Mapping the Performance of Heuristics for Constraint Satisfaction

José Carlos Ortiz-Bayliss, Ender Özcan, Andrew J. Parkes and Hugo Terashima-Marín

***Abstract*— Hyper-heuristics are high level search methodologies that operate over a set of heuristics which operate directly on the problem domain. In one of the hyper-heuristic frameworks, the goal is automating the process of selecting a human-designed low level heuristic at each step to construct a solution for a given problem. Constraint Satisfaction Problems (CSP) are well know NP complete problems. In this study, behaviours of two variable ordering heuristics Max-Conflicts (MXC) and Saturation Degree (SD) with respect to various combinations of constraint density and tightness values are investigated in depth over a set of random CSP instances. The empirical results show that the performance of these two heuristics are somewhat complementary and they vary for changing constraint density and tightness value pairs. The outcome is used to design three hyper-heuristics using MXC and SD as low level heuristics to construct a solution for unseen CSP instances. It has been observed that these hyper-heuristics improve the performance of individual low level heuristics even further in terms of mean consistency checks for some CSP instances.**

I. Introduction

Hyper-heuristics are high level methods that attempt to learn and exploit patterns within the relative effectiveness of different heuristics. Hence, before attempting to use them is it reasonable to ask: "Can we first demonstrate and understand that there is something that hyperheuristics have a chance of learning and exploiting?". In this paper, we directly look for such 'exploitable patterns'. We do this in the context of random CSP problems, though of course we hope that the results will ultimately have wider applicability.

The general Constraint Satisfaction Problem (CSP) can be defined by a set of variables X, where each variable can get a value from a corresponding domain D subject to a set of constraints C [44]. Solving a CSP requires a search for the values for each variable in such a way that their assignments do not violate any given constraint [14]. CSPs belong to the NP-Complete class [23] and there is a wide range of theoretical and practical applications (see for example [32], [29], [18]).

José Carlos Ortiz-Bayliss is with the Center For Intelligent Computing and Robotics, Tecnológico de Monterrey, Campus Monterrey, Monterrey, Mexico (email: a00796625@itesm.mx).

Ender Özcan is with the School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, United Kingdom (phone: +44 115 95 15544; email: exo@cs.nott.ac.uk).

Andrew J. Parkes is with the School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, United Kingdom (http://www.cs.nott.ac.uk/∼ajp/).

Hugo Terashima-Marín is with the Center For Intelligent Computating and Robotics, Tecnológico de Monterrey, Campus Monterrey, Monterrey, Mexico (phone: +52 818158-2045; email: terashima@itesm.mx).

As a classic search problem, a CSP is usually solved using a Depth Search Tree (DFS) [41] where every variable represents a node in the tree. Every time a variable is instantiated, the constraints must be checked to verify that none of them are violated. When an assignment is in conflict with one or more constraints, the instantiation must be undone, and another value must be considered for that variable. If there are not any values available, the value of a previously instantiated variable must be changed. One of the well known techniques is *backtracking*. This technique always goes up a single level in the search tree while all values for a variable have been tested. *Backjumping* is another powerful technique for retracting and modifying the value of a previously instantiated variable. Backjumping increases efficiency and cuts the search space by going up more levels in the search tree. Another way to reduce the search space is using *constraint propagation*. With constraint propagation the idea is to propagate the effect of one instantiation to the rest of the variables due to the constraints among the variables. Thus, every time a variable is instantiated, the values of the other variables that are not allowed because of the current instantiation are removed. Because of this, only allowed values for the variables remain.

When solving a CSP, the order in which the variables are instantiated, and the order in which the values for those variables are selected affects the complexity of the search [38]. This ordering is done via a dynamic fashion where the order of the variables is formed through the search based on some criteria over the features of the remaining uninstantiated variables. In this study, we used two well known variable ordering heuristics: *Max-Conflicts* and *Saturation Degree* [9]. It is important to recall that none of these heuristics has been proven to be more efficient than the other one for CSP instances. We also use Min-Conflicts [30] as value ordering heuristic to improve the performance of the system.

There is a relation between the structure of CSP and the difficulty of solving them with search algorithms [11]. Specifically, the median search cost of many search algorithms for CSP exhibits a sharp peak as a structural parameter is varied. This peak coincides with the transition from under-constrained to over-constrained instances, often manifested as an abrupt change in the probability that an instance has a solution. This peak corresponds approximately to the value of p_2 at which half the problems are insoluble and half are soluble, the point referred to by Crawford and Auton as the crossover point [12]. Inside this region, the most difficult soluble problems and the most difficult insoluble problems co-exist [42]. For larger values of p_2, as

problems become uniformly insoluble, the fall in the median consistency checks is much more gradual, and it is an over-simplification to describe this side of the phase transition as a region of easy problems [42]. Although it does become easy to prove insolubility for p_2 close to 1, many of the problems in this region are easy only by comparison with the insoluble problems occurring in the phase transition [42].

Hyper-heuristics are methods that attempt to learn and exploit 'patterns' and regularities in the relative effectiveness of different heuristics. For example, that relative effectiveness differ in some non-trivial fashion depending on the problem characteristics. It is essential for this that there really is some such non-trivial pattern to be learned. A trivial pattern would be 'noise' putting the hyper-heuristic in danger of trying to learn non-statistically significant patterns that will not generalise. Another trivial pattern would be total dominance, in which case a heuristic might simply be omitted. In this paper, we show an example, of such a non-trivial pattern of relative effectiveness of two heuristics, and some preliminary work on how a hyper-heuristic might exploit it. Our intention is that it will also lead to more focussed work on understanding how and why the effectiveness of heuristics vary with the problem instance characteristics.

