Hyper-heuristics for the Dynamic Variable Ordering in Constraint Satisfaction Problems

H. Terashima-Marín, J. C. Ortiz-Bayliss
ITESM-Intelligent Systems
Av. E. Garza Sada 2501
Monterrey, NL, 64849 Mexico
{terashima, a00796625}@itesm.mx

P. Ross
School of Computing
Napier University
Edinburgh EH10 5DT UK
P.Ross@napier.ac.uk

M. Valenzuela-Rendón
ITESM-Intelligent Systems
Av. E. Garza Sada 2501
Monterrey, NL, 64849 Mexico
valenzuela@itesm.mx

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. The testebed is composed of problems randomly generated using an algorithm proposed by Prosser [17].

Categories and Subject Descriptors

I.2 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods and Search*

General Terms

Algorithms

Keywords

Evolutionary Computation, Hyper-heuristics, Optimization, Constraint Satisfaction Problems, Dynamic Variable Ordering

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 [20]. 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 [17]. 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 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 because 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 [5]. 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 instances of CSP, so they can be solved. The procedure learns the hyper-heuristic by going through a training phase using instances with a variety of 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 [4,14]. Recently, Terashima et al used a combination between low level heuristics and a Genetic Algorithm for dynnamic variable ordering in CSP's but did not incorporate the concept of hyper-heuristic [22]. More recently, Terashima et al also used a GA based method to produce hyper-heuristics for the 2-D cutting stock problem with outstanding results [23].

Evolutionary Computation usually includes several types of evolutionary algorithms [24]: Genetic Algorithms [9,13], Evolutionary Strategies [18,21], and Evolutionary Programming [1,6]. In this research we use a GA with variable length chromosomes, a resemblance of what is called a *messy*-GA [10].

2.1 The variable ordering problem in CSP's

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 [5]. Also, the literature is rich in heuristics designed for this task: Fail-First [12], Saturation Degree [2], Rho [8], Kappa [8] and E(N) [8], 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.

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).- Also called Minimum Remaining Values (MRV), it selects the variable with the minimun number of available values in its domain. This heuristic decreases the branching factor associated to the search by selecting the variables that faster take to dead ends [12].

- Modified Fail First (mFF).- It selects the variable involved in the maximum number of constraints. It is a modification of the Fail-First heuristic and they both provide different results when used in the instances.

- Saturation Degree (Bz).- This heuristic has been used mostly for graph coloring, but can be easily adapted to be useful in this research. It selects the variable that is involved in the maximum number of constraints from variables already instanciated [2].

- *Rho*.- This heuristic was first used by Ian P. Gent [8] and it is based on the estimation of the solution density p, given by:

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

  where a constraint c prohibits in average a fraction p_c of possible assignations.

  It selects the heuristic that maximizes the solution density of the resulting subproblem, that is, the variable with the most conflictive constraints. The resulting subproblem contains the larger amount of states which are solutions to the problem.

- E(N).- This heuristic selects a variable such that the subproblem maximizes the expected number of solutions. This number is calculated as:

  $$E(N) = \prod_{v \in V} (d_v) \times \prod_{c \in C} (1 - p_c) \tag{2}$$

  where d_v represents the uniform domain size of variable v.

  This heuristic also maximizes the solution density of the new subproblem. The selection criterion of the E(N) heuristic is a combination of the Fail-First and Rho heuristics [8].

- *Kappa*.- It selects the next variable such that the new subproblem minimizes the *kappa* factor, wich is calculated as follows:

  $$k = \frac{-\sum_{c \in C} log_2(1 - p_c)}{\sum_{v \in V} log_2(d_v)} \tag{3}$$

  The *kappa* factor indicates how restricted an instance is. It is likely that instances with $k \ll 1$ are less restricted and that others with con $k \gg 1$ probably are highly restricted, and maybe they have no solution [8].

- Min-conflicts.- This is a very simple heuristic and it is 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 [15, 16], however it was adapted to work for variable ordering.

Some of these heuristics are described also by Gent et al [8].

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 heuristicto 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 [19] and Terashima et al [23], 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. The current state P of the problem is a much-simplified representation of the actual state, and is described in more detail in section 3.2.1.

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 heursitic 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 two groups: the training and the testing set. The general procedure consists in solving first all instances in both 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 both the testing and training sets are then solved with this general hyper-heuristic.

