Using Learning Classifier Systems to Design Selective Hyper-Heuristics for Constraint Satisfaction Problems

José C. Ortiz-Bayliss*, Hugo Terashima-Marín†, and Santiago E. Conant-Pablos†
*Automated Scheduling, Optimisation and Planning (ASAP)
School of Computer Science, University of Nottingham, UK
Email: Jose.Ortiz_Bayliss@nottingham.ac.uk
†Tecnológico de Monterrey
Monterrey, Mexico
Email: {terashima, sconant}@itesm.mx

***Abstract*—Constraint satisfaction problems (CSP) are defined by a set of variables, where each variable contains a series of values it can be instantiated with. There is a set of constraints among the variables that restrict the different values they can take simultaneously. The task is to find one assignment to all the variables without breaking any constraint. To solve a CSP instance, a search tree is created where each node represents a variable of the instance. The order in which the variables are selected for instantiation changes the form of the search tree and affects the cost of finding a solution. Many heuristics have been proposed to help to decide the next variable to instantiate during the search and they have proved to be helpful for some instances. In this paper we explore the use of learning classifier systems to construct selective hyper-heuristics that dynamically select, from a set of variable ordering heuristics for CSPs, the one that best matches the current problem state in order to perform well on a wide range of instances. During a training phase, the system constructs state-heuristic rules as it explores the search space. Heuristics with good performance at certain points are rewarded and become more likely to be applied in similar situations. The approach is tested on random instances, providing promising results with respect to the median performance of the variable ordering heuristics used in isolation.**

I. Introduction

The CSP is a fundamental problem in artificial intelligence, very important in theory, but also with many immediate applications ranging from vision, language comprehension, scene labelling, knowledge representation, scheduling and diagnosis (see for example: [1], [2] and [3]). A CSP is defined by a set of variables X, where each variable is associated a domain D_x of values subject to a set of constraints C [4]. The goal is to find a consistent assignment of values to variables in such a way that all constraints are satisfied, or to show that a consistent assignment does not exist.

Several deterministic methods to solve CSPs exist (see for example [5] and [1]) and solutions are found by searching systematically through the possible assignments to variables, guided by heuristics. It is a common practice to use Depth First Search (DFS) to solve CSPs. When using DFS to solve CSPs, each node of the tree represents a variable of the instance and, the deeper we go in that tree, the larger the number of variables that have already been assigned a feasible value. Every time a variable is instantiated, a consistency check occurs to verify that the current assignment does not conflict with any of the previous assignments given the constraints within the instance. When an assignment produces a conflict with one or more constraints, the instantiation must be undone, and a new value must be assigned to that variable. When the number of feasible values of the current variable decreases to zero, the value of a previously instantiated variable must be changed. This process is known as backtracking [6]. It is a common practice to complement the search with a constraint propagation method [7], which reduces the search space by removing unfeasible values from the domains of the remaining variables given the previous instantiations.

Learning Classifier Systems (LCS) [8], [9] are adaptive rule-based systems that automatically build the set of rules they manipulate. These rules are called classifiers, and they allow the system to respond to different situations of the environment. In this work, we describe a methodology to use LCS to respond to the changing features of CSP instances and dynamically select a variable ordering heuristic to be used at each step of the search.

This paper is organized as follows. In Sec. II a brief revision of similar studies is presented. The heuristics used in this investigation are described in Sec. III. Section IV describes in detail the learning classifier system and its main components. Section V presents the experiments and main results of this investigation. Finally, the conclusion and future work are presented in Sec. VI.

II. Background

The idea of dynamically selecting the most suitable solution method from a set of algorithms or heuristics is not new and it has been applied under different names in the literature. For example, reactive search embeds machine learning techniques into search heuristics for self-tuning of operating parameters [10], [11], algorithm portfolios attempt to allocate a period for running a chosen algorithm from a set of algorithms in a time-sharing environment [12], [13], hyper-heuristics [14], [15], [16], [17] are high-level methodologies that either select among different heuristics given the properties of the instance at hand (selective hyper-heuristics) or create

new heuristics based on the main components of a set of heuristics (generative hyper-heuristics). There is also a growing interest in adaptive memetic algorithms [18] that utilise a co-evolutionary framework for parameter tuning and operator selection. To be consistent with the existing definitions, the solution model described in this investigation falls into the category of selective hyper-heuristics.

