# Using Hyper-heuristics for the Dynamic Variable Ordering in Hard Binary Constraint Satisfaction Problems

Hugo Terashima-Marín[1], José C. Ortiz-Bayliss[1], Peter Ross[2], and Manuel Valenzuela-Rendón[1]

[1] Tecnológico de Monterrey - Center for Intelligent Systems, Monterrey, NL, 64849, Mexico
{terashima, a00796625, valenzuela}@itesm.mx
[2] School of Computing, Napier University, Edinburgh EH10 5DT UK
P.Ross@napier.ac.uk

**Abstract.** The idea behind hyper-heuristics is to discover some combination of straightforward heuristics to solve a wide range of problems. To be worthwhile, such combination should outperform the single heuristics. This paper presents a GA-based method that produces general hyper-heuristics for the dynamic variable ordering within Constraint Satisfaction Problems. The GA uses a variable-length representation, which evolves combinations of condition-action rules producing hyper-heuristics after going through a learning process which includes training and testing phases. Such hyper-heuristics, when tested with a large set of benchmark problems, produce encouraging results for most of the cases. There are instances of CSP that are harder to be solved than others, this due to the constraint and the conflict density [4]. The testebed is composed of hard problems randomly generated by an algorithm proposed by Prosser [18].

## 1 Introduction

A Constraint Satisfaction Problem (CSP) is defined by a set of variables $X_1$, $X_2$, ..., $X_n$ and a set of constraints $C_1$, $C_2$, ..., $C_m$. Each variable $X_i$ has a nonempty domain $D_i$ of possible values. Each constraint $C_i$ involves some subset of variables and specifies the allowable combinations of values for that subset [21]. When trying to find a solution for this kind of problems, it is impossible to avoid the implicit issue of determining which variable is the next to be instantiated. Many researchers have proved, based on analysis and experimentation, the importance of the variable ordering and its impact in the cost of the solution search [18]. This ordering has repercussions in the complexity of the search, which means that finding methods that help to efficiently order these variables is an important issue. For small combinatorial problems, exact methods can be applied. However, when larger and more complex problems appear, exact solutions are not a reasonable choice since the search space grows exponentially, and so does the time for finding the optimal order. Various heuristic

and approximate approaches have been proposed that guarantee finding near optimal solutions. However, it has not been possible to find a reliable method that is able to solve all instances of a given problem. In general, some methods work well for particular instances, but not for all of them.

It is possible to establish different criteria to face the ordering problem, and it can be done either in static or in dynamic fashion. In the static way, the order of the variables is set from the start and is kept during the complete search procedure. This ordering does not guarantee to find a feasible solution given that it is a permutation problem, deriving in an exponential growth in the number of variables. In the other hand, in the dynamic variable ordering, the order is constructed during the search, based on some criterion about the characteristics of the variables left to instantiate. With the dynamic ordering the search space is not as large as in the static ordering. It has been proved in many studies that dynamic ordering gives better results than static ordering [6]. When working with dynamic ordering we can use heuristics to select the next variable to instantiate.

The aim of this paper is to explore a novel alternative on the usage of evolutionary approaches to generate hyper-heuristics for the dynamic variable ordering in CSP. A hyper-heuristic is used to define a high-level heuristic that controls low-level heuristics [3]. The hyper-heuristic should decide when and where to apply each single low-level heuristic, depending on the given problem state. The choice of low-level heuristics may depend on the features of the problem state, such as CPU time, expected number of solutions, values on the objective function, etcetera. Selecting a particular heuristic is dynamic, and depends on both the problem state produced by the previous heuristic applied and the search space to be explored in that point of time.

The investigation in this article, presents a method to generate a general hyper-heuristic intended to provide a way to order variables for a wide variety of hard instances of CSP, so they can be solved. The procedure learns the hyper-heuristic by going through a training phase using hard instances with different features. The generated hyper-heuristic is tested later with a collection of unseen examples providing acceptable results. The general method is based on a variable-length Genetic Algorithm, where the chromosome is conformed of a series of blocks, representing condition-action rules.

The paper is organized as follows. Section 2 presents the variable ordering problem, the solution method proposed and its justification. This is followed by the experimental setup, the results, their analysis and discussion in section 3. Finally, in section 4 we include our conclusions and some ideas for future work.