The complete process is presented in Figure 1.

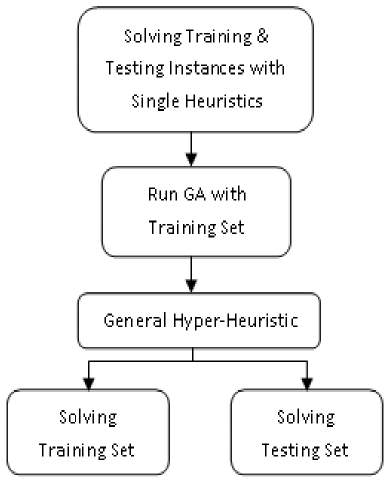

Figure 1: Solution Model.

2.3.1 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.

2.3.2 Genetic Operators

We dealt in this investigation with two crossover and three mutation operators. The first crossover operator is very similar to the normal two-point crossover. Since the number of blocks in each chromosome is variable, each parent selects the first and last block independently. However the points selected inside each corresponding block are the same for both parents. The blocks and poinst are chosen using a uniform distribution. The other crossover operator works at block level, and it is very similar to the normal one-point crossover. This operator exchanges 10% of blocks between parents, meaning that the first child obtains 90% of information form the first parent, and 10% from the second one.

The first mutation operator randomly generates a new block and adds it at the end of the chromosome; the second operator randomly selects and eliminates a block within the chromosome; and the last one randomly selects a block in the chromosome and a position inside that block to replace it with a new number between -3 and 3, generated with a normal distribution with mean 0.5 and truncated accordingly.

2.3.3 The Fitness Function

The most common criterion to mesure 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:

$$FF = \frac{BH_i}{HH_i} \tag{4}$$

This evaluation guarantees that, when the general hyper-heuristic produces better results, the fitness function will return a number greater than one.

The GA cycle consists of the following steps:

1. Generate initial population.

2. Assign five problems to each chromosome and get its fitness.

3. Apply selection (tournament), crossover and mutation operators to produce two children.

4. Assign five problems to each new child and get its fitness.

5. Replace the two worst individuals with the new offspring.

6. Assign a new problem to every individual in the new population and recompute fitness.

7. Repeat from step three until a termination criterion is reached.

To compute the fitness for each chromosome (at steps two and four of the above cycle), 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 a weighted average and it is given by:

$$FF_m^l = \frac{FF_m^{l-1} \cdot mp_m + \sum_{i=1}^{5} FF(m_i)}{mp + 5} \tag{5}$$

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 and computed with equation 4. After each generation l, a new problem is assigned to each individual m in the population and its fitness is recomputed again by a weighted average as follows:

$$FF_m^l = \frac{FF_m^{l-1} \cdot mp_m + FF(m)}{mp + 1} \tag{6}$$

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 significative if for most of the instances an individual has a low performance. In the other hand, if a hyper-heuristics behaves really good for most of the problems, a bad result for a single case does not affect negatively so much.

2.3.4 GA Parameter Set

After previous experimentation, the parameters for the GA used in this investigation were set as follows: population size, 20; number of generations, 100; crossover probability, 1.0; and mutation probability, 0.1.

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 instances using the algorithm proposed by Prosser [17]. This algorithm generates CSP's 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 [17]. In this research, two criteria are used to modify the CSP instances used in the experiments. The first criterion is the size of the problem, which can be modified through the change of the number of variables (n) and the size of the uniform domain size (m). The second criterion is the tightness of the constraints, which is related to the value of p_2. The collection includes 460 different instances: 230 for training and the same number for testing. Both the training and testing set are divided into two groups each one:

- Group I. Instances with constant number of variables and variable uniform domain size. These instances are divided into nine subgroups of 20 instances each: A1 ($n = 10$, $m = 10$), A2 ($n = 10$, $m = 15$), A3 ($n = 10$, $m = 20$), B1 ($n = 15$, $m = 10$), B2 ($n = 15$, $m = 15$), B3 ($n = 15$, $m = 20$), C1 ($n = 20$, $m = 10$), C2

(n = 20, m = 15) and C3 (n = 20, m = 20). There is a different group I for training and for testing.