Section II presents a brief description about hyper-heuristics and their applications to CSP in previous studies. Section III describes the heuristics used in our CSP solver. The preliminary experiments about the heuristics are presented in Section IV. In Section V we present the main experiments related to the construction of the hyper-heuristics and the results obtained. Finally, Section VI presents the conclusion and future work.

II. Hyper-heuristics

Hyper-heuristics are identified as the methodologies that search the space generated by a set of low level heuristics for solving computationally hard problems. Hyper-heuristic terminology was first introduced by Denzinger et al. [15] in 1997, however the idea of combining strengths of multiple heuristics goes back to 1960s ([17], [13]). Previous surveys on hyper-heuristic methodologies can be found in [5], [40], [10] and [7]. There are many other related studies on adaptive intelligent search methodologies. For example, *reactive search* embeds machine learning techniques into search heuristics for self-tuning of operating parameters [1], [2]. *Algorithm portfolios* attempt to allocate a period for running a chosen algorithm from a set of algorithms in a time-sharing environment [26], [22]. There is also a growing interest in *adaptive memetic algorithms* [27], [34] that utilise a co-evolutionary framework for parameter tuning and operator selection.

Hyper-heuristics can be divided into two main classes: methodologies that *select* or *generate* heuristics. More on the classification of hyper-heuristics can be found in [10] and [8]. Burke et al. [6] provides an overview of the latter type of hyper-heuristics. As representative studies on the hyper-heuristic methodologies to generate heuristics, Dimopoulos and Zalzala evolve priority dispatching rules for the single machine scheduling problem in [16], while Fukunaga uses genetic programming as an automated heuristic discovery system for the SAT problem ([19], [20], [21]). On the other hand, most of the hyper-heuristic methodologies to select low level heuristics are based on a similar iterative framework in which the search is performed in two successive stages: heuristic selection and move acceptance. A low level heuristic can be *perturbative* or *constructive*. A perturbation heuristic operates on a complete solution and modifies it for improvement, whereas a construction heuristic incrementally builds a complete solution. Özcan et al. [36] and Bilgin et al. [3] provide the details of some selection hyper-heuristics based on perturbation low level heuristics. As an example, Nareyek [33] uses a selection hyper-heuristic that combines reinforcement learning as a heuristic selection method and accepts all moves. The solution methodology in this study is based on a CSP solver (DragonBreath engine) which chooses a neighbourhood for a given constraint. The selection of the neighbourhood is performed through online learnt utility values for each low level heuristic. One of the recent studies that attempts to test the level of generality of a hyper-heuristic is provided in [37] over different types of vehicle routing problems. The authors employ a hyper-heuristic which is based on a variant reinforcement learning heuristic selection mechanism combined with simulated annealing move acceptance.

In this study, our focus will be on the hyper-heuristic methods that select construction low level heuristics. The previously proposed hyper-heuristics applied for CSP are meta-heuristics as seen on the hyper-heuristic approach presented by Terashima-Marín et al. [43], who proposed an evolutionary framework to generate hyper-heuristics for variable ordering in CSP. A genetic algorithm has the task to create the rules for the application of the variable ordering heuristics. LaTorre et al. [28] have also applied a genetic algorithm to solve combinatorial problems with promising results. An attempt to extend the results presented by Terashima et al. [43] was done by Ortiz et al. [35], where a second stage to the evolutionary model is incorporated by using a neural network module.

The solution approach presented in this study, can be explained as follows: (1) First we solved many instances with different features and obtained data about how good each of the heuristics was for every instance.(2) We used that information to generate a hyper-heuristic that decides which heuristic to apply and iteratively construct a solution to the instances. As we can see, there is a training stage where the information is gathered and a second stage where the information from stage 1 is used.

III. Constructing a solution to a CSP

A solution for a given CSP is constructed iteratively based on two variable ordering heuristics: Max-Conflict and Saturation Degree. Each one of these heuristics reorder the variables to be instantiated dynamically at each step during the construction process. A value among valid values must be selected and assigned to the chosen variable considering the

constraints. These values are also ordered using Min-Conflict heuristic. A CSP solver is implemented that makes use of constraint propagation and backjumping. The heuristics used in this system are briefly explained in the following subsections.

1) Max-Conflict (MXC): The Max-Conflict heuristic is very simple, and the main idea is to select the variable that is involved in the larger number of conflicts among the constraints in the instance. This instantiation will produce a subproblem that minimises the number of conflicts among the variables left to instantiate. This heuristic is a variation of the Min-Conflicts heuristic for value ordering [30], [31], but it was adapted to work for variable ordering and provides the benefit of being very easy to implement and fast to be executed.

2) Saturation degree (SD): The Saturation Degree heuristic has been more frequently used for graph colouring, but it is possible to adapt it for being applied to the variable ordering problem in general CSP. The degree of a node is defined as the number of nodes adjacent to it. Thus, the saturation degree of a node is the number of adjacent nodes that have already been instantiated. In this way, the idea of the saturation degree heuristic is to take an advantage of the topology of the constraint graph to select the variable which participates in the larger number of constraints with instantiated variables [41]. In other words, this heuristic uses information from the constraint graph to select the most restricted variable with instantiated variables.

