Variable and Value Ordering Decision Matrix Hyper-heuristics: a Local Improvement Approach

José Carlos Ortiz-Bayliss[1], Hugo Terashima-Marín[1], Ender Özcan[2], Andrew J. Parkes[2], and Santiago Enrique Conant-Pablos[1]

[1] Tecnológico de Monterrey, Campus Monterrey
Monterrey, Mexico, 64849
{jcobayliss@gmail.com, terashima@itesm.mx, sconant@itesm.mx}
[2] University of Nottingham, Jubilee Campus
Nottingham, United Kingdom, NG8 1BB
{ender.ozcan@nottingham.ac.uk, ajp@cs.nott.ac.uk}

Abstract. Constraint Satisfaction Problems (CSP) represent an important topic of study because of their many applications in different areas of artificial intelligence and operational research. When solving a CSP, the order in which the variables are selected to be instantiated and the order of the corresponding values to be tried affect the complexity of the search. Hyper-heuristics are flexible methods that provide generality when solving different problems and, within CSP, they can be used to determine the next variable and value to try. They select from a set of low-level heuristics and decide which one to apply at each decision point according to the problem state. This study explores a hyper-heuristic model for variable and value ordering within CSP based on a decision matrix hyper-heuristic that is constructed by going into a local improvement method that changes small portions of the matrix. The results suggest that the approach is able to combine the strengths of different low-level heuristics to perform well on a wide range of instances and compensate for their weaknesses on specific instances.

Keywords: Constraint Satisfaction, Hyper-heuristics, Variable and Value Ordering

1 Introduction

A Constraint Satisfaction Problem (CSP) is defined as follows: given a set of n variables with its respective domain of possible values m and a set of constraints, each one involving a subset of variables, find all possible n-tuples such that each n-tuple is an instantiation of the n variables satisfying all the constraints. For this research we have only considered those CSP in which the domains are discrete, finite sets and the constraints involve only one or two variables (binary constraints). Various studies have been developed to randomly generate instances of binary CSP (see for example [15]), and those studies have shown that random binary CSP have very interesting properties which make them an important topic of study. The relevance of studying CSP lies on the fact that they are an important technique used to find a solution for many artificial intelligence problems [11].

Binary CSP present two important properties that are used in this research: the constraint density (p_1) and the constraint tightness (p_2). The constraint density is a measure of the proportion of constraints within the instance; the closer the value of p_1 to 1, the larger the number of constraints in the instance. The constraint tightness (p_2) represents a proportion of the conflicts within the constraints. A conflict is a pair of values $\langle x, y \rangle$ that is not allowed for two variables at the same time. The higher the number of conflicts, the more unlikely an instance has a solution.

Stated as a classic search problem, a CSP is usually solved using a Depth First Search (DFS) where every variable represents a node in the tree. Every time a variable is instantiated, the constraints in which that variable is involved must be checked to verify that none of them is violated. When an assignation violates one or more constraints, the instantiation must be undone, and another value must be considered for that variable. If there are not any values available, the value of the previous instantiated variable must be changed. This technique is known as backtracking. There are some improvements to this basic search method which try to reduce the number of revisions of the constraints (consistency checks) like constraint propagation and backjumping. With constraint propagation the idea is to propagate the effect of one instantiation to the rest of the variables due to the constraints among them. 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. Backjumping is another powerful technique for retracting and modifying the value of a previously instantiated variable but is different to backtracking because backjumping can go back more than one level at the time when a backward movement is needed.

The selection of the next variable to instantiate and its respective value affects the search complexity and represents an opportunity to optimize the search. Every CSP contains characteristics that could make it more suitable to a certain heuristic. If these characteristics can be identified, then the problems would be solved more efficiently. Some heuristics for variable and value ordering exist, but none of them has been able to behave well for all instances. As we can observe, the selection of the right heuristic for the current instance is not trivial.