- Group II. Instances with constant size and variable probability of conflict generation. These instances contain 10 variables with a uniform domain size of 10 and p_1 of 0.5. They are divided into five subgroups of 10 instances each: D1 (p_2 = 0.2), D2 (p_2 = 0.4), D3 (p_2 = 0.6), D4 (p_2 = 0.8) and D5 (p_2 = 1.0).

The training set was used to generate the general hyper-heuristic shown in Table 1, which represents the best individual in the last generation of the training process from our best run. It includes 9 rules indicating the different problem states and the associated heuristic to be applied. In the resulting hyper-heuristic, we can observe that despite the same single heuristic appears more than once in the action part, the conditions for applying it are quite different.

Table 1: General Hyper-heuristic produced by the GA.

v	d	c	lc	mc	sc	r	k	h
0.27	1.71	0.83	0.46	-0.47	-0.95	0.10	-0.08	FF
-0.74	-1.91	-0.55	0.37	0.82	0.24	1.64	-0.51	Rho
0.71	0.68	-0.99	-0.24	-0.21	0.00	-0.17	0.90	E(N)
0.42	-1.80	0.55	-0.61	-0.80	-0.64	-1.44	-0.67	FF
0.10	1.49	-0.41	-1.24	-0.23	0.38	-1.32	-1.13	E(N)
-0.15	0.03	0.97	-0.96	-0.57	-0.39	0.39	-1.25	Bz
-0.60	-1.56	-0.30	0.50	-0.55	0.52	0.52	-0.01	Bz
1.25	-0.47	-0.32	0.67	1.22	-0.43	-0.44	0.63	Bz
0.08	0.85	-1.41	0.38	-1.04	-0.15	-0.74	0.46	FF'

Aiming at testing the model effectiveness, three kinds of experiments were carried out.

3.1 Experiment Type I

In this experiment we test the general hyper-heuristic with a specific subset of instances from both the training and the testing set. The number of variables (n) was set to the values 10, 15 and 20. All the other parameters were constant and set as follows: m = 10, p_1 =0.5 and p_2 =0.5. This experiment used the instances from group I (A1, B1 and C1).

Next, we solved independently both the instances in the training and testing set using the general hyper-heuristic from Table 1. Results are compared against those generated by the best result from the low level heuristics. Table 2 sumarizes the results from this experiment and clasifies them in three main categories:

1. When the hyper-heuristic makes less consistency checks than the best result obtained from the use of the low level heuristics for certain instance.
2. When the hyper-heuristic makes the same consistency checks than the best result from the the use of the low level heuristics for certain instance. A difference of $\pm 3\%$ is considered as if both approaches use the same amount of consistency checks.
3. When the hyper-heuristic makes the more consistency checks than the best result from the the use of the low level heuristics for certain instance.

It is interesting to see, if we focus in the results obtained by the hyper-heuristic, that 60% of problems in the training

Table 2: Results from experiment type I.

	Reduction		Equal	Increment	
Set	> 15%	15% to 3%	± 3%	3% to 15%	> 15%
Training	0.00	0.00	60.00	16.67	23.33
Testing	0.00	0.00	51.67	8.33	40.00

set were solved with no extra consistency checks or even with a little reduction compared with the best result from the low level heuristics. In 16.67% of problems, the hyper-heuristic obtained solutions with a number of consistency checks between 3 and 15% more than the best heuristic. Finally, in a 23.33% of the instances the hyper-heuristic required more than 15% of consistency checks than the best heuristic. It is important to emphasize that the best single heuristic is not always the same for all instances.

For the testing set, whose instances were not seen before by the hyper-heuristic, results show that the hyper-heuristic has a good performance in 51.67% problems, using the same amount of consistency checks than the best heuristic, more than 3% but lower than 15% in 8.33% of problems and more than 15% in 40% of the instances.

The results obtained from the experiment type I, when using the general hyper-heuristic to solve the training set, are shown in Figure 2. It is also included the average consistency check from all the low level heuristics.

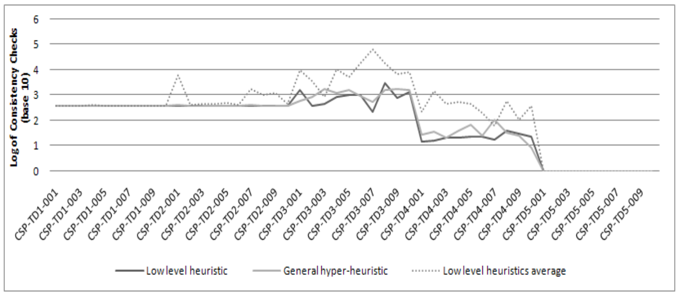

Figure 2: Results of experiment type I with instances from training set.