LCS have been used before to produce selective hyper-heuristics in the domain of cutting stock [19], bin-packing problems [20], [21], [22] and more recently in Modularised Fleet Mix Problem [23]. With regard to CSPs, one of the first attempts to systematically map CSPs to algorithms and heuristics according to the features of the problems was presented by Tsang and Kwan [24]. In that study, the authors presented a survey of algorithms and heuristics for solving CSPs and proposed a relation between the formulation of the CSP and the most adequate solving method for that formulation. Petrovic and Epstein [25] studied the idea of producing mixtures of heuristics that work well on particular classes of instances. Also, algorithm portfolios for Constraint programming have been successfully studied before [26], [27] with promising results. More recently, Ortiz-Bayliss et al. [28] developed a study about heuristics for variable ordering within CSPs and a way to exploit their different behaviours to construct hyper-heuristics by using a static decision matrix to select the heuristic to apply given the current problem state. Terashima-Marín et al. [29] proposed a framework based on a messy genetic algorithm to generate hyper-heuristics for variable ordering in CSPs. Bittle and Fox [30] described a hyper-heuristic approach for variable and value ordering for CSPs based on a symbolic cognitive architecture augmented with constraint-based reasoning as the machine learning mechanism for their hyper-heuristics. More recent studies have also included the use of back-propagation and learning vector quantization neural networks to produce hyper-heuristics for variable and value ordering within CSPs [31], [32].

III. Ordering Heuristics

In this investigation we decided to use four variable ordering heuristics:

Minimum Remaining Values (MRV) [33], [34]. MRV selects the variable with the smaller number of available values in its domain.

Expected Number of Solutions (ENS) [35]. ENS selects the next variable in such a way that the subproblem maximizes the expected number of solutions $E(N)$, where the value of $E(N)$ is calculated as:

$$E(N) = \prod_{x \in X} |D_x| \times \prod_{c \in C} (1 - p_c). \quad (1)$$

where D_x represents the domain size of variable x and p_c the fraction of unfeasible tuples of values in constraint c.

Maximum Forward Degree (MFD) [36], [37]. MFD prefers the variables connected to the maximum number of uninstantiated variables (forward degree of the variable).

Kappa (K) [35]. K selects the variable that minimizes κ of the remaining subproblem. κ is a measure of constrainedness which serves as an indicator of the hardness of the instances with respect to their sizes. For example, instances with $\kappa \ll 1$ have many solutions while instances with $\kappa \gg 1$ are likely to be unsatisfiable. κ is calculated as follows:

$$\kappa = \frac{-\sum_{c \in C_x} \log_2(1-p_c)}{\log_2(D_x)} \quad (2)$$

We decided to include these heuristics because they have been widely studied in the literature, providing very competent results. In case of ties, lexical ordering is used over the tied variables.

Even though we know that value ordering also affects the cost of the search, the selection among different value ordering heuristics is beyond the scope of this investigation. In all cases, the values are ordered according to the Min-Conflicts heuristic [38], which prefers the value involved in the minimum number of conflicts (forbidden pairs of values between two variables).

IV. The LCS as a Selective Hyper-heuristic

Our solution approach uses a LCS as a selective hyper-heuristic. Given a CSP instance, the LCS has the task to decide which variable ordering heuristic to apply at every node of the search. Once a heuristic is applied, the features of the instance change, and a new subproblem rises. This process is repeated until a solution is found (or evidence that the instance is unsolvable is found). Thus, the process is dynamic and the selection of the next heuristic to be applied to the problem depends completely on the current problem state under exploration. Our model uses a CSP solver implemented in Java which uses AC3 [39] as constraint propagation method and backjumping [40] to improve the search. We are aware that other techniques may be used and obtain different results, but we consider that this configuration serves for the purpose of the investigation.

To be consistent with previous classifications about hyper-heuristic approaches, our approach is an offline, selective hyper-heuristic [41].

In a very general way, the LCS works as follows: the LCS is invoked every time a new variable needs to be selected (decision point, d_i). Then, at each decision point, the hyper-heuristic (represented by the set of classifiers in the system) is invoked and as a result, one variable ordering heuristic is used to determine the next variable to instantiate. Once we have decided which variable to instantiate, a value for that variable must be selected according to the Min-Conflicts heuristic. The selected value is assigned to the variable and the constraint propagation is executed. After propagation, the resulting instance has changed its structure, and the remaining variables have also had their domains pruned. Thus, the values of p_1 and p_2 are now different. The process is repeated until a solution is found or the instance is proved unsatisfiable.

Internally, LCS process can be summarized as:

a) The system reads the features of the instance at hand by using its detectors.
b) The system selects, from the population of available classifiers $[P]$, those that match the problem state and includes them in $[M]$.

c) One of the classifiers is selected from $[M]$ (more details will be provided in the next sections). All the classifiers in $[M]$ that contain the same action as the selected classifier will be part of the action set $[A]$.
d) The action a, of the selected classifier is applied to the instance (the action corresponds to one variable ordering heuristic). The instance is modified. If the search is not over, the process is repeated from step a.
e) If the search is over, the reinforcement mechanism assigns rewards to certain classifiers (more details will be provided in the next sections).

Figure 1 presents the general model of the LCS proposed.

A. The Classifiers

The LCS contains a list of classifiers that represent the rules that determine the behaviour of the system. Each classifier in the system consists of three elements:

1) A condition (the values of constraint density and tightness).
2) An action (a heuristic to apply).
3) The cumulative performance of the classifier that works as a quality measure.