## 2 Solution Approach

In the literature, one can see that Evolutionary Computation has been used in few CSP investigations [5,15]. Recently, Terashima et al used a combination between low level heuristics and a Genetic Algorithm for dynamic variable ordering in CSP's but did not incorporate the concept of hyper-heuristic [23]. More re-

cently, Terashima et al also used a GA based method to produce hyper-heuristics for the 2-D cutting stock problem with outstanding results [24]. Evolutionary Computation usually includes several types of evolutionary algorithms [25]: Genetic Algorithms [10,14], Evolutionary Strategies [19,22], and Evolutionary Programming [1,7]. In this research we use a GA with variable length chromosomes, a resemblance of what is called a *messy*-GA [11].

### 2.1 The variable ordering problem in CSP

In CSP, the variable ordering is relevant topic due to its impact in the cost of the solution search. As we mentioned before, there are two different ways to set this order: static and dynamic. This research focuses on the dynamic fashion, which uses heuristics to select the next variable. It has been empirically proved in previous studies that dynamic variable ordering is more efficient than the static approach [6]. Also, the literature is rich in heuristics designed for this task: Fail-First [13], Saturation Degree [2], *Rho* [9], *Kappa* [9] and *E(N)* [9], to mention some. However, none of these heuristics has been proved to be efficient in every instance of the problems.

Every time an instantiated variable is checked to verify if it does not violate any constraint is called a constraint check. Counting the constraint checks used to solve a specific instance is used in this work to compare the performance of the different heuristics and the general hyper-heuristic.

Some CSP exhibit a phase transition, where problems change from having a solution to not having any, and this occurs at a critical value of connectivity. Problems that occur below the critical value of connectivity are easy to solve, and problems above the critical value have not a solution because of the high value of connectivity. It is only around the critical value of connectivity that is hard to decide if a problem has a solution [18] and these instances requiere a great computational effort [4]. In this research, we used only hard instances of CSP.

### 2.2 The Set of Heuristics Used

In CSP, the related heuristics refer to the way the next variable is selected and which value has to be used to instantiate that variable. In this investigation we focus on the first kind of heuristics, those related to the order in which variables are selected to be instantiated. Value ordering is beyond the scope of this research and it is left for future work. Some of the heuristics were taken from the literature and others were adapted. We chose the most representative heuristics in its type, considering their individual performance presented in related studies and also in an initial experimentation on a collection of benchmark problems.

The ordering heuristics used in this research are: Fail First (FF), Modified Fail First (MFF), Saturation Degree (Bz), *Rho (R)*, E(N), *Kappa (K)*, and Min-Conflicts (MC). These heuristics, except for MFF and MC are described in detail by Gent et al. [9]. MFF is a modification of the Fail-First heuristic and the MC heuristic is a very simple heuristic based on the idea of selecting a variable that

produces the subproblem that minimizes the number of conflicts in the variables left to instantiate. This heuristic was originally used for value ordering [16, 17], however it was adapted to work for variable ordering.

### 2.3 Combining Heuristics with the proposed GA

The concept of hyper-heuristic is motivated by the objective to provide a more general procedure for optimization [3]. Meta-heuristics methods usually solve problems by operating directly on the problem. Hyper-heuristics deal with the process to choose the right heuristic for solving the problem at hand. The aim is to discover a combination of simple heuristics that can perform well on a whole range of problems. For real applications, exhaustive methods are not a practical approach. The search space might be too large, or the number and types of constraints may generate a complex space of feasible solutions.

It is common to sacrifice quality of solutions by using quick and simple heuristics to solve problems. Many heuristics have been developed for specific problems. But, is there a single heuristic for a problem that solves all instances well? The immediate answer is no. Certain problems may contain features that would make a specific heuristic to work well, but those features may not be suitable for other heuristics. The idea with hyper-heuristics is to combine heuristics in such a way that a heuristic's strengths make up for the drawbacks of another.

The solution model used in this investigation carries features from previous work by Ross et al. [20] and Terashima et al. [24], in which the main focus is to solve one dimensional and two dimensional bin-packing problems, respectively. In the research presented in this article, a GA with variable-length individuals is proposed to find a combination of single heuristics to order variables to solve efficiently a wide variety of instances of CSP.