Figure 3 shows the results from experiment type I for the testing set.

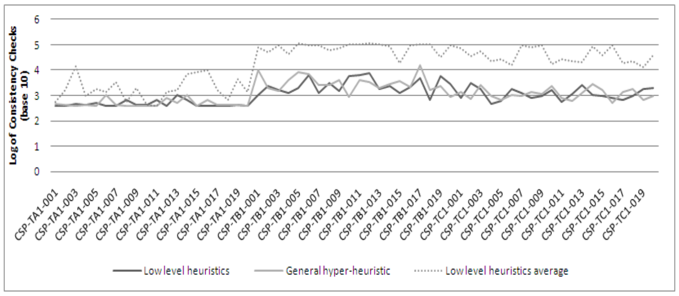

Figure 3: Results of experiment type I with instances from testing set.

We can observe in both figures that, in general, the hyper-heuristic behaves well when is compared against the best result from the low level heuristics. In some cases, the general hyper-heuristic reduces the number of consistency checks needed to solve the problem using the best heuristic.

We present the number of consistency checks expressed as a logarithm, as presented by Prosser [17]. If we compare the results with the average of the low level heuristics, we can clearly identify that the hyper-heuristic is better for almost all cases. This difference in the performance exists because, as we mentioned before, there is not a low level heuristic good for all instances. Furthermore, in some cases a heuristic requires too many consistency checks and increments the average value.

3.2 Experiment Type II

For this experiment we test the general hyper-heuristic with another subset of instances from both the training and the testing set. This problems are harder to solve than those presented in the experiment type I. The uniform domain size (m) was set to the values 10, 15 and 20. All the other parameters were constant and set as follows: $n = 15$, p_1 =0.5 and p_2 =0.5. This experiment used the group I instances (B1, B2 and B3).

For this second experiment, the results show that the hyper-heuristic has a competitive performance, but is far from representing a drastic reduction in the number of consistency checks needed to solve the instances. For the training set, about the 42% of all the instances were solved with the same performance with both the best low level heuristic and the general hyper-heuristic. For the 37% of the instances, the results are less encouraging, because the require more than 15% of consistency checks to solve the problems. In the Testing set the results are similar to those in the training set. A 32% of the instances is solved with the same number of consistency checks (or with a worthless difference). Again, almost half of the instances required more than 15% consistency checks to be solved when using the general hyper-heuristic. These results are summarized in Table 3.

Table 3: Results from experiment type II.

	Reduction		Equal	Increment	
Set	> 15%	15% a 3%	± 3%	3% a 15%	> 15%
Training	0.00	0.00	41.67	20.00	36.67
Testing	0.00	0.00	31.67	16.67	51.67

If we compare the performance of the best result of the low level heuristics, the general hyper-heuristic and the average of the low level heuristics, we can observe that the approach exhibits a competitive behavior in most of the cases. However, in the cases where the hyper-heuristic reduces the number of consistency checks, does not represent a reduction greater than 3%. The Figure 4 shows the results of experiment I for the training set.

Figure 5 shows the results from experiment type II for the testing set.

In these figures we observe a similar behavior than in experiment I. When compared to the average number of consistency checks, the general hyper-heuristic proves to be a good approach to solve this kind of problems. At about 95% of all the instances in the training and testing set are solved with a smaller number of consistency checks than the average of the low level heuristics, representing a reduction greater than 15% in all cases.

3.3 Experiment Type III

For this last experiment, the number of variables and the uniform domain size were set to 10. The value of p_1 was set to 0.5. The parameter p_2 varied between the values 0.2 and 1.0, with increments of 0.2. This experiment used instances from the group II (D1, D2, D3, D4 and D5) from both the training and the testing set. We observe in the results from Figure 6 and 7 that, for a value of $p_2 \leq 0.4$ and $p_2 \geq 0.8$ both the best low level heuristic and the general hyper-heuristic show the same performance. For $0.4 < p_2 < 0.8$ there are minimal differences. The results show that the transition phase is located in p_2 around 0.4.