The condition is composed by a set of values in the range $[0, 1]$, one value per feature in the problem state. We have tried to keep the problem state as simple as possible and that is why we decided to include only two well-studied problem features: the constraint density, p_1 and the constraint tightness, p_2. The constraint density is calculated as the proportion of existing constraints among all the variables over the maximum number of possible constraints. It provides an idea of how constrained an instance is, but it only considers the constraints and not the conflicts among those constraints (the forbidden pairs of values between two variables). Regarding the constraint tightness, we will calculate it as the proportion of existing forbidden tuples of values over the maximum number of conflictive tuples over the existing constraints.

We think that adding more features to the problem state representation may provide more information for the hyper-heuristics to make better decisions about which heuristic to apply. We decided to use p_1 and p_2 because there is evidence that these features can be used to characterize CSP instances and help to create a mapping from the state of the instance to one suitable heuristic [28].

In our classifiers, the condition representation does not indicate the presence or absence of a given feature in the CSP instance, but specific values of these features. The action specifies one variable ordering heuristic to apply when the condition of the classifier is satisfied. Finally, the cumulative performance measures the quality of the results when the classifier has been used before. The reinforcement component is responsible of updating the cumulative performance of each classifier based on its previous performance. All the values, except for the cumulative performance, are coded by using a binary representation.

Figure 2 presents an example of a valid classifier in our model. The first ten bits in the condition part correspond to the value of the constraint density, while the remaining ten

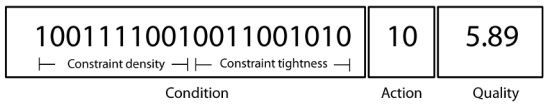

Fig. 2: An example of a classifier within the system

correspond to the value of the constraint tightness. Because the constraint density and tightness are always in the range $[0, 1]$, and 10 bits allow us to represent 1024 numbers, the range is divided in 1024 discrete steps (each step of 0.000976, which is the limit of the resolution). Then, the classifier shown in Fig. 2 represents the condition $(p_1 = 0.6181, p_2 = 0.1973)$. The interpretation of the action is direct: only two bits are necessary to code the four options of variable ordering heuristics. Then, the classifier in this example suggests to apply heuristic 2 when the problem state matches the condition described above.

B. Creating and Training the Classifiers

Because we are dealing with a multi-step environment, the reinforcement component updates to the corresponding classifiers (the ones that fired at distinct nodes of the search) only after the instance has been solved or proved to be unsatisfiable. There is one main question regarding how to estimate the performance of the classifiers: what should we look at to state that the system provided good or bad advice? In this research we are only interested in the first solution to the instances. Then, if the instance is satisfiable, there is no difference in which branch the solver takes to find the first solution. It does not matter how many times we have to backtrack, at the end, if the instance has at least one solution (and we have enough time), we will be able to find a branch that leads to one solution. What we propose is to give, to each one of the decisions over that branch, a reward inversely proportional to the number of values tried. Every value tried that is not successful is not part of the solution and contributes negatively to the search cost because it expands unnecessary nodes, increasing the search cost. In our approach, no penalties are given to poor quality classifiers. For example, if a given node in the solution branch tried 65% of its available values before being part of a solution, the reward for such node will be 0.35. Every time a reward is given to a classifier, it is added to its cumulative performance. Under this scheme, the largest reward per node any classifier can receive in the search tree is 1. It may be the case that one classifier fires several times during the search. Then, each classifier receives a reward for each node where it fires, according to the number of values tried. One important limitation regarding our reinforcement mechanism is that, at the moment, it can only obtain feedback from branches that lead to a solution. If the instance is unsatisfiable, no branch will lead to a solution and then, it will not be possible for the reinforcement component to assign a reward to any classifier. As a remark, the hyper-heuristics we develop in this paper are not able to learn from unsatisfiable instances but, as shown in Sec. V, they can be applied to both satisfiable and unsatisfiable instances and achieve very competitive results.

In order to produce more general behaviours, it is also needed to create new competent classifiers. To create new