We have also investigated other heuristics like Minimum Remaining Values (MRV), Most Constrained Variable (MCV), Rho, $E(N)$ and Kappa (see for example [25], [24]); but in this paper we have focussed on just the two that we previously mentioned: MXC and SD, not because they are necessarily the absolutely best choices, but because they are simple and illustrate most clearly the patterns that can occur. We intend to give a more complete study in future work.

With regard to the value ordering, we have included Min-Conflicts to our solver. The Min-conflicts heuristic prefers the value involved in the minimum number of conflicts. This heuristic is trying to leave the maximum flexibility for subsequent variable assignments. If we select the value that is involved in the minimum number of conflicts, we can suppose that the resulting subproblem will have more solutions than the other subproblems.

IV. Preliminary experiments with ordering heuristics

The binary CSP instances for the experiments are randomly generated in two stages. In the first stage, a constraint graph G with n nodes is randomly constructed and then, in the second stage, the incompatibility graph C is formed by randomly selecting a set of edges (incompatible pairs of values) for each edge (constraint) in G. More details on the framework for problem instance generation can be found in [39] and [42]. The parameter p_1 determines how many constraints exist in a CSP instance and it is called constraint density, whereas p_2 determines how restrictive the constraints are and it is called constraint tightness.

The number of consistency checks reflecting the search effort is used as a performance measure in the experiments. Thus, the smaller the number of consistency checks is, the better the heuristic is. For the first experiment, we created a grid of points covering the range $[0, 1]$ for p_1 and p_2, with increments of 0.025. This set of instances is called Set I. In this way, we obtain a grid of 41×41 points. For every point in the grid we generated 50 random CSP using random model B where all the constraints are defined in extension [38]. In this model, there should be exactly $p_1 n(n-1)/2$ constraints (rounded to the nearest integer), and for each pair of constrained variables, the number of inconsistent pairs of values should be exactly $m^2 p_2$ (where m is the uniform domain size of the variables). Each instance was generated containing 20 variables, a uniform domain size of 10 values and the corresponding values of p_1 and p_2 depending on the position on the grid. For each variable ordering heuristic, we obtained the mean, the median and the standard deviation of the consistency checks over the 50 instances of every point on the grid and then used those values for further analysis. For the second experiments, we generated 50 random instances with 25 variables and 12 values in their domains, but only for the region in the space where $p_1 = p_2$, these last instances correspond to Set II. Finally, we generated Set III which contains 50 instances per point in the region of the space where $p_1 = p_2$. The instances in Set III were generated with 20 variables and 10 values in the domains and were used to test the hyper-heuristics against each other and the two low level heuristics.

An Intel Core 2 Duo windows machine having 4 GB of memory is used during the experiments. The overall time for generating and solving the problem instances in Set I, II and III took 498, 20 and approximately 5 minutes, respectively.

A. Mean Consistency Checks

Figure 1 presents the plots of the logarithm base 10 of the mean consistency checks used per each heuristic when solving the grid of instances from Set I. It is not clear from these pictures which heuristic is better in some regions of the space, for example, close or inside the phase transition.

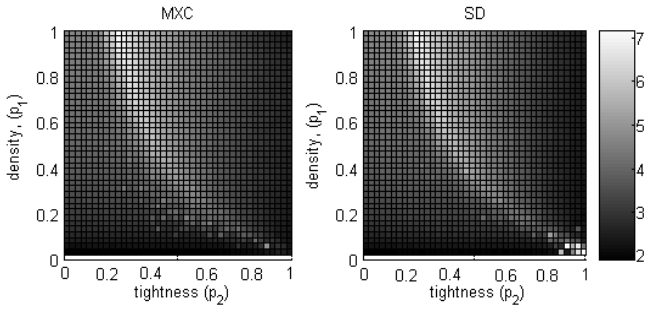

Fig. 1. Plot of the mean consistency checks (with the logarithm given by the darkness of the pixel) for the instances in Set I.

If we look into the bottom right part of the plot for SD

we can observe that there is an increment in the number of consistency checks for some of the points in that area. Sometimes, SD makes really bad decisions when it deals with low-constrained instances but with a very high tightness. The standard deviation in this region of the space, when using SD is really high and, as the mean is very sensitive to extreme values, the mean consistency checks for SD at points with large p_2 and small p_1, is very large.

If we observe Figure 1 we can realize that there is a relation of the hardness peak to the satisfiability threshold. As is well-known the hardest parts tend to associate with thresholds, though not all thresholds are hard. The hardness of a 'threshold instance' varies significantly depending on its location on the threshold.

Using the information of the sample means and the standard deviations at every point for each heuristic, we were able to produce a plot where we show the best heuristic for every point. To decide about the best heuristic, we developed a test of hypothesis with 5% of significance about the population means of both heuristics. If there is no statistical evidence that the population means are different, the point in the space is plotted using the symbol 'o'. If the population mean of MXC is smaller than the population mean of SD, the point is plotted using a '+'. Finally, if the population mean of SD is smaller than the one for MXC, the point is plotted with the symbol 'x'. Figure 2 shows the best heuristic for every point in the grid based on the statistical analysis of the population means for every point in the grid.

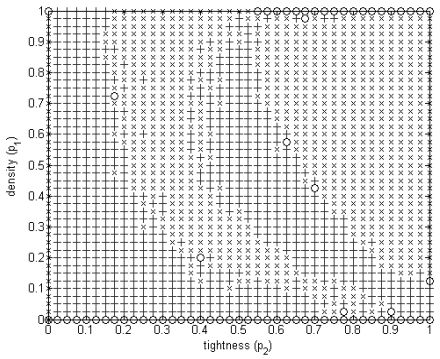

Fig. 2. Best heuristic for every point in the grid.