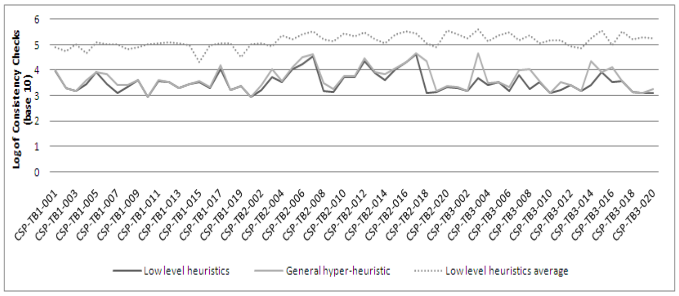

Figure 4: Results of experiment type II with instances from training set.

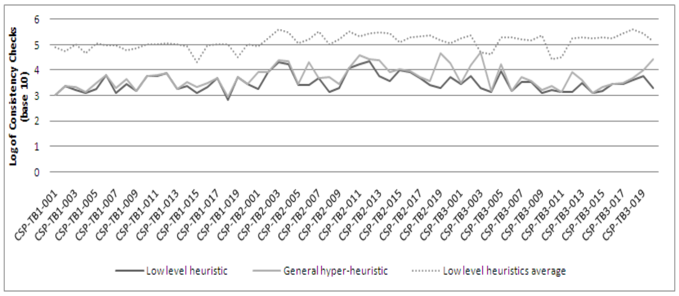

Figure 5: Results of experiment type II with instances from testing set.

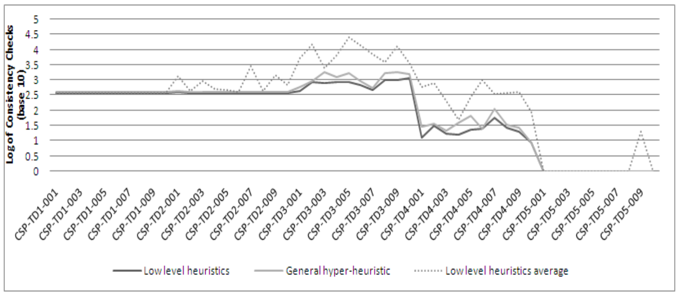

Figure 6: Results of experiment type III with instances from training set.

Figure 7 shows the results from experiment type III for the testing set.

In these graphics is clear that, when a problem has a small number of constraints or when has too many constraints is fast to faind the solution or to determine that there is not any. For values of p_2 smaller than 0.6 and greater than 0.8 we observe that both the best low level heuristic and the general hyper-heuristic have the same performance. For

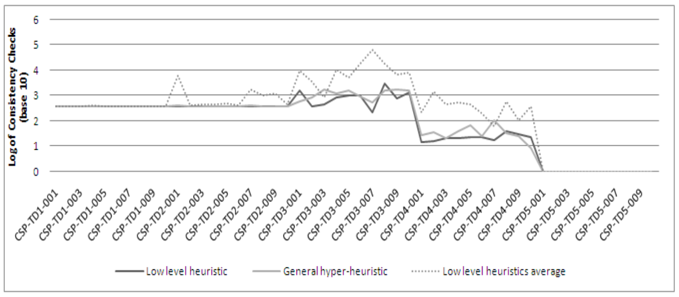

Figure 7: Results of experiment type III with instances from testing set.

values of $0.6 \leq p2 \leq 0.8$ there are differences that can be observed in the figures. In most of the cases the best result from the low level heuristics requieres less consistency checks than the general hyper-heuristic, but for some instances the hyper-heuristic reduces the number used by the bes heuristic. If we compare the general hyper-heuristic against the average of consistency checks of low level heuristics, the results are outstanding, because the hyper-heuristic is never worse than the average of the low level heuristics.

The detailed results from experiment III are presented in Table 4.

Table 4: Results from experiment type III.