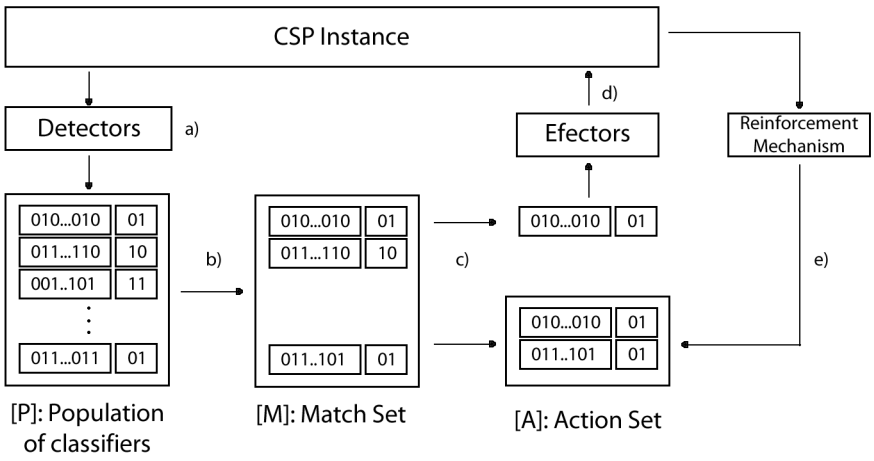

Fig. 1: The general model of the LCS proposed

classifiers, a generational genetic algorithm is used. Each time the action set $[A]$ is formed, there is a probability of 0.001 that the genetic algorithm occurs over population $[P]$. It has a very low probability of occurrence because the action set is formed every time a variable has to be selected (larger probability values would increase the size of the list of classifiers very fast). Standard genetic operators were implemented for this purpose: tournament selection of size two, one point crossover and one-bit mutation. In our implementation, there is no replacement of classifiers. Then, the new classifiers are always added to $[P]$. The reason for this is because we need to observe the performance of the new classifiers by being part of the search and we cannot estimate the performance at the moment when the classifiers are created. To avoid the uncontrolled growth of $[P]$, there is a removal routine that is invoked after each cycle, where a cycle corresponds to one complete step in the training process: the process of solving all the instances in a training set one time). This routine removes from $[P]$ all the classifiers which have not been part of a solution branch. This is, classifiers that were not active during the cycle.

Because p_1 and p_2 are coded in the condition of the classifier, we can directly compare how far these features are from the current problem state by using the euclidean distance. We decided that all the classifiers with a distance between their condition and the problem state smaller than a certain threshold θ_d, will be added to $[M]$ (the condition is satisfied if the values of p_1 and p_2 coded in the condition part of the classifier are close enough to the current problem state). If no classifier matches the current problem state, a covering process is invoked where a new random classifier that matches the problem state must be created. To create the new classifier, the exact problem state is coded into the condition of the classifier and a random variable ordering heuristic is selected for the action.

Once we have included all satisfied classifiers in $[M]$, it is time to select which action to apply. The selection process works in two different ways. When the LCS is being trained, it works in exploration mode. In exploration mode, the classifier to fire is randomly selected (where all the classifiers in $[M]$ have the same probability of being selected). This is made in order to test the performance of as many classifiers as possible. When the training is over, the LCS changes to exploitation mode, where only the best classifier in $[M]$ is selected to fire (the one with the largest cumulative performance). Also, in exploitation mode the covering procedure is deactivated. Then, if $[M]$ is empty, the classifier with the minimal distance from its condition to the current problem state will be selected to provide the heuristic to apply regardless of the value of θ_d.

Once a classifier has been selected, all the classifiers in $[M]$ that contain the same action as the selected classifier are added to $[A]$. At this point, the reinforcement component will keep track of which classifiers in $[A]$ correspond to the different nodes in the search tree. Then, when the instance is solved, the reinforcement module will assign rewards only to the classifiers that provided good advice: the ones that fired at the nodes of the solution branch. As we have mentioned before, at the moment our methodology cannot learn from unsatisfiable instances.

V. Experiments and Results

We produced 10 different hyper-heuristics as the result of 10 independent runs of the model. Then, 10 classifier systems were obtained as part of the hyper-heuristic generation process. Each classifier system represents a hyper-heuristic: the rules that decide which heuristic to apply given a certain problem state (defined by p_1 and p_2). The number of consistency checks reflecting the search effort is used as a cost estimator in the experiments. Thus, the smaller the number of consistency checks, the better the solution method.

The instances used in this investigation were randomly generated with model F [42]. Under this generation approach, up to $p_1 p_2 m^2 n(n-1)/2$ conflicts are independently selected (repetitions are allowed) out of the $m^2 n(n-1)/2$ possible conflicts (where n is the number of variables and m the uniform domain size). It then generates a constraint between connected vertices in the graph with exactly $p_1 n(n-1)/2$ edges and throws out any conflicts that involve disconnected nodes in this graph.

We produced three instance sets with 20 variables and 10 values in their domains: a set used for training, composed of 50 satisfiable instances, a testing set composed by 200 satisfiable instances and a second testing set that contains 200 unsatisfiable instances. These sets are called set A, B and C, respectively. Set A was used to train the hyper-heuristics and sets B and C were used only for testing purposes.

For each independent run, the hyper-heuristics were trained for 100 cycles by using set A (at each cycle, all the instances in set A are solved once). The initial population of the LCS is kept empty and all the classifiers are created during the training process. The probability of applying the genetic algorithm is 0.001 as specified in previous sections. The value for the minimum distance θ_d was set to 0.10 for all the runs.

