it has the smaller average number of consistency checks among the four heuristics. The four hyper-heuristics produce very competitive results, each one achieving an average performance which is smaller than the average obtained with MRV (and any other heuristic).

Table 1: Performance of HH01-04 on S-Sat

HH	h_0	Avg_0	Avg_n	Reduction
HH01	MRV	28688	19881	30.6%
HH02	K	77209	22890	70.3%
HH03	MFD	26020	23211	10.7%
HH04	BZ	52815	25664	51.4%

Table 2: Performance of HH05-08 on S-Unsat

HH	h_0	Avg_0	Avg_n	Reduction
HH05	MRV	91832	91149	0.7%
HH06	K	111053	91212	17.9%
HH07	MFD	104682	91212	12.9%
HH08	BZ	132741	91212	31.3%

The case for the hyper-heuristics trained with the instances from S-Unsat is slightly different from the previous experiment. Here, the hyper-heuristics converge to an almost identical performance and the average consistence checks saved by the hyper-heuristics compared against MRV is not significant in practice (less than 0.7%). Even thought the hyper-heuristics achieve a very large reduction with respect to K, MFD and BZ, MRV is still the best option among the heuristics and it seems to be equally good than any of the hyper-heuristics produced.

The set of hard instances, S-Hard, was used as training set to produce hyper-heuristics HH09-12. The results suggest that K is the best heuristic for hard instances. The hyper-heuristics produced reduce the number of consistency checks required by their corresponding default heuristics, but as in the case of unsatisfiable instances, the reduction between the best hyper-heuristic HH09 and K is small, approximately 1.5% (which corresponds to 18807 consistency checks).

4.2 Experiment II

For this experiment we decided to test the performance of the hyper-heuristics on a set that combines satisfiable and unsatisfiable instances. The idea with this experiment is to test how flexible the hyper-heuristics are, given that they are used on a set of more diverse instances. The set of instances used for this experiment is S-Sat/Unsat, which

Table 3: Performance of HH09-12 on S-Hard

HH	h_0	Avg_0	Avg_n	Reduction
HH09	MRV	1560914	1233209	21.0%
HH10	K	1252016	1242421	0.8%
HH11	MFD	1817252	1239895	31.8%
HH12	BZ	2492328	1240081	50.2%

Table 4: Performance of MRV, K, MFD, BZ and HH01-04 on S-Sat/Unsat

HH	Avg. Checks
MRV	146793
K	110262
MFD	160934
BZ	225827
HH01	110868
HH02	121455
HH03	133077
HH04	120881

Table 5: Confidence intervals (with a 5% of confidence) for MRV, K, MFD, BZ and HH01-04 on S-Sat/Unsat

HH	Avg. Checks
MRV:	$65150 < \mu_{MRV} < 228437$
K:	$60283 < \mu_{K} < 160242$
MFD:	$74224 < \mu_{MFD} < 247645$
BZ:	$119606 < \mu_{BZ} < 332047$
HH01:	$61052 < \mu_{HH01} < 160684$
HH02:	$66464 < \mu_{HH02} < 176446$
HH03:	$70220 < \mu_{HH03} < 195934$
HH04:	$65875 < \mu_{HH04} < 175887$

contains both satisfiable and unsatisfiable instances. The results of this experiment are shown in Table 4.

It is interesting to observe that even though MRV achieved the best average performance in S-Sat and S-Unsat, it does not achieve the best result for S-Sat/Unsat. This is actually one of the main drawbacks with heuristics: they may be the best option for a set of instances but we have no guarantee that they will also provide the best results when the instances change. Of course, with hyper-heuristics there is also no guarantee that once a hyper-heuristic is good for a set of instances it will remain working well for a different set. Nevertheless, hyper-heuristics combine the strengths of the single heuristics, and that is why it is likely that when the hyper-heuristics are tested on different sets of instances, they will achieve good results. This idea of generality is supported by the results in Table 4, where the hyper-heuristics HH01-04, which were trained with the set of satisfiable instances S-Sat, achieve results which are better than the ones of MRV, MFD and BZ on S-Sat/Unsat. Furthermore, the difference between K and HH01 is really small (only 606 consistency checks). Of course, these results are based only on the sample means. For this reason we decided to estimate the confidence intervals for the real means (by using a t distribution and a 5% of confidence), in order to confirm that the differences are statistically significant.