In order to have a closer look at what is happening at some regions of the search space, we decided to present a plot with the consistency checks used by both MXC and SD for the 50 instances at two different points in the space. Figure 3 shows the consistency checks of MXC against those of SD for the point $(p_2 = 0.25, p_1 = 0.25)$. As the previous analysis expected, the suggests that MXC is a better option for the instances at this point of the grid. At this point the mean of consistency checks by MXC is 3174 and the mean for SD is 3749, which means that the mean of consistency checks of MXC reduces in about 15% the number of consistency check that SD, in average, requires to solve the instances at the point$(p_2 = 0.25, p_1 = 0.25)$. The standard deviation of MXC is larger than the standard deviation of SD, 1553.81 and 619.91, respectively. The high standard deviation of MXC occurs because of one isolated instance that required a much larger number of consistency checks than the rest of the instances.

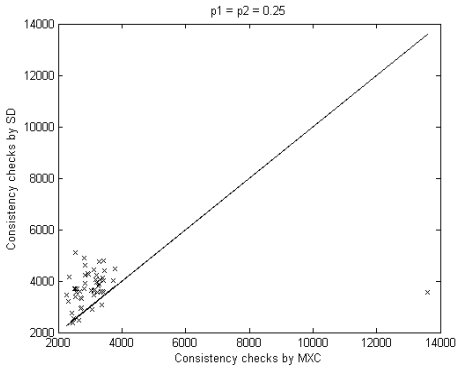

Fig. 3. Consistency checks by MXC and SD at point (0.25, 0.25).

The results for the point $(p_2 = 0.70, p_1 = 0.90)$ show that SD is a better option for instances at this point (Figure 4). The mean of consistency checks done by MXC is 6242 while the number for SD is 4953. This difference represents a reduction close to 20% in the mean of consistency checks. If we observe the standard deviations of consistency checks done by the two heuristics at point $(p_2 = 0.70, p_1 = 0.90)$, we can observe that MXC has a standard deviation of 2574.9 and SD of 1235.9. This suggests that SD is more reliable because the number of consistency checks from one instance to another do not change as much as they do when using MXC.

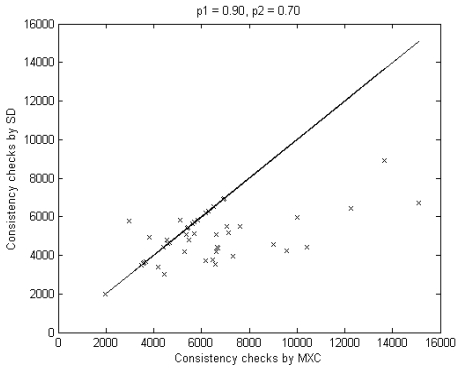

Fig. 4. Consistency checks by MXC and SD at point (0.70,0.90).

It is very important to notice that all the experiments and tests prove the idea that there is not one single heuristic capable of solving all the instances and achieve the best results. For some points in the space, MXC is the best option while for others, the best option becomes SD. If we are able to use this information to construct a hyper-heuristic that correctly selects the best heuristic for each point, we expect to achieve better results than with the application of one single heuristic through all the space.

V. Experiments and main results

This section presents the main results of the research. We include the constructive process for the hyper-heuristics and the tests we did to these hyper-heuristics.

A. Behavior of heuristics

To have a clearer picture of the behaviour of the heuristics, we selected just a slice from the space where $p_1 = p_2$. From this slice, we plotted the logarithm base 10 of the mean consistency checks for both heuristics and also the probability that the each point is satisfiable. Figure 5 shows in the logarithm base 10 of the mean consistency checks made by MXC, SD and the probability that an instance at point $p_1 = p_2$ is satisfiable. As we can observe, depending on the position in the space, the heuristics present a different behaviour. It is also possible to identify that there are some points where the heuristic changes from being the best option to be a not so good choice. At this point we do not have enough information to conclude about why these changes in the behaviour take place at those points.

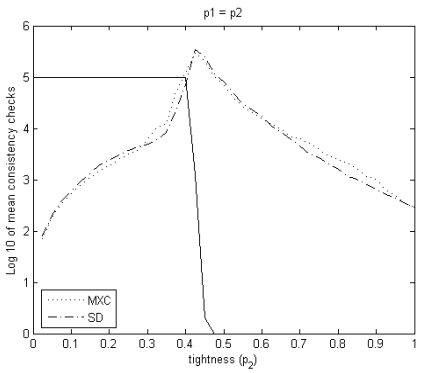

Fig. 5. Slice in the space where $p_1 = p_2$ for the instances in Set I.

We also tried larger instances to generate the same plot and confirm the idea that the changes occur not only for instances with 20 variables and 10 values. For this new plot for larger instances, we only generated 50 instances in the region where $p_1 = p_2$. Every instance contains 25 variables and a uniform domain size of 12 values. Figure 6 shows the results of the mean consistency checks for each heuristic at every point where $p_1 = p_2$.