A. Methods that Perform 'Well' on a Wide Range of Instances

Once the 10 hyper-heuristics were produced, we decided to measure their performance on set A, the same set that was used to train them. In order to make the proper comparison, each instance in the set was solved by using the four variable ordering heuristics and their results were saved for further analysis. Because the we are interested in reusable methods that perform 'well' on a wide range of instances (even though they do not perform as well as a very specialized method for some instances), we have decided to compare the ten hyper-heuristics against the median of the four variable ordering heuristics. MEDIAN will be defined as the median cost per instance of the four heuristics. The total cost of any method is calculated as the sum of the costs per instance of such methods. The cost is given in terms of consistency checks.

Table I presents the results of the total costs of MEDIAN and HH01-10 on the three sets of instances (the cases where the hyper-heuristics reduce the cost of MEDIAN on each set are shown in bold). We can observe that six of the ten heuristics produced reduce the total cost obtained by MEDIAN on set A. Nevertheless, not all the reductions are significant in practice. Hyper-heuristics HH02, HH04 and HH09 are interesting because their total cost represent savings with respect to MEDIAN of 34.6%, 34.58% and 28.19%, respectively. The cases where the hyper-heuristic present a bad performance with respect to MEDIAN are also important. HH01, HH07 and HH10 are extremely expensive methods. The fact that these hyper-heuristics present such a bad performance suggests that the model is not safe from producing very bad rules for the selective application of heuristics as the search progresses.

When tested on set B, we observed a general decrease in the performance of most of the hyper-heuristics. This reduction of performance occurs because the instances in set B, are unseen instances with similar properties to the ones in set A. These results suggest that it may be possible that some of the hyper-heuristics suffer from over fitting (see for example HH02 and HH09), which makes them very specialized on the training set, but incapable of generalizing to unseen cases.

Nevertheless the poor results of most of the hyper-heuristics on set B, HH04 and HH08 proved that the generalization of hyper-heuristics is feasible with our model. The cost produced by HH04 represents a saving of 28.19% with respect to the total cost of MEDIAN in set B. The reduction achieved by HH08 is insignificant in practice.

TABLE I: Total cost of MEDIAN and HH01-10 on the three sets of instances.

Method	Set A	Set B	Set C
MEDIAN	2404766	5659860	70625117
HH01	10587284	6167147	73383766
HH02	**1572595**	6917329	72595445
HH03	**2326553**	7240054	**69106131**
HH04	**1573148**	**4064183**	72392432
HH05	**2216734**	8721954	**67553311**
HH06	9088435	6329811	**69823492**
HH07	2453401	7466952	71172956
HH08	**2311319**	**5632825**	**69769309**
HH09	**1726811**	7602499	75008492
HH10	11919465	7404892	75880623

We have observed that our model produces hyper-heuristics that can be applied to satisfiable instances. We also showed that some of those hyper-heuristics are capable of generalizing and obtain competent results on both sets A and B, composed exclusively by satisfiable instances. We already mentioned the limitation that our reinforcement strategy presents when dealing with unsatisfiable instances, but here is still one remaining question: can we use hyper-heuristics trained only with satisfiable instances and still achieve acceptable results on unsatisfiable ones? To answer this question we tested HH01-10 on set C (composed only by unsatisfiable instances). The results of this comparison are shown in Table I, column set C.

On set C, four hyper-heuristics reduce the total cost obtained by MEDIAN. Unfortunately, these reductions are not significant in practice. HH05, which achieves the largest reduction with respect to MEDIAN, is capable of saving 4.35% of the consistency checks required by MEDIAN. Even though the reductions are not significant in practice, they provide evidence that our approach is able to produce competent hyper-heuristics for unsatisfiable instances even when only satisfiable instances are used during the training phase. It is important to mention that, even though we cannot claim that hyper-heuristics HH03, HH05, HH06 and HH08 are able to reduce the total cost of MEDIAN (with a significant reduction in practice), we can indeed say that these are good quality hyper-heuristics. In the sense of a general method, they are able to be used for unseen instances with distinct properties to the ones that were used to train them and still be competitive with respect to the median cost obtained from the four heuristics.

B. The Reliability Index

We have tested the performance of the hyper-heuristics against the total cost of MEDIAN on each set. A more interesting case occurs when we compare the performance of each hyper-heuristic against he performance of MEDIAN on each instance in the sets. In this experiment, we are interested in knowing the percentage of instances in each set where the cost of the search produced by the hyper-heuristics is lower than the cost produced by MEDIAN.

The results of this experiment are shown in Table II. This experiment provides information about the percentage of instances where each hyper-heuristic is better than MEDIAN, in terms of consistency checks; the experiment does not consider how large the reductions are. Thus, the results of Table II must be used only as a reference to estimate how reliable a hyper-heuristic is. For example, a hyper-heuristic that almost always

TABLE II: Reliability index for HH01-10 on the three sets of instances.