We can observe from Table 5 that the intervals of K and HH01 are very similar. The variance among the results of HH01 is slightly larger than the variance obtained with K, and that is the reason why the interval for HH01 is slightly wider than the interval for K. The upper bounds of HH02-04 are below the upper bounds of MRV, MFD and BZ, which confirms that they are good methods to solve the mixed instances in S-Sat/Unsat.

What properties does HH01 have that make it a flexible hyper-heuristic? To answer this, we analysed HH01 with more detail. HH01 contains nine vectors. If we analyse the space $p_1 \times p_2$ according to HH01, the space is divided into regions where the heuristics should be applied. The space and the regions where each heuristic should be applied according to HH01 are presented in Figure 2. As we can observe, the hyper-heuristic represents by itself a rule for the application of the heuristics according to the instance features. According to the results of the single heuristics for S-Sat, MRV achieved the best results (see Table 1 for more information) and then, we can understand why the local search process assigned a large portion of the space to such heuristic (29.5%). But it is something strange that even when K obtained the worst average performance for S-Sat, the local search method assigned a largest proportion of the space to K, 42.5%. The smallest regions correspond to MFD and BZ, with 19.25% and 6%, respectively.

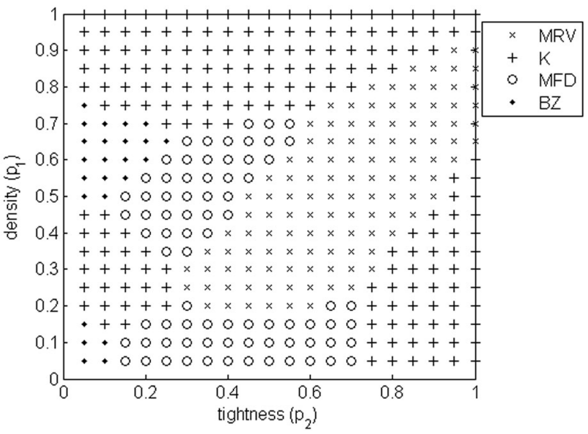

Figure 2: The space $p_1 \times p_2$ classified per regions according to the vectors in HH01

Table 6: Performance of HH013-15 on S-Sat (MRV removed from the set of heuristics)

HH	h_0	Avg_0	Avg_n	Reduction
HH13	K	77209	23430	70.3%
HH14	MFD	26020	20220	10.7%
HH15	BZ	52815	22980	51.4%

4.3 Experiment III

We observed that for the instances in S-Sat and S-Unsat, MRV seems to be the best heuristic. It is then expected that the local search method attaches MRV to a large number of vectors within the hyper-heuristics. What happens if we remove MRV from the list of available heuristics and we run the local search algorithm? Can we expect that a combination of 'not-so-good' heuristics provides results which are comparable to the results of the best heuristic? To answer these questions we ran the local search algorithm three more times, once per heuristic (K, MFD and BZ) and using S-Sat as training set. For each run 100 cycles and a maximum distance between vectors of 0.2 were used.

The results presented in Table 6 indicate that it was not possible to achieve results as good as the ones presented in Section 4.1 when MRV was removed from the set of heuristics that could be attached to the vectors. Nevertheless, we found something interesting: HH14 and HH15 improved their average performance with respect to HH03 and HH04, respectively, when MRV was not included in these hyper-heuristics. This finding is important because suggests that the way the local search process explores the heuristic space may exclude some important points in the space given the set of heuristics, even though a good set of heuristics is provided.