Hyper-heuristics are a modern approach to take advantage of the selective application of the low-level heuristics based on the current problem features. Even though the idea of combining multiple heuristics goes back to 1960s [8] the term hyper-heuristic was first introduced by Denzinger et al. [7] in 1997. Surveys on hyper-heuristic methodologies can be found in [5] and [3]. Hyper-heuristics can be divided into two main classes: those which select from existing heuristics and the ones that generate new heuristics. A more detailed description about the classification of hyper-heuristics can be found in [5] and [4]. As representative studies on the hyper-heuristic methodologies that generate new heuristics, we can cite Fukunaga, who uses genetic programming as an automated heuristic discovery system for the SAT problem [9]. Conversely, most of the hyper-heuristics that select from existing 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. One of the first attempts to systematically map CSP to algorithms and heuristics according to the features of the problems was presented by Tsang and Kwan [18], in 1993. In that study, the authors presented a survey of algorithms and heuristics for solving CSP and they proposed a relation between the formulation of the CSP and the most adequate solving method for that formulation. More recently, Ortiz-Bayliss et al. [13] developed a study about heuristics for variable ordering within CSP 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 state of the problem. More studies about hyper-heuristics applied to CSP include the work done by Terashima-Marín et al. [17], who proposed an evolutionary framework to generate hyper-heuristics for variable ordering in CSP; and the research developed by Bittle and Fox [1] who presented a hyper-heuristic approach for variable and value ordering for CSP based on a symbolic cognitive architecture augmented with case based reasoning as the machine learning mechanism for their hyper-heuristics. Ortiz-Bayliss et al [14] recently presented a study where they represent variable ordering hyper-heuristics as integer matrices and a genetic algorithm is used to evolve the structure of the matrices in order to generate hyper-heuristics.

Our model produces hyper-heuristics represented as matrices of integers by going through a local improvement process. Each matrix represents a rule for the application of the ordering heuristics based on the values of p_1 and p_2 of the instance at hand.

This paper is organized as follows: Section 2 describes in detail the solution model developed for this research. The experiments and main results are shown in Section 3. Finally, Section 4 presents the conclusions and future work.

2 Solution Approach

In this section we present the solution model proposed in this research. We also include a brief description of the variable and value ordering heuristics and the CSP instances used in this investigation. Finally, the decision matrix hyper-heuristic and the local improvement process are described.

2.1 Variable and Value Ordering

Many researchers have proved the importance of the order of the variables and its impact in the cost of the solution search [15]. The search space grows exponentially with respect to the number of variables, and so does the time for finding the optimal ordering. Once a variable has been selected to be instantiated we need to decide which value, among all the feasible ones, will be used for that instantiation. This ordering also has relevance to the search, because it also affects the complexity of the search.

Various heuristic and approximate approaches have been proposed that find good solutions for some instances of the problem. However, it has not been possible to find a reliable method to solve well all instances of CSP. In this study, we have included two variable ordering heuristics and two value ordering heuristics. Max-Conflicts (MXC) and Saturation Degree (SD) [2] are the variable ordering heuristics and we have included Min-Conflicts (MINC) [12] and Max-Conflicts (MXC) as value ordering heuristics.

A solution for any given CSP is constructed selecting one variable at a time based on one of the two variable ordering heuristics used in this investigation: MXC and SD. Each one of these heuristics reorders the variables to be instantiated dynamically at each step during the construction process. Later, a value must be selected and assigned to the chosen variable considering the constraints and using MINC or MXC as value ordering heuristic depending on the instance features. A CSP solver that makes use of constraint propagation and backjumping was implemented for developing this research. The ordering heuristics used in this investigation are briefly explained in the following lines.

MXC is a very simple and fast heuristic, and the main idea is to select the variable which values are involved in the larger number of conflicts among the constraints in the instance. MXC can be used both for variable ordering and for value ordering. When used for variable ordering, the instantiation will produce a subproblem that minimises the number of conflicts among the variables left to instantiate. When applied to value ordering, the principle of selecting the 'most conflictive' value remains, but now the heuristic only chooses values from the selected variable. The other variables are not considered because they have not been selected for instantiation.