The basic concept is that, given a problem state $P$, this is associated with the closest point in the chromosome which carries the ordering rules to be applied. This application will transform the problem to a new state $P'$. The purpose is to solve a problem by constructing the answer, deciding on the heuristic to apply at each step.

A chromosome in the messy GA represents a set of labelled points within this simplified problem state space; the label of any point is a heuristic. The chromosome therefore represents a complete recipe for solving a problem, using a simple algorithm: until the problem is solved, (a) determine the current problem state $P$, (b) find the nearest point to it, (c) apply the heuristic attached to the point, and (d) update the state. The GA is looking for the chromosome (representing a hyper-heuristic) which contains the rules that apply best to any intermediate state in the solving process of a given instance.

The instances are divided into three groups: the training and the testing set I and II. The general procedure consists in solving first all instances in the three sets with the single heuristics. This is carried out to keep the best solution that is later used also by the GA we propose. The next step is to let the GA work on the training set until termination criterion is met and a general hyper-heuristic

is produced. All instances in the testing and training sets I and II are then solved with this general hyper-heuristic.

**Representation.** Each chromosome is composed of a series of *blocks*. Each block $j$ includes nine numbers. The first eight represent an instance of the problem state. The first number indicates the percentage of variables that remain to be instantiated ($v_j$). The second number represents the percentage of values in all the domains left ($d_j$). The next four numbers are related to the constraints that still remain to be instantiated. For this, we defined a three category classification for constraints: large constraints are those that prohibit a fraction $p_c \geq 0.55$, medium constraints have a $0.45 \leq p_c < 0.55$ and small constraints have a $p_c < 0.45$. These values were obtained through previous experimentation. With these categories we defined that the third number represents the percentage of all constraints($c_j$), the fourth number indicates the percentaje of large constraints ($lc_j$), the fifth one shows the percentage of medium constraints ($mc_j$), and the sixth number represents the percentage of small constraints ($sc_j$). The next two numbers represent the value of the *rho* ($r_j$) and *kappa* ($k_j$) factors, respectively. The last number represents the variable ordering heuristic associated to the instance of the problem state. For a given problem state, the inital eight numbers would lie in a range between 0 and 1, so that the actual problem state is a point inside the unit eight-dimensional space. Nevertheless, we allow the points defined in each block to lie outside the unit cube, so we redefined the range to be from $-3$ to 3. At each step, the algorithm applies the heuristic that is associated to the block that is closest to actual problem state. We measure the distance $d$ between the problem state $P'$ and the instance inside each block $j$ using the euclidean distance formula.

**The Fitness Function.** The most common criterion to measure the efficiency of a search algorithm used to determine if a CSP instance has or does not have a solution is the count of the consistency checks made during the search. To calculate the quality of an individual there are two steps that must be done:

1. Every CSP instance is solved using each low level heuristic. The CSP instance solver combines backtracking and forward checking to solve the instances. The best heuristic result, for each specified instance $i$ is stored (let us call it $BH_i$). These results are prepared in advance of running the GA.
2. Each one of the problems assigned to the individual is solved using the coded hyper-heuristic. The best result of the low level heuristics ($BH_i$) is divided by the number of consistency checks used during the search ($HH_i$). Combining all together, the fitness function ($FF_i$) is computed as: $FF = \frac{BH_i}{HH_i}$. This evaluation guarantees that, when the general hyper-heuristic produces better results, the fitness function will return a number greater than one.

To compute the fitness for each chromosome, the distance between the solution obtained by that individual with respect to the best result given by the

single heuristic ($BH_i$) is measured. The fitness is an average, and it is given by: $FF_m^l = \frac{FF_m^{l-1} \cdot mp_m + \sum_{i=1}^{5} FF(m_i)}{mp+5}$, where $FF_m^{l-1}$ is the fitness for individual $m$ in the previous generation; $mp_m$ is the number of problems individual $m$ has seen so far; $FF(m_i)$ is the fitness obtained by individual $m$ for the each problem assigned to the hyper-heuristic. After each generation $l$, a new problem is assigned to each individual $m$ in the population and its fitness is recomputed again by: $FF_m^l = \frac{FF_m^{l-1} \cdot mp_m + FF(m)}{mp+1}$.