5. CONCLUSIONS AND FUTURE WORK

We have described a methodology based on a local search strategy to produce hyper-heuristics for variable ordering within CSP. The hyper-heuristics produced are very competitive, being able to improve the average performance of the heuristics, specially on satisfiable instances. We found that the set of heuristics to be used to produce a hyper-heuristic is a critical element to consider and it is not enough to provide good heuristics but to also provide diversity to achieve better results.

When tested on mixed instances, the hyper-heuristics showed their real contribution. The four hyper-heuristics produced for a set of only satisfiable instances achieved really good results on a set of unseen satisfiable and unsatisfiable instances. These hyper-heuristics, which were not trained for unsatisfiable instances, performed better than three of the four heuristics. K, which resulted to be the best heuristic for the set of mixed instances, has a behaviour which is statistically similar to the performance of HH01 (which used MRV as default heuristic). Then, results confirm our initial idea that once a hyper-heuristic has been trained (even on an specific set of instances), the properties and strengths of the different heuristics involved help the hyper-heuristic to obtain an acceptable performance even when the instances change. Regarding the CSP instances used in this investigation, we are aware that it is not enough to test our approach only on random instances. We want to test our hyper-heuristics on structured instances taken from existing CSP repositories.

Even though the improvements achieved by the hyper-heuristics produced are small for some sets, it must not be interpreted as a failure in the model. The hyper-heuristics reduce the average effort on the sets where they were trained for and also behave well and produce acceptable results when tested on unseen instances with different properties to the ones used for training. We suspect that the use of very homogeneous instances does not provide enough structure for the hyper-heuristics to be able to grip on, then a simple but good heuristic specialized for those instances is very difficult to overcome. Then, it seems natural that in future developments we include more structured instances or at least, instances from different random CSP generators to produce such heterogeneity in the CSP instances.

As future work we have considered including new variable ordering heuristics and also incorporating a decision process for value ordering, in such a way that the hyper-heuristic also can decide among a set of value ordering heuristics to complement the search. Additionally, we want to explore more alternatives to test the performance of the hyper-heuristics, not only the average consistency checks on the sets.

Finally, we are considering the effect of the frequency the hyper-heuristic decides which heuristic to apply. The current implementation invokes the selection of heuristics every time a variable is to be instantiated. We want to consider the effect of postponing the selection of the next ordering heuristic by one or two variables, instead of doing the selection at each node of the search. With this, the hyper-heuristic will be invoked fewer times, and once a variable ordering heuristic has been selected for being used, it would be active for various nodes until a new selection process takes place.

6. ACKNOWLEDGMENTS

This research was supported in part by ITESM under the Research Chair CAT-144 and the CONACYT Project under grant 99695.

7. REFERENCES

[1] 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*, 6(4):329–344, 2001.

[2] J. Allen and S. Minton. Selecting the right heuristic algorithm: Runtime performance predictors. In G. McCalla, editor, *Advances in Artificial Intelligence*, volume 1081 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 1996.

[3] 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*, pages 61–75, 1996.

[4] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[5] S. A. Bittle and M. S. Fox. Learning and using hyper-heuristics for variable and value ordering in constraint satisfaction problems. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2209–2212. ACM, 2009.

[6] D. Brelaz. New methods to colour the vertices of a graph. *Communications of the ACM*, 22, 1979.

[7] E. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. Technical Report NOTTCS-TR-SUB-0907061259-5808, School of Computer Science, University of Nottingham, 2009.

[8] E. Burke, G. Kendall, R. O'Brien, D. Redrup, and E. Soubeiga. An ant algorithm hyper-heuristic. In *Proceedings of the fifth Metaheuristics International Conference (MIC'03)*, volume 10, pages 1–10, 2003.