SD 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. The saturation degree of a node X_i is the number of different colours to which X_i is adjacent [2]. In graph colouring, one implicit constraint is that two adjacent nodes cannot be assigned the same colour. Thus, all connected instantiated nodes must have different colors. In our research, we define the saturation degree of a node X_i as the number of adjacent nodes to X_i that have already been instantiated. In this way, our implementation of SD takes advantage of the topology of the constraint graph to select the most restricted variable given the progress in the search.

MINC is one simple and commonly used value ordering heuristic in CSP. When using MINC, the heuristic prefers the value (from the selected variable) that is involved in the minimum number of conflicts [12]. MINC will try to leave the maximum flexibility for subsequent variable assignments.

By combining the variable and value ordering heuristics, we obtain four combinations: MXC/MINC, SD/MINC, MXC/MXC and SD/MXC. From this point on we will refer to each one of these combinations as ordering heuristics. In this way, each heuristic describes one way to order the variables and, at the same time, a way to order the values from the variables.

2.2 CSP Instances and The Problem State Representation

The binary CSP instances used for the experiments in this research 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 [16]. 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. 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). Every time a variable is assigned a new value and the infeasible values are removed from domains of the remaining uninstantiated variables, the values of p_1 and p_2 change and a sub-problem with new features appears. This is the reason why we decided to use the constraint density and tightness to represent the problem state and guide the selection of the low-level heuristics. Our idea is that these two features can be used to describe a CSP instance and to create a relation between instances and heuristics.

There is a relation between the structure of CSP and the difficulty of solving them with search algorithms [6]. Specifically, the median search cost of search algorithms for CSP exhibits a sharp peak as the values of p_1 and p_2 change. This peak coincides with the transition from under-constrained to over-constrained instances (region known as transition phase), often manifested as an abrupt change in the probability that an instance has a solution. Inside this region, the most difficult soluble problems and the most difficult insoluble problems co-exist [16].

The collection of instances used for training includes 130 different hard instances, divided into two sets: 30 for Set A and 100 for Set B. Both sets A and B are composed by distinct instances generated with the parameters $(n = 20, m = 10, p_1 = 0.6, p_2 = 0.33)$, which correspond to hard instances [6]. An additional Set C is used only for testing purposes. For Set C we generated 405 instances, with $n = 20$, $m = 10$, $p_1 = 0.6$ and p_2 in the range $[0, 1]$ with increments of 0.0125. The complete collection of instances used in this research includes both easy instances (outside the transition phase) and hard instances (inside the transition phase).

2.3 The Decision Matrix Hyper-heuristic and The Local Improvement Approach

The decision matrix hyper-heuristic is formed by a matrix of integers, where the rows are used to represent the constraint density (p_1) and the columns the constraint tightness (p_2). There are other features that can be used to describe a CSP instance (see for example: k and ρ in [10]) but p_1 and p_2 have been usually applied to for this purpose. The value of each element represents the heuristic to apply when the instance features correspond to the values of p_1 and p_2 coded in the axes. Each cell of the matrix corresponds to one decision point, the points where the hyper-heuristic decides which heuristic to apply to continue the search. Each cell contains a value from 0 to 3, which is used to specify the ordering heuristic to be used when the instance features match the ones at that decision point. Because the matrix presents increments of 0.10 on both p_1 and p_2 axes (and these parameters lie in the range $(0, 1]$), the resulting matrices contain 10×10 cells. We tried different resolutions and sizes for the matrices and we observed that increasing the resolution of the matrix does not necessary provide a better performance. After preliminary studies we found that the best size of the matrices (in the number of cells per rows and columns) lies in the range $[10, 20]$. We observed that small decision matrices do not contain enough decision points to produce a significant change of heuristics during the search. Thus, the hyper-heuristics obtained are very monotonous. Conversely, than 20 rows and columns produced over-detailed matrices that required larger amounts of time during the improvement process. The results produced by this over-detailed matrices did not show to be worth the additional improvement time. When the hyper-heuristic is created, all the cells are initialized to -1, meaning that the cell has never been used as decision point. When the hyper-heuristic is used to solve an instance, the decision points used during the search are initialized. There are two ways to initialize the decision points: (1) selecting one random ordering heuristic or (2) using a default ordering heuristic. For this research we decided to use the second way to initialize the matrices.