Method	Set A	Set B	Set C
HH01	52.0%	53.5%	52.5%
HH02	54.0%	47.5%	**62.5**%
HH03	**66.0**%	48.5%	47.0%
HH04	56.0%	**59.5**%	61.5%
HH05	56.0%	54.0%	46.5%
HH06	56.0%	46.0%	38.0%
HH07	42.0%	51.5%	**62.5**%
HH08	64.0%	47.0%	47.0%
HH09	46.0%	43.5%	61.0%
HH10	58.0%	44.0%	50.0%

TABLE III: Total cost of MRV. ENS, MFD, K and HH01-10 on the three sets of instances.

Method	Set A	Set B	Set C
MRV	5687250	5076235	73739901
ENS	4654948	7881271	68888714
MFD	**1530821**	**4965278**	82940806
K	3294987	7406344	**65116999**

performs better than MEDIAN regardless of the instance solved would be a very reliable hyper-heuristic. This hyper-heuristic may not produce the best results on each instance (compared against one specialized heuristic) but it would rarely produce extremely bad results. Then, the larger the percentages in the cells of Table II, the smaller the percentage of instances with bad results for the hyper-heuristic and the more reliable the hyper-heuristic is. We propose the term *reliability index* to refer to the percentage of instances that are solved with less consistency checks than MEDIAN.

In the sense of the previous explanation, HH04 is the most reliable hyper-heuristic with an average reliability index of 59% among the three sets. This is not surprising, because according to the results presented in Table I (the largest indexes on each set are presented in bold), HH04 is one of the most competent heuristics with respect to the total cost of the search. HH08, which is the best hyper-heuristic in terms of total cost, obtains an average reliability index of 52.67% (and a high reliability index on set A, 64%), which is also a good indicator about the quality of this hyper-heuristic.

C. A More Challenging Comparison

In previous sections we compared the performance of the ten hyper-heuristics produced against MEDIAN, in terms of total cost and analysing the results on each instance. Even though we are not looking for an algorithm capable of outperforming every single heuristic on every instance, we would like to know how our hyper-heuristics behave when compared against the best heuristic. For this reason, in this section we compare each hyper-heuristic against the best heuristic for each set. The results of the total cost of the search for each heuristic on the three sets are presented in Table III. The total cost of each hyper-heuristic on sets A, B and C was already presented in Table I.

For set A, MFD is the best heuristic, requiring a total cost of 1530821 consistency checks. Regarding the hyper-heuristics produced by our approach, none of them is better than MFD in terms of total cost of the search, but HH02 and HH04 are very close to the cost obtained by MFD. The best heuristic for set B is again, MFD (which is not surprising because both sets A and B contain instances with similar properties) with a total cost of 4965278 consistency checks. In this case, HH04 is better than MFD, obtaining a reduction of 18.15% consistency checks with respect to the total cost of MFD on set B. Finally, set C is dominated by K, with a total cost of 65116999 consistency checks. None of the hyper-heuristics seems to be match for this heuristic on set C. This result is expected because the performance of the hyper-heuristics when compared against MEDIAN was not as promising as in sets A and B.

The fact that no hyper-heuristic was able to reduce the total cost of the best heuristic on set C should not be misinterpreted as a bad result for our approach. We stress the idea that we are looking for reusable general methods. If we observe the total cost of MFD, which was the best heuristic on sets A and B, it now presents the worst results. As we mentioned before, heuristics tend to be very specialized and expose their weaknesses when the properties of the instances change. If we now look at the results of HH04 (the hyper-heuristic with the highest average reliability index) and HH08 (the best hyper-heuristic in terms of total cost when compared against MEDIAN), we can observe that their costs are much lower than the cost of MFD. This results suggest that our approach can indeed produce hyper-heuristics that are general enough to selectively apply a suitable heuristic according to the problem state under exploration to maintain the cost of the search under an acceptable value.

D. Time Analysis

One of the main criticisms to the hyper-heuristic approach is the additional cost of generating one hyper-heuristic. With regard to this, training a hyper-heuristic requires to solve all the instances in the training set every cycle of the training process. If the cost of solving the instances by using k heuristics is t, the cost of solving the whole set with a hyper-heuristic is expected to be around t/k. For a training process that runs for n cycles, an estimation of the cost of generating a hyper-heuristic is given by $\frac{nt}{k}$. Once the hyper-heuristic has been produced, every time it is invoked there is an additional cost for evaluating the problem state and deciding which classifier will fire at each node of the search. In all the experiments presented in this investigation, the cost of selecting the classifier seems to be insignificant during the search. We consider that a more detailed time analysis is required to clearly state the additional cost produced by the selection process and the running time of each heuristic.