In both cases the evaluation is the result of both the past fitness and the current fitness. A very good performance for certain instances is not very significant if for most of the instances an individual has a low performance. In the other hand, if a hyper-heuristic behaves really good for most of the problems, a bad result for a single case does not affect negatively too much.

## 3 Experiments and Results

This section presents the experiments carried out during the investigation and the results obtained. The benchmark set is composed of randomly generated hard instances using the algorithm proposed by Prosser [18]. This algorithm generates instances of CSP characterized by the 4-tuple $< n, m, p_1, p_2 >$, where $n$ is the number of variables, $m$ is the uniform domain size, $p_1$ is the probability that a constraint exists between a pair of variables, and $p_2$ as the tightness of constraints [18]. The collection includes 1250 different instances: 600 for the training set, 400 for the testing set I and 250 for the training set II. Both the training and the testing set I are composed by distinct instances with $n = 20$ and $m = \{10, 20\}$ (A and B instances). The training set II contains instances with $n = 30$ and $m = 10$ (C instances). A more detailed description of the instances and the number of instances generated for every set is presented in Table 1.

**Table 1.** Description of the hard instances.

| Class | $n$ | $m$ | $p_1$ | $p_2$ | Training set | Testing set I | Training set II |
|---|---|---|---|---|---|---|---|
| A1 | 20 | 10 | 0.20 | 0.65 | 60 | 40 | 0 |
| A2 | 20 | 10 | 0.40 | 0.45 | 60 | 40 | 0 |
| A3 | 20 | 10 | 0.60 | 0.33 | 60 | 40 | 0 |
| A4 | 20 | 10 | 0.80 | 0.27 | 60 | 40 | 0 |
| A5 | 20 | 10 | 1.00 | 0.22 | 60 | 40 | 0 |
| B1 | 20 | 20 | 0.10 | 0.90 | 60 | 40 | 0 |
| B2 | 20 | 20 | 0.20 | 0.75 | 60 | 40 | 0 |
| B3 | 20 | 20 | 0.30 | 0.63 | 60 | 40 | 0 |
| B4 | 20 | 20 | 0.40 | 0.55 | 60 | 40 | 0 |
| B5 | 20 | 20 | 0.50 | 0.48 | 60 | 40 | 0 |
| C1 | 30 | 10 | 0.10 | 0.75 | 0 | 0 | 50 |
| C2 | 30 | 10 | 0.20 | 0.53 | 0 | 0 | 50 |
| C3 | 30 | 10 | 0.30 | 0.40 | 0 | 0 | 50 |
| C4 | 30 | 10 | 0.40 | 0.30 | 0 | 0 | 50 |
| C5 | 30 | 10 | 0.50 | 0.27 | 0 | 0 | 50 |

The training set was used to generate the general hyper-heuristic (HH) shown in Table 2, which represents the best individual in the last generation of the training process from the best result of ten runs. For each run, the parameters used to generate the general HH were: a population size of 20 during 100 cycles,

crossover probability of 1.0, and mutation probability of 0.1. The general HH obtained includes 4 rules indicating the different problem states and the associated heuristic to be applied.

**Table 2.** General hyper-heuristic produced by the GA.

| $v$ | $d$ | $c$ | $lc$ | $mc$ | $sc$ | $r$ | $k$ | $h$ |
|---|---|---|---|---|---|---|---|---|
| -0.075 | 0.643 | -1.045 | 0.513 | 0.536 | 0.8169 | -0.194 | -0.1739 | MC |
| -0.238 | 1.117 | -0.02 | -1.1259 | 0.24 | -1.085 | 0.516 | -0.646 | *Rho* |
| 0.083 | -0.871 | 1.041 | -0.025 | -2.0139 | 0.713 | -0.197 | -2.257 | *Kappa* |
| 0.24 | 0.662 | -0.493 | 1.089 | 0.138 | -1.165 | 0.786 | -0.083 | $E(N)$ |

Aiming at testing the model effectiveness, four experiments were carried out.