	Reduction		Equal	Increment	
Set	> 15%	15% a 3%	± 3%	3% a 15%	> 15%
Training	0.00	0.00	63.33	15.00	21.67
Testing	0.00	0.00	58.33	13.33	28.33

3.4 Analysis on the hyper-heuristics produced

Looking at the results, it is clear in certain cases, that the method to form hyper-heuristics, and the hyper-heuristics themselves are not efficient yet, at least with respect to the number of consistency checks used for each instance if compared against the best result from all the single heuristics. The GA-based procedure has found hyper-heuristics composed of a set of rules which associate the problem state to a combination of variable ordering heuristics. However, it is important to get a better feeling of the real advantages or the proposed approach, and the practical implications of using it. For example, regarding the computational cost for delivering solutions by our approach, it is slightly higher than the time used by the simple heuristics which run in just few seconds. When comparing the hyper-heuristics versus the best result from the single low level heuristics the behavior is less efficient for several cases. However, we must recall that the best result was not always produced by the same heuristic, and in real cases there is no way to know that one heuristic is good for every instanc of the problems. There are certain characteristics that make one heuristic more suitable than other.

When comparing the hyper-heuristic against the consistency checks average used by all the low heuristics the results are outstanding. For every instance in the training set the hyper-heuristic requieres less consistency checks than the average from the low level heuristics. When applying it to the testing set the results are similar: a 0.02% of the instances were solved using more consistency checks than the average from low level heuristics. Compared to the average, the hyper-heuristic is capable of reducing in many cases more than the 15% of the consistency checks used by the average of the single heuristics.

Results also confirm the idea behind hyper-heuristics that by exploiting the problem-specific features by means of choosing a set of heuristics which best adapt to that, a better performance can be achieved.

We finally compare the best three individuals of the last generation. This is, three different hyper-heuristics generated by our solution model. The results presented in figure 8 show that these heuristics have a similar behavior for many instances, but there are differences that make the general hyper-heuristic (HH_1) the best option to solve a wide range of CSP instances.

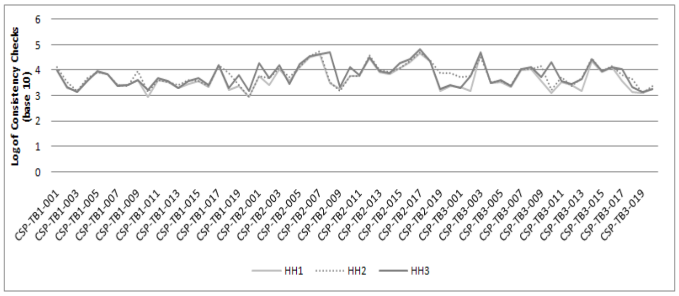

Figure 8: Results of experiment type II when using three hyper-heuristics with instances from testing set.

In table 5 we present a comparison between the percentage of reduction or increment of consistency checks used by these three hyper-heuristics and the best result from the low level heuristics.

Table 5: Results from the comparisson of three hyper-heuristics.

	Reduction		Equal	Increment	
	> 15%	15% a 3%	± 3%	3% a 15%	> 15%
HH_1	0.00	0.00	41.67	20.00	36.67
HH_2	0.00	1.67	11.67	18.33	68.33
HH_3	1.67	0.00	6.67	11.67	80.00

Even when HH_3 is capable of reducing in 1.67% the number of constraint checks used by the best low level heuristic, it presents a very poor behavior for most of the instances. HH_2 has also a less efective performance than HH_1. From these last results we can observe that HH_1 is the best hyper-heuristic from the three options.

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 very efficiently, in a few cases a 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 another algorithms for the CSP solving module, etcetera. It would be also interesting to work the approach for including not uniform domain size for every variable.

5. ACKNOWLEDGMENTS

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

6. 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] 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.

[5] R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 38(2):211–242, 1994.

[6] D. B. Fogel, L. A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, New York, 1966.

[7] M. Garey and D. Johnson. *Computers and Intractability.* W.H. Freeman and Company, New York, 1979.

[8] 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.

[9] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Adison Wesley, 1989.

[10] D. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis and first results. *Complex Systems*, pages pp. 93–130, 1989.

[11] B. L. Golden. Approaches to the cutting stock problem. *AIIE Transactions*, 8:256–274, 1976.

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

[13] J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] I. Rechenberg. *Evolutionstrategie: Optimierung technischer systeme nach prinzipien dier biolischen evolution.* Frommann-Holzboog, Stuttgart, 1973.

[19] 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.

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

[21] H. P. Schwefel. *Numerical Optimization of Computer Models.* Wiley, Chinchester, UK, 1981.

[22] 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.

[23] 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

[24] R. A. Wilson and F. C. Keil. *The MIT Encyclopedia of the Cognitive Science.* MIT Press, Cambridge, Massachussets, 1999.