Also, there is a difference in the execution time of the different heuristics at each node given by the operations used to calculate the next variable to instantiate. In this investigation, MRV is the fastest heuristic to make a decision (it performs fewer operations), followed by MFD. ENS and K are the slowest heuristics in this research. Because of this, the execution time of a hyper-heuristic also depends on the heuristics it uses at different nodes of the search. Of course, if using a slow heuristic at a given node represents significant savings in the exploration time of the remaining tree, it seems reasonable to use such heuristic instead of a faster one that does not produce such savings. Finding a good trade-off between these aspects is part of the hyper-heuristic generation process and, in this case, is left completely to the LCS.

E. Many Heuristics, Many Hyper-heuristics, Which One to Use?

In this case we already know which are the best heuristics for each set. We may question the need of a hyper-heuristic approach if we can evaluate the performance of k heuristics and then, use the best of them for subsequent trials. This may of course work if we plan to solve only a specific set of instances. In that case, it seems reasonable to put all the effort to obtain the best possible result on that set of instances. But, if we are considering to solve many sets of instances, with different properties, we need to think of a reusable method that can be applied to as many instances as possible with an acceptable performance. If that is the case, it is worth to invest time on producing a hyper-heuristic.

Heuristics are specialized to some classes of instances and are very sensitive to the properties of the instances in those classes. There is no guarantee that a good heuristic for one class of instances will work well on others (even though they have similar properties). The same occurs with hyper-heuristics; we cannot guarantee that they will behave well on all the instances of the problem. But, there is a high probability that a competent hyper-heuristic, when presented an unseen instance, will correctly select a suitable heuristic to keep the cost of the search at an acceptable level. Of course, not all the hyper-heuristics produced are good ones (see for example HH01, HH06 and HH10 on Table I on set A), but deciding which hyper-heuristic to keep is also a part of the generation process. How to automatically discriminate among hyper-heuristics is beyond the scope of this investigation. In our experience, there is no need to force the selection of only one hyper-heuristic. According to the needs of the user, it may be the case that two or more hyper-heuristics are to be used in practice. For example, should we have to decide among the hyper-heuristics produced, we would select both HH04 and HH08. We have already produce two competent and reliable methods, why not to use them both? Of course, if we had to stay only with one, the decision would be more difficult and we would try with the most reliable heuristic on set B, which we may think of as the most reliable hyper-heuristic and most capable of generalization.

VI. Conclusion and future work

This paper describes a selective hyper-heuristic approach based on a LCS. The approach produces acceptable results but more investigation is needed in order to improve the performance. The results suggest that it is feasible to map the CSP problem state (described by the constraint density and tightness) to one suitable ordering heuristic and that this mapping can be done through a LCS. This is not the first study where LCS have been used as selective hyper-heuristics, but it is the first time where the approach is applied to CSPs. The approach proposed in this investigation is different from previous works in the sense that it is not a standard LCS taken from literature; the classifier system was specifically modified for exploiting the features of search within CSPs.

One common misunderstanding is to think of a hyper-heuristic as some kind of 'super-algorithm' that needs to overcome all the heuristics on any instance to be worth the generation effort. Instead, hyper-heuristics are a general solution technique that tries to behave well on different instances. The hyper-heuristics presented in this investigation are, in general, very competent when compared against the median cost of the four heuristics and, when compared against the best heuristics on each set, they proved to be capable of competing and achieving promising results. Specially on the set of unseen unsatisfiable instances, some hyper-heuristics proved to be a generic solution that takes advantage of the selection of different heuristics to make up for the weaknesses of some of them.

We have used the constraint density and tightness as features to characterize CSP instances. At this point, we think that more features are needed to produce better results. Deciding which additional features to include in the model is a very important topic that needs to be properly analysed in a further study. Also, we know that random CSP instances are useful to test new algorithms but we are aware of the need of using real instances to test our model and analyse its practical contribution.

Also, one of the main concerns for future work is the improvement of the reinforcement module. The current reinforcement mechanism is very simple and it can only learn from satisfiable instances. Even though we proved that the hyper-heuristics trained exclusively with satisfiable instances can be applied to unsatisfiable instances without significant decrease of the performance, it is important to extend the model to learn from instances with no solution.

Finally, along with the improvement of the reinforcement mechanism, the inclusion of value ordering heuristics to the selection process of the hyper-heuristic is another important consideration for future work.

VII. Acknowledgments

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

References

[1] R. Dechter, "Constraint networks," in *Encyclopedia of Artificial Intelligence*. Wiley, 1992, pp. 276–286.

[2] D. Waltz, "Understanding line drawings of scenes with shadows," in *The Psychology of Computer Vision*. McGraw-Hill, 1975, pp. 19–91.

[3] R. Dechter, "Constraint satisfaction," in *MIT Encyclopedia of the Cognitive Sciences (MITECS)*, 1998.

[4] C. P. Williams and T. Hogg, "Using deep structure to locate hard problems," in *Proceedings of AAAI'92*, 1992, pp. 472–477.