### 3.1 Experiment Type I

In this experiment we test the general hyper-heuristic (HH) with instances already seen. We solved the instances in the training set using the general HH from Table 2. Results are compared against those generated by the best result from the low-level heuristics and the average of constraint checks done by the low-level heuristics. Table 3 sumarizes the results from this experiment and clasifies them into three main categories: (1) the general HH uses less consistency checks than the best heuristic, (2) the general HH uses the same amount of consistency checks (a difference of $\pm 3\%$ is considered as if both approaches use the same amount of consistency checks) and (3) the general HH uses more consistency checks that the best result from the low-level heuristics.

**Table 3.** Results of experiment type I.

| | HH vs Best result of the low-level heuristics | | | | | HH vs average result of the low-level heuristics | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Reduction | | Equal | Increment | | Reduction | | Equal | Increment | |
| Class | > 15% | 15% to 3% | ± 3% | 3% to 15% | > 15% | > 15% | 15% to 3% | ± 3% | 3% to 15% | > 15% |
| A1 | 0.00 | 0.00 | 41.67 | 13.33 | 45 | 98.33 | 1.67 | 0.00 | 0.00 | 0.00 |
| A2 | 0.00 | 0.00 | 56.67 | 16.67 | 26.67 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| A3 | 0.00 | 0.00 | 88.33 | 5.0 | 6.67 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| A4 | 0.00 | 5.00 | 65.00 | 26.67 | 3.33 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| A5 | 0.00 | 8.33 | 18.33 | 61.67 | 11.67 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B1 | 0.00 | 0.00 | 55.00 | 8.33 | 36.67 | 95.00 | 5.00 | 0.00 | 0.00 | 0.00 |
| B2 | 0.00 | 0.00 | 56.67 | 10.00 | 33.33 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B3 | 0.00 | 0.00 | 65.00 | 10.00 | 25.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B4 | 0.00 | 0.00 | 95.00 | 3.33 | 1.67 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B5 | 0.00 | 0.00 | 100.00 | 0.00 | 0.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |

As we can observe, the results show that the HH presents a competitive behavior for most of the classes. Some instances in classes A4 and A5 were solved using less consistency checks than the best result of the low-level heuristics, a fact that was not present in other classes. The best results of the HH against the best result from the low-level heuristics were obtained for $m = 20$, $n = 10$, $p_1 = 0.8$ and $p_2 = 0.27$, parameters that correspond to class A4. The results of the general HH versus the average constraint checks of the low-level heuristics

suggest that the HH is a good choice to solve different instances of CSP. The general HH is almost always much better than the average, representing a reduction greater than 15% in almost every instance. The outstanding results obtained when comparing the HH against the average of constraint checks is due to the bad behavior of some heuristics for certain of these instances. As we mentioned before, for some instances a low-level heuristic has a very good performance, but for other it is simply very poor. These bad results raise the average and it makes possible that the HH has a better perfomance.

### 3.2 Experiment Type II

The second experiment uses the same general HH used in the experiment type I. This time, the unseen instances of the testing set I were solved using the HH, and the results were compared against the results of the best low-level heuristic and the average of constraints checks used by the single heuristics for each instance. The instances used in this experiment were not used during the training phase, but they were generated with similar parameters. The results of this experiment are shown in Table 4.

**Table 4.** Results of experiment type II.

| | HH vs Best result of the low-level heuristics | | | | | HH vs average result of the low-level heuristics | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Reduction | | Equal | Increment | | Reduction | | Equal | Increment | |
| Class | > 15% | 15% to 3% | ± 3% | 3% to 15% | > 15% | > 15% | 15% to 3% | ± 3% | 3% to 15% | > 15% |
| A1 | 0.00 | 0.00 | 45.00 | 12.50 | 42.50 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| A2 | 35.00 | 7.50 | 2.50 | 2.50 | 52.50 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| A3 | 0.00 | 2.50 | 72.50 | 15.00 | 10.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| A4 | 0.00 | 10.00 | 65.00 | 17.50 | 7.50 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| A5 | 0.00 | 12.50 | 27.50 | 35.00 | 25.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B1 | 0.00 | 0.00 | 37.50 | 17.50 | 45.00 | 95.00 | 2.50 | 2.50 | 0.00 | 0.00 |
| B2 | 0.00 | 0.00 | 65.00 | 12.50 | 22.50 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B3 | 0.00 | 0.00 | 87.50 | 2.50 | 10.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B4 | 0.00 | 0.00 | 87.50 | 10.00 | 2.50 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| B5 | 0.00 | 0.00 | 100.00 | 0.00 | 0.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |

In this experiment, the results for class A2 are outstanding. In 42.5% of the instances the HH overcomes the best result of the low-level heuristics. This result is interesting, because the same parameters were used to generate the training set and the results for class A2 were not as good as in that case. It is natural to think that the results for a specific class in the training set should be similar to those in the testing set I for that class too, because they were both generated with the same parameters. It seems that the random numbers generator had a great impact in the generation for some classes of instances, in this case, for class A2. For classes A3 to A5 the results are similar to those presented in the training set: the HH achieves a reduction in the number of constraint checks used by the best result of the low-level heuristics in some cases. If we compare the results of the HH against the average of constraints checks of the low-level heuristics we observe again that the HH is much better than the average. For every class of instances, the HH uses a very reduced number of constraints checks compared to the used by the average of the low-level heurisics.

### 3.3 Experiment Type III

For experiment type III we used the general HH presented in the last two experiments. In this experiment, the instances of the testing set II were solved using the HH, and the results were compared as before. The instances used in this experiment were not used during the training phase, and they were not generated with similar parameters. The results of this experiment are shown in Table 5.

**Table 5.** Results of experiment type III.

| | HH vs Best result of the low-level heuristics | | | | | HH vs average result of the low-level heuristics | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Reduction | | Equal | Increment | | Reduction | | Equal | Increment | |
| Class | > 15% | 15% to 3% | ± 3% | 3% to 15% | > 15% | > 15% | 15% to 3% | ± 3% | 3% to 15% | > 15% |
| C1 | 0.00 | 0.00 | 40.00 | 12.00 | 48.00 | 96.00 | 2.00 | 0.00 | 2.00 | 0.00 |
| C2 | 0.00 | 0.00 | 68.00 | 10.00 | 22.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| C3 | 0.00 | 0.00 | 94.00 | 2.00 | 4.00 | 100.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| C4 | 0.00 | 0.00 | 60.00 | 2.00 | 38.00 | 84.00 | 10.00 | 6.00 | 0.00 | 0.00 |
| C5 | 0.00 | 10.00 | 74.00 | 12.00 | 4.00 | 42.00 | 10.00 | 46.00 | 2.00 | 0.00 |

The instances used for this experiment were never seen by the HH and had different parameters to those used in training. In other words, the HH was not trained to solve these classes of instances. The results show that the HH is competitive, but it is unable to reduce the best result of the low-level heuristic for classes C1 to C4. For class C5 a reduction is present in 10% of the instances, but it is not a reduction greater that 15% in the number of constraints checks. When comparing the HH against the average of constraint checks used by the low-level heuristics, the results are as good as in the past three experiments. For class C5, the general HH was only able to achieve a reduction greater than 15% in 42% of the instances. The results are not as good as in the past two experiments, but they are still very promissing.

### 3.4 Experiment Type IV

For this experiment we compared the general Hyper-heuristic with each of the low-level heuristics when solving the testing set I. The performance of the low-level heuristics and the HH is shown in Table 6.

**Table 6.** Low-level heuristics and general hyper-heuristic compared against the best result of the low-level heuristics for the testing set I.

| | Reduction | | Equal | Increment | |
|---|---|---|---|---|---|
| Method | > 15% | 15% to 3% | ± 3% | 3% to 15% | > 15% |
| *Kappa* | 0.00 | 0.00 | 73.00 | 10.00 | 17.00 |
| HH | 0.00 | 2.67 | 54.00 | 24.67 | 18.67 |
| *E(N)* | 0.00 | 0.00 | 31.00 | 28.67 | 40.33 |
| FF | 0.00 | 0.00 | 6.67 | 27.00 | 66.33 |
| *Rho* | 0.00 | 0.00 | 5.67 | 23.67 | 70.66 |
| MC | 0.00 | 0.00 | 2.67 | 0.33 | 97.00 |
| MFF | 0.00 | 0.00 | 1.33 | 0.67 | 98.00 |
| Bz | 0.00 | 0.00 | 0.00 | 18.33 | 81.67 |

As we can observe, the HH is competitive and is only overcome by the *kappa* heuristic, which shows the best results in 73% of the instances. The HH achieves a reduction in the minimum number of constraint checks required by the low-level heuristic for 2.67% of the instances, and uses the minimum number of constraint checks achieved by the heuristics for at least half of the instances.