It is easy to observe from Figure 6 that the points where the change in the best heuristic are still present for larger instances. Once again, we do not have a clear idea about what is happening at those points that produces the change in the behaviour of the heuristics that we can see on both plots. What we can suppose from these pictures is that at the phase transition there is too much noise and it would be difficult to decide which heuristic is the best for that region. Figures 5 and 6 suggest that for small values of tightness ($p_2 < 0.27$), the best heuristic is MXC; and that for higher values ($p_2 > 0.65$) the best heuristic is SD.

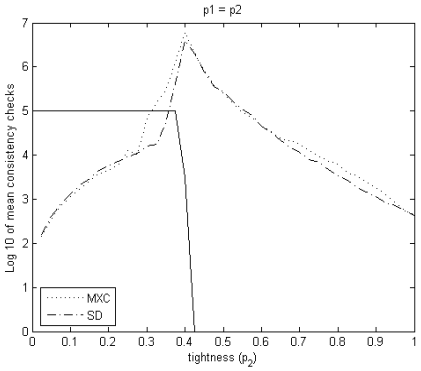

Fig. 6. Slice in the space where $p_1 = p_2$ for the instances in Set II.

B. The hyper-heuristic approach

We have shown that each of the heuristics takes an advantage of some specific characteristics of the instances. Thus, a more efficient alternative is to apply a different heuristic depending on the current instance state and not the same all over the search.

A first attempt to use the information obtained from the preliminary experiments to construct a simple static hyper-heuristic consisted in taking the information from Figure 2 to create a binary decision matrix. This matrix contains a 0 where MXC must be applied and 1 when SD must. This static hyper-heuristic is called SHH and works as follows: at each node of the search, we measure the density and the tightness of the instance at hand and we look into the decision matrix and select the low level heuristic which is the best option for that point. We use that heuristic to instantiate one variable of the instance and the process is repeated for each node in the search tree until a solution is found. The way we calculate the density (p_1) and the tightness (p_2) at each node of the search is done as follows: p_1 is estimated as the number of non empty edges in the constraint graph over the maximum possible number of edges. It is, if the number of nodes is n (the number of variables in the instance), the maximum possible number of edges is given by $n \times (n-1)/2$. For p_2, we estimate the tightness of each constraint and then we obtain an average to calculate the tightness of the whole instance. If a given constraint c between two variables v_i and v_k with domains d_i and d_j, respectively; prohibits r combinations of values between the two variables, then the tightness (p_2) of that constraint is given by $r/(|d_i| \times |d_j|)$.

Once one variable has been instantiated, the resulting subproblem is no longer a random instance with the same distribution of the original instance. In fact, the structure of this instance could be very different from the original random instance. If we consider this, it may not be the best option to have a static decision matrix because the information to select the low level heuristic during the search may probably be wrong. This is the reason why a second idea for a simple hyper-heuristic is needed. This idea consists of using a probabilistic approach because we think that a softer roulette-wheel decision is less sensitive to errors

than a 0/1 decision, but this may be confirmed with further analysis. The idea consists of creating a probability matrix, where the probability of selecting one heuristic over the other at every point of the grid depends on the number of instances that each one proved to be the best option. Figure 7 shows the probability matrix for the grid of instances from Set I. It is the probability for an instance at the given $(p1, p2)$ that MXC will do better than SD, as determined empirically by taking many instances and simply measuring which heuristic performs best. It is our intention to use data such as this to help drive the decisions taken by a hyper-heuristic. The probability of applying SD is the complement of the probability of using MXC.

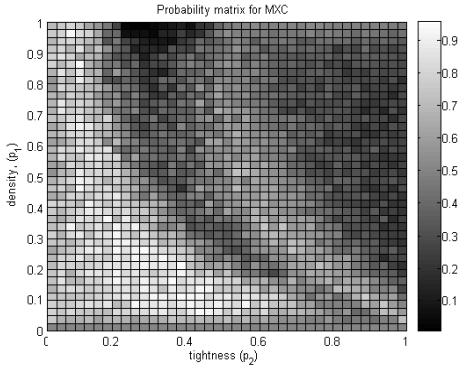

Fig. 7. Probability plot for the grid of instances in Set I.

For each node in the search, the density and tightness are calculated for the instance at hand and the variable ordering heuristic to apply could be selected in two different ways by using the information contained in this probability matrix. This gives place to two different probabilistic hyper-heuristics:

- DHH. A deterministic hyper-heuristic that always applies the low level heuristic with the highest probability at the the point of the grid that matches the values of p_1 and p_2 for the instance at the search node. For those points where the probability of applying MXC is equal to the probability of applying SD, the selection is made randomly with 50% of chance for each heuristic.
- PHH. A probabilistic hyper-heuristic that randomly selects the low level heuristic based on the probability matrix at the the point of the grid that matches the values of p_1 and p_2 for the instance at the search node.

The main difference between these two hyper-heuristics is that while DHH always selects the heuristic with the highest probability of achieve the best result, PHH contains a random decision. In average, PHH should behave as DHH, because it should select the most of the times the heuristic with the highest probability, but for specific instances, the results can be quite different.

These three simple hyper-heuristics were used to solve unseen instances in the region $p_1 = p_2$ corresponding to Set III and compared against MXC and SD on these instances. Once again, for each point $p_1 = p_2$ we have 50 instances. The results are presented in Figure 8. As we can see in Figure 8, this plot also includes the changes in the best solution method but now it has more changing points because we have more methods involved. From this picture it is possible to identify some interesting behaviours in the hyper-heuristics:

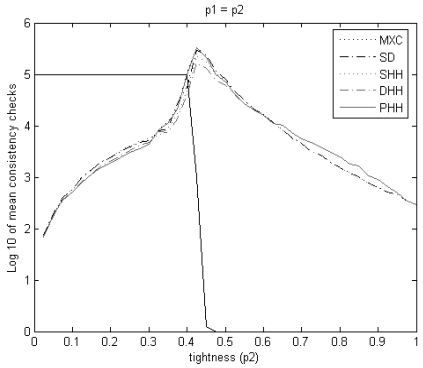

Fig. 8. Slice in the space where $p_1 = p_2$ for the instances in Set III.