[9] K. Chakhlevitch and P. Cowling. Hyperheuristics: Recent developments. In C. Cotta, M. Sevaux, and K. Sörensen, editors, *Adaptive and Multilevel Metaheuristics*, volume 136 of *Studies in Computational Intelligence*, pages 3–29. Springer, 2008.

[10] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of International Joint Conferences on Artificial Intelligence (IJCAI'91)*, pages 331–337, 1991.

[11] 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), volume 1, pages 116–125. ACM Press, 1988.

[12] P. Cowling, G. Kendall, and E. Soubeiga. Hyperheuristics: A robust optimisation method applied to nurse scheduling. In *Seventh International Conference on Parallel Problem Solving from Nature (PPSN'02)*, Lecture Notes in Computer Science, pages 851–860. Springer, 2002.

[13] A. H. C. N. Eoin O'Mahony, Emmanuel Hebrard and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. 2008.

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

[15] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

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

[17] 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)*, pages 179–193, 1996.

[18] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[19] T. Hogg. Refining the phase transitions in combinatorial search. *Artificial Intelligence*, 81:127–154, 1996.

[20] 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*, pages 169–174. AAAI Press / The MIT Press, 2000.

[21] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[22] S. Minton, M. D. Johnston, A. Phillips, and P. Laird. Minimizing conflicts: A heuristic repair method for CSP and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[23] J. C. Ortiz-Bayliss, E. Özcan, A. J. Parkes, and H. Terashima-Marín. Mapping the performance of heuristics for constraint satisfaction. In *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC'10)*, pages 1–8. IEEE Press, 2010.

[24] J. C. Ortiz-Bayliss, H. Terashima-Marín, P. Ross, and S. E. Conant-Pablos. Evolution of neural networks topologies and learning parameters to produce hyper-heuristics for constraint satisfaction problems. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation (GECCO'11)*, pages 261–262. ACM, 2011.

[25] S. Petrovic and S. L. Epstein. Random subsets support learning a mixture of heuristics. *International Journal on Artificial Intelligence Tools*, pages 501–520, 2008.

[26] P. Prosser. Binary constraint satisfaction problems: Some are harder than others. In *Proceedings of the European Conference in Artificial Intelligence*, pages 95–99, 1994.

[27] P. W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[28] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach.* Prentice Hall, 1995.

[29] J. D. Schaffer and L. J. Eshelman. Combinatorial optimisation by genetic algorithms: The value of the genotype/phenotype distinction. In *First International Conference on Evolutionary Computation and its applications (EvCa'96)*, pages 110–120. Springer, 1996.

[30] B. M. Smith. A tutorial on constraint programming. Technical report, University of Leeds, 1995.

[31] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155–181, 1996.

[32] B. M. Smith and S. A. Grant. Trying harder to fail first. In *Thirteenth European Conference on Artificial Intelligence (ECAI'97)*, pages 249–253. John Wiley & Sons, 1997.

[33] H. S. Stone and J. M. Stone. Efficient search techniques: an empirical study of the n-queens problem. *IBM Journal of Research and Development*, 31(4):464–474, 1987.

[34] R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, 38(10):1495–1509, 1992.

[35] H. Terashima-Marín, J. C. Ortiz-Bayliss, P. Ross, and M. Valenzuela-Rendón. Hyper-heuristics for the dynamic variable ordering in constraint satisfaction problems. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO'08)*, pages 571–578. ACM, 2008.

[36] H. Terashima-Marín, P. Ross, and M. Valenzuela-Rendón. Evolution of constraint satisfaction strategies in examination timetabling. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, pages 635–642. Morgan Kaufmann, 1999.

[37] E. Tsang and A. Kwan. Mapping constraint satisfaction problems to algorithms and heuristics. Technical Report CSM-198, Department of Computer Sciences, University of Essex, 1993.

[38] R. Wallace. Analysis of heuristic synergies. In B. Hnich, M. Carlsson, F. Fages, and F. Rossi, editors, *Recent Advances in Constraints*, volume 3978 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2006.