## 4 Conclusions and Future Work

This document has described experimental results in a model based on a variable-length GA which evolves combinations of condition-action rules representing problem states and associated selection heuristics for the dinamic variable ordering problem in CSP. These combinations are called hyper-heuristics. Overall, the scheme identifies efficiently general hyper-heuristics after going through a learning procedure with training and testing phases. When applied to unseen examples, those hyper-heuristics solve many of the problems efficiently, in some cases better than the best single heuristic for each instance. Ideas for future work involve extending the proposed strategy to solve problems including heuristics for value ordering and include other algorithms for the CSP solving module.

## 5 Acknowledgments

This research was supported in part by ITESM under the Research Chair CAT-010 and the CONACYT Project under grant 41515.

## References

1. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic programming: An Introduction*. Morgan Kaufmann Publishers, Inc, London, 1998.
2. D. Brelaz. New methods to colour the vertices of a graph. *Comunications of the ACM*, 22, 1979.
3. E. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern research technolology. In *Handbook of Metaheuristics*, pages 457–474. Kluwer Academic Publishers, 2003.
4. P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the real hard problems are. In *Proceedings of IJCAI-91*, pages 331–337, 1991.
5. B. G. W. Craenen, A. E. Eiben and J. I. van Hemert. Comparing evolutionary algorithms on binary constraint satisfaction problems. *Evolutionary Computation, IEEE Transactions on*, 7(5):424–444, 2003.
6. R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 38(2):211–242, 1994.
7. D. B. Fogel, L. A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, New York, 1966.
8. M. Garey and D. Johnson. *Computers and Intractability*. W.H. Freeman and Company, New York, 1979.

9. I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T.Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP-96*, pages 179–193, 1996.
10. D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Adison Wesley, 1989.
11. D. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis and first results. *Complex Systems*, pages pp. 93–130, 1989.
12. B. L. Golden. Approaches to the cutting stock problem. *AIIE Transactions*, 8:256–274, 1976.
13. R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
14. J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.
15. E. Marchiori and A. Steenbeek. A genetic local search algorithm for random binary constraint satisfaction problems. *Proceedings of the 2000 ACM symposium on Applied computing.* Volume 1, pages 458–462. Como, Italy. 2000.
16. S. Minton, M. D. Johnston, A. Phillips, and P. Laird. Minimizing conflicts: A heuristic repair method for csp and scheduling problems. *Artificial Intellgence*, 58:161–205, 1992.
17. 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*, pages 17–24, 1990.
18. P. Prosser. Binary constraint satisfaction problems: Some are harder than others. In *Proceedings of the European Conference in Artificial Intelligence*, pages 95–99, Amsterdam, Holland, 1994.
19. I. Rechenberg. *Evolutionstrategie: Optimierung technischer systeme nach prinzipien dier biolischen evolution.* Frommann-Holzboog, Stuttgart, 1973.
20. P. Ross, J. M. Blázquez, S. Schulenburg, and E. Hart. Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyper-heuristics. *Proceedings of GECCO 2003*, pages 1295–1306, 2003.
21. S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach.* Prentice Hall, 1995.
22. H. P. Schwefel. *Numerical Optimization of Computer Models.* Wiley, Chinchester, UK, 1981.
23. H. Terashima-Marín, R. Calleja-Manzanedo and M. Valenzuela-Rendón. Genetic Algorithms for Dynamic Variable Ordering in Constraint Satisfaction Problems. Advances in Artificial Intelligence Theory, 16: pages 35–44, 2005.
24. H. Terashima-Marín, C. J. Farías-Zárate, P. Ross and M. Valenzuela-Rendón. A GA-Based Method to Produce Generalized Hyper-heuristics for the 2D-Regular Cutting Stock Problem. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 591–598. Seattle, Washington, USA, 2006
25. R. A. Wilson and F. C. Keil. *The MIT Encyclopedia of the Cognitive Science.* MIT Press, Cambridge, Massachussets, 1999.