Because we want to have a closer look at the behaviour of the proposed hyper-heuristics, we have divided the original plot shown in Figure 8 into three pictures, each one containing a specific region of the space. We consider that these three regions can give us a more clear picture about the performance of the hyper-heuristics.

Figure 9 shows the results obtained in the range $[0.25, 0.35]$. As we can see from the picture, PHH tends to behave as MXC and achieve the best results in the small values of p_2 but both methods do not perform well for higher values. The opposite happens with SHH, which starts doing more consistency checks for small values of p_2, and as long as we increase the value of p_2, the results improve. DHH seems to perform as an average of the other methods and is very reliable for this particular region of the space.

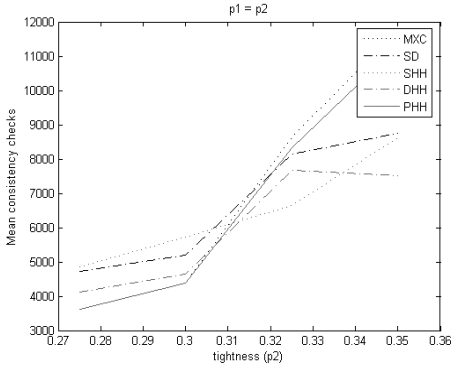

Fig. 9. Comparison of MXC, SD and the three hyper-heuristics in the range $[0.25, 0.35]$.

As we get close to the phase transition, the peaks in the mean consistency checks are higher. In Figure 10 we can observe the results obtained in the range $[0.35, 0.50]$, which contains the phase transition. It is clear that for this region of the space, the best option is DHH. It is also remarkable, that

SHH also performs really good for these instances because it is able to beat the best result from both MXC and SD in this region. PHH is just a little bit better than MXC, but their means run very close to each other along the space.

In any way, we observe that in Figure 10, around the phase transition, the hyper-heuristics can reduce the mean consistency checks by up to a factor of two.This is possible at all because the choices are made dynamically within the search tree rather than statically picking the same heuristic for all nodes. However, it is a good sign that even these simple hyper-heuristic methods have successfully exploited this possibility.

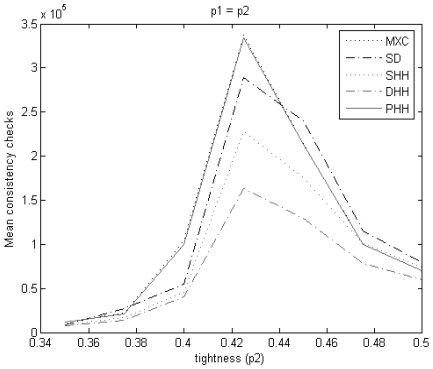

Fig. 10. Comparison of MXC, SD and the three hyper-heuristics in the range $[0.35, 0.50]$.

After the transition phase, in the region where almost all the instances are unsolvable, we observe a change in the behaviour of the hyper-heuristics. They all tend to behave as the low level heuristics. Figure 11 shows the results obtained in the range $[0.50, 0.70]$, and from this picture we can see that even when DHH is the best method because it achieves the less mean consistency checks, it starts performing as MXC once the value of $p_2 = 0.575$ is reached. Form that point on, MXC, PHH and DHH have the same performance. Something similar happens with SD and SHH, and both methods start achieving the same results also after $p_2 = 0.57$. For $p_2 > 0.625$, the best option is always to use SD or SHH, they both perform equally well on those instances. MXC, DHH and PHH are not a good option for instances with very large values of p_1 and p_2.

We can see that the three hyper-heuristics have a different performance depending on the region of the space. This is something that we expected because none of these heuristics is able to adapt during the search. The approaches used until now, only use the information form the runs in the training set and once the hyper-heuristic is formed is used without modifying the values in the decision matrix. It is also important to note that our original idea about SHH, where we supposed that a static 0/1 decision matrix would be not as good as a probabilistic approach is not completely right. SHH seems to be better than PHH for $p_2 > 0.31$ (and in some small regions is even better than DHH). The other probabilistic hyper-heuristic, DHH, seems to be very good

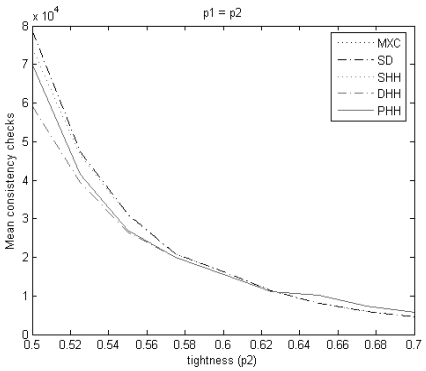

Fig. 11. Comparison of MXC, SD and the three hyper-heuristics in the range $[0.50, 0.70]$.

and beat SHH and the other methods in most of the space, but it also presents problems when dealing with high values of p_2.