[5] V. Kumar, "Algorithms for constraint satisfaction: a survey," *AI Magazine*, vol. 13, no. 1, pp. 32–44, 1992.

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

[7] E. C. Freuder, "Synthesizing constraint expressions," *Communications of the ACM*, vol. 21, no. 11, pp. 958–966, 1978.

[8] J. R. Holland, *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[9] L. B. Booker, D. E. Goldberg, and J. H. Holland, "Classifier systems and genetic algorithms," *Artif. Intell.*, vol. 40, no. 1-3, pp. 235–282, 1989.

[10] R. Battiti, "Reactive search: Toward self-tuning heuristics," in *Modern Heuristic Search Methods*, V. J. Rayward-Smith, I. H. Osman, C. R. R. CR, and G. D. Smith, Eds. John Wiley & Sons, 1996, pp. 61–83.

[11] R. Battiti and M. Brunato, "Reactive search: Machine learning for memory-based heuristics," Università Degli Studi Di Trento, Tech. Rep., 2005.

[12] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, vol. 27, pp. 51–53, 1997.

[13] M. Gagliolo and J. Schmidhuber, "Dynamic algorithm portfolios," *Annals of Mathematics and Artificial Intelligence*, vol. 47, pp. 3–4, 2006.

[14] J. Denzinger, M. Fuchs, M. Fuchs, F. F. Informatik, and T. Munchen, "High performance ATP systems by combining several AI methods," in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97*. Morgan Kaufmann, 1997, pp. 102–107.

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

[16] P. Ross and J. Marín-Blázquez, "Constructive hyper-heuristics in class timetabling," in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC'05)*, vol. 2. IEEE Press, 2005.

[17] A. S. Fukunaga, "Evolving local search heuristics for sat using genetic programming," in *Genetic and Evolutionary Computation Conference (GECCO'04)*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 3103, pp. 483–494.

[18] Y.-S. Ong, M.-H. Lim, N. Zhu, and K. W. Wong, "Classification of adaptive memetic algorithms: a comparative study," *IEEE Transactions on Systems, Man, and Cybernetics*, 2006.

[19] H. Terashima-Marín, E. J. Flores-Álvarez, and P. Ross, "Hyper-heuristics and classifier systems for solving 2d-regular cutting stock problems," in *Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO'05)*. ACM, 2005, pp. 637–643.

[20] J. G. Marín-Blázquez and S. Schulenburg, "Multi-step environment learning classifier systems applied to hyper-heuristics," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation (GECCO'06)*. ACM, 2006, pp. 1521–1528.

[21] ——, "A hyper-heuristic framework with XCS: learning to create novel problem-solving algorithms constructed from simpler algorithmic ingredients," in *Proceedings of the 2003-2005 international conference on Learning classifier systems*. Springer-Verlag, 2007, pp. 193–218.

[22] P. Ross, S. Schulenburg, J. G. Marín-Blázquez, and E. Hart, "Hyper-heuristics: Learning to combine simple heuristics in bin-packing problems," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*. Morgan Kaufmann, 2002, pp. 942–948.

[23] K. Shafi, A. Bender, and H. A. Abbass, "Multi objective learning classifier systems based hyperheuristics for modularised fleet mix problem," in *Simulated Evolution and Learning*, ser. Lecture Notes in Computer Science, L. Bui, Y. Ong, N. Hoai, H. Ishibuchi, and P. Suganthan, Eds. Springer Berlin Heidelberg, 2012, vol. 7673, pp. 381–390.

[24] E. Tsang and A. Kwan, "Mapping constraint satisfaction problems to algorithms and heuristics," Department of Computer Sciences, University of Essex, Tech. Rep. CSM-198, 1993.

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

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

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

[28] 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)*. IEEE Press, 2010, pp. 1–8.

[29] 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)*. ACM, 2008, pp. 571–578.

[30] 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*. ACM, 2009, pp. 2209–2212.

[31] J. Ortiz-Bayliss, H. Terashima-Marín, and S. Conant-Pablos, "Neural networks to guide the selection of heuristics within constraint satisfaction problems," in *Pattern Recognition*, ser. Lecture Notes in Computer Science, J. Martínez-Trinidad, J. Carrasco-Ochoa, C. Ben-Youssef Brants, and E. Hancock, Eds. Springer, 2011, vol. 6718, pp. 250–259.

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

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

[34] P. W. Purdom, "Search rearrangement backtracking and polynomial average time," *Artificial Intelligence*, vol. 21, pp. 117–133, 1983.

[35] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T.Walsh, "An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem," in *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP'96)*, 1996, pp. 179–193.

[36] E. C. Freuder, "A sufficient condition for backtrack-free search," *Journal of the ACM*, vol. 29, no. 1, pp. 24–32, 1982.

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

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

[39] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[40] 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*, 1978, pp. 268–277.

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

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