VI. Conclusion

We have shown that it is possible to map a point in the CSP space defined by constraint $density \times tightness$ ($p_1 \times p_2$) to a variable ordering heuristic, namely MXC or SD. The idea that the low level heuristics have a different performance based on the features of the instances of a given problem provide us the opportunity of designing a more general approach, like hyper-heuristics. In this work, we developed three constructive hyper-heuristics based on the information obtained while solving some instances using each individual low level heuristic separately. This information derived from a set of training problem instances was used later within the hyper-heuristics in such a way that which heuristic to apply is decided according to their individual performance. When used to solve unseen instances, all three offline learning hyper-heuristics generate acceptable performances. Dynamically deciding which heuristic to select at a node in the search tree based on constraint density and tightness generates an improved performance in terms of mean consistency checks for some problem instances. The described hyper-heuristics for constraint satisfaction successfully combine the strengths of each low level construction heuristic, while avoiding their weaknesses.

As a future work, we are interested in applying these techniques to instances from CSP libraries and to real instances. We also aim to generate an adaptive hyper-heuristic which updates the values in the probabilities matrix during the search process dynamically. This way, the hyper-heuristic approach will be employing an online learning mechanism and it is expected to be more general, since it will be able to adapt to the new features of the instances at hand. It is also important to consider more variable ordering heuristics to observe the effects in the distribution of heuristics and the probability matrix. It will be also interesting to investigate whether adding more heuristics can improve the overall performance of the system or not. It might be even possible

to group similar heuristics or combine them to generate new heuristics.

REFERENCES

[1] BATTITI, R. Reactive search: Toward self-tuning heuristics. In *Rayward-Smith VJ, Osman IH, Reeves CR, Smith GD (eds) Modern Heuristic Search Methods* (1996), John Wiley & Sons Ltd., Chichester, pp. 61-83.

[2] BATTITI, R., BRUNATO, M. *Reactive search: Machine learning for memory-based heuristics*. In: *Gonzalez TF (ed) Approximation Algorithms and Metaheuristics, Taylor and Francis Books (CRC Press), Washington, DC* (2007) chap 21, pp. 1-17.

[3] BILGIN, B., ÖZCAN, E., KORKMAZ, E. *An experimental study on hyper-heuristics and exam scheduling*. Selected papers from the International Conference on Practice and Theory of Automated Timetabling 2006, Lecture Notes in Computer Science, vol. 3867, pp. 85-104, 2007.

[4] BURKE, E., KENDALL, G., O'BRIEN, R., REDRUP, D., AND SOUBEIGA, E. An ant algorithm hyper-heuristic. In *Proceedings of the fifth Metaheuristics International Conference (MIC'03)* (Kyoto, Japan, August 25-28 2003), vol. 10, pp. 1–10.

[5] BURKE, E., HART, E., KENDALL, G., NEWALL, J., ROSS, P., AND SHULENBURG, S. Hyper-heuristics: an emerging direction in modern research technology. In *Handbook of metaheuristics* (2003), Kluwer Academic Publishers, pp. 457–474.

[6] BURKE, E., HYDE, M., KENDALL, G., OCHOA, G., ÖZCAN, E., AND WOODWARD, J. R. Exploring hyper-heuristic methodologies with genetic programming. In *C. L. Mumford and L. C. Jain (Eds.), Computational intelligence: Collaboration, fusion and emergence* (2009) pp. 177–201.

[7] BURKE, E., HYDE, M., KENDALL, G., OCHOA, G., ÖZCAN, E., AND RONG, Q. (2009b). A survey of hyper-heuristics. *Computer Science Tech. Rep. No. NOTTCS-TR-SUB-0906241418-2747.* (2009) Nottingham, UK: University of Nottingham.

[8] BURKE, E., HYDE, M., KENDALL, G., OCHOA, G., ÖZCAN, E., AND WOODWARD, J. R. (2009c). A classification of hyper-heuristic approaches. *Computer Science Tech. Rep. No. NOTTCS-TR-SUB-0907061259-5808).* (2009) Nottingham, UK: University of Nottingham.

[9] BRELAZ, D. New methods to colour the vertices of a graph. *Comnications of the ACM 22* (1979).

[10] CHAKHLEVITCH, K., AND COWLING, PI. Hyperheuristics: Recent developments. In *Cotta C, Sevaux M, Sorensen K (eds) Adaptive and Multilevel Metaheuristics, Studies in Computational Intelligence* (2008), vol 136, Springer, pp. 3–29.

[11] CHEESEMAN, P., KANEFSKY, B., AND TAYLOR, W. M. Where the really hard problems are. In *Proceedings of IJCAI-91* (1991), pp. 331–337.

[12] CRAWFORD, J. M., AND AUTON, L. D. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence* (1993), pp. 21–27.

[13] CROWSTON, W.B., GLOVER, F., THOMPSON, G.L., AND TRAWICK, J.D. Probabilistic and parametric learning combinations of local job shop scheduling rules. *ONR Research memorandum, GSIA* (1963) Carnegie Mellon University, Pittsburgh -(117)

[14] DECHTER, R. Constraint networks. In *Encyclopedia of Artificial Intelligence* (1992), Wiley, pp. 276–286.

[15] DENZINGER, J., FUCHS, M., AND FUCHS, M. High performance ATP systems by combining several ai methods. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 97)* (1997) pp. 102–107.

[16] DIMOPOULOS, C., AND ZALZALA, A.M.S. Investigating the use of genetic programming for a classic one-machine scheduling problem. *Advances in Engineering Software* (2001) 32(6):489-498

[17] FISHER, H., AND THOMPSON, G.L. Probabilistic learning combinations of local job-shop scheduling rules. In *Factory Scheduling Conference* (1961) Carnegie Institue of Technology

[18] FREUDER, E. C., AND MACKWORTH, A. K. *Constraint-Based Reasoning*. MIT/Elsevier, Cambridge, 1994.

[19] FUKUNAGA, A.S. (2002) *Automated discovery of composite sat variable-selection heuristics.* In *Eighteenth national conference on Artificial intelligence* (Edmonton, Alberta, Canada, 2002) pp. 641–648

[20] FUKUNAGA, A.S. *Evolving local search heuristics for SAT using genetic programming.* In *et al KD (ed) LNCS 3103. Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO '04)* (Springer-Verlag, Seattle, WA, USA, 2008) pp. 483–494

[21] FUKUNAGA, A.S. *Automated discovery of local search heuristics for satisfiability testing. Evolutionary Computation (MIT Press)* (2008) 16(1):31-1

[22] GAGLIOLO, M., AND SCHMIDHUBER, J. *Dynamic algorithm portfolios*. In *Proceedings AI and MATH 06, Ninth International Symposium on Artificial Intelligence and Math- ematics* (Fort Lauderdale, Florida, 2006).

[23] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[24] GENT, I., MACINTYRE, E., PROSSER, P., SMITH, B., AND T.WALSH. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP-96* (1996), pp. 179–193.

[25] HARALICK, R. M., AND ELLIOTT, G. L. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14* (1980), 263–313.

[26] HUBERMAN, B.A., LUKOSE, R.M., AND HOGG, T. *An economics approach to hard computational problems*. *Science* (1997) 275:5154

[27] KRASNOGOR, N., AND SMITH, J.E. *Emergence of profitable search strategies based on a simple inheritance mechanism*. In *Proceedings of the 2001 Genetic and Evolutionary Computation Conference, Morgan Kaufmann* (2001)

[28] LATORRE, ANTONIO, PEÑA, JOSÉ M., ROBLES, VICTOR, MUELAS, SANTIAGO Using multiple offspring sampling to guide genetic algorithms to solve permutation problems In *GECCO'08: Proceedings of the 10th annual conference on Genetic and evolutionary computation* (2008), ACM.

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

[30] MINTON, S., PHILLIPS, A., AND LAIRD, P. Solving large-scale csp and scheduling problems using a heuristic repair method. In *Proceedings of the 8th AAAI Conference* (1990), pp. 17–24.

[31] MINTON, S., JOHNSTON, M. D., PHILLIPS, A., AND LAIRD, P. Minimizing conflicts: A heuristic repair method for csp and scheduling problems. *Artificial Intellgence 58* (1992), 161–205.

[32] MONTANARI, U. Networks of constraints: fundamentals properties and applications to picture processing. *Inf. Sci. 7* (1974), 95–132.

[33] NAREYEK, A. Choosing search heuristics by non-stationary reinforcement learning. In *Resende MGC, de Sousa JP (eds) Metaheuristics: Computer Decision-Making* (2003) Kluwer, chap 9, pp. 523-544.

[34] ONG, Y.S., LIM, M.H., ZHU, N., AND WONG, K.W. *Classification of adaptive memetic algorithms: a comparative study*. *IEEE Transactions on Systems, Man, and Cybernetics, Part B* (2006) 36(1):141-152

[35] ORTIZ-BAYLISS, J. C., TERASHIMA-MARÍN, H., , ROSS, P., ROSADO, J. I. F., AND VALENZUELA-RENDÓN, M. A neuro evolutionary approach to produce general hyper-heuristics for the dynamic variable ordering in hard binary constraint satisfaction problems. In *GECCO'09: Proceedings of the 11th annual conference on Genetic and evolutionary computation* (2009), ACM.

[36] ÖZCAN, E., BILGIN, B., AND KORKMAZ, E. A comprehensive analysis of hyper-heuristics. *Intelligent Data Analysis 12*(2008), 3–23.

[37] PISINGER, D., ROPKE, S. A general heuristic for vehicle routing problems. *Computers and Operations Research* (2007) 34:2403-2435

[38] PROSSER, P. Binary constraint satisfaction problems: Some are harder than others. In *Proceedings of the European Conference in Artificial Intelligence* (Amsterdam, Holland, 1994), pp. 95–99.

[39] PROSSER, P. An empirical study of phase transitions in binary constraint satisfaction problems. Tech. Rep. Report AISL-49-94, University of Strathclyde, 1994.

[40] ROSS, P. *Hyper-heuristics*. In *Burke EK, Kendall G (eds) Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, (2005), Springer, chap 17, pp. 529–556.

[41] RUSSELL, S., AND NORVIG, P. *Artificial Intelligence A Modern Approach*. Prentice Hall, 1995.

[42] SMITH, B. M. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence 81* (1996), 155–181.

[43] TERASHIMA-MARÍN, H., ORTIZ-BAYLISS, J. C., ROSS, P., AND VALENZUELA-RENDÓN, M. Hyper-heuristics for the dynamic variable ordering in constraint satisfaction problems. In *GECCO'08: Proceedings of the 10th annual conference on Genetic and evolutionary computation* (2008), ACM.

[44] WILLIAMS, C. P., AND HOGG, T. Using deep structure to locate hard problems. In *Proc. of AAAI-92* (San Jose, CA, 1992), pp. 472–477.