We will try to clarify how the decision matrix hyper-heuristic is used with an example. Imagine we have a hyper-heuristic coded in a 4×4 matrix with increments of 0.25 in p_1 and p_2. This matrix has the next values on each axis: 0.25, 0.5, 0.75 and 1.0 (the values are uniformly distributed in the range $(0, 1]$). In this example, we assume that the decision matrix has already been updated via the improvement process. Thus, the values of the cells that were used during the training have already been assigned one of the four possible ordering heuristics. The points that were not visited during the training process do not have one heuristic assigned. In case an unseen instance is presented to the hyper-heuristic and it forces the matrix to visit one of these unassigned cells, the value for that decision point will be determined using the default ordering heuristic. Figure 1 shows the hyper-heuristic discussed. Suppose that an instance P is presented to this hyper-heuristic and the features of instance P are $p_1 = 0.8$ and $p_2 = 0.70$ (during the search, p_1 is estimated as the number of non empty edges in the constraint graph over the maximum possible number of edges and p_2 is obtained by the average of the tightness of all the constraints). In that case, the cell to be accessed in the decision matrix would be $(2, 2)$, which corresponds to the values $p_1 = 0.75$ and $p_2 = 0.75$, due to the resolution of the matrix. In this example, heuristic 0 would be selected as ordering heuristic. If we change the resolution of the matrix, the number of cells will change along with the values of the constraint density and tightness coded in the axes. Once the ordering heuristic has been used, the original instance P is transformed into P', which values of p_1 and p_2 may be different from P. If we continue applying the hyper-heuristic, we will be moving through the decision matrix. Every time we get closer to the solution, we also get closer to the origin of the decision matrix. When we finally find a solution to the instance, the cells visited in the matrix allow us to map the search tree for that specific instance.

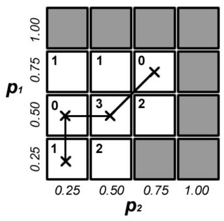

Fig. 1. An example hyper-heuristic.

The local improvement approach uses a very simple method to upgrade the matrices and improve the performance of the hyper-heuristics. The method is based on the idea of changing the decision matrix, one cell at the time, and accepting the change only if it improves the performance of the hyper-heuristic on a set of instances (training set). Basically, we are implementing a hill climbing search on the space of heuristics. Every time we change the ordering heuristic corresponding to one decision point, the rest of the search tree from that node may be different. Thus, new points will be activated in the decision matrix and their cells initialized. If we observe the search tree on the space $p_1 \times p_2$, it is clear that, as we get closer to the solution (the deeper nodes in the search tree), we find the smaller the values of p_1 and p_2. Based on this observation we concluded that the nodes of the tree which are located further from the origin are visited first than the closer ones. We need to keep a record of the number of changes per cell in the matrix, otherwise the improvement method will always change the same decision point. Thus, we proposed an order to change the decision points in the matrix: the further points from the origin that have not achieved the maximum number of changes have priority to be updated. Because the process is not randomized, if we run the process twice with the same parameters we will obtain the same results. The decision points to be updated depend only on the instances solved during the training and the cells visited because of the search. The local improvement process is described in the following lines:

1. Initialize the decision matrix hyper-heuristic.
2. Solve the instances in the training set with the hyper-heuristic and obtain the average consistency checks per instance ($avg_0(HH)$).
3. Update the decision matrix. Only one cell is changed according to the criteria already described.
4. Solve the instances in the training set with the updated hyper-heuristic and obtain the average consistency checks per instance ($avg(HH)$). If $avg(HH) < avg_0(HH)$, make $avg_0(HH) = avg(HH)$ and accept the change. Otherwise, cancel the change and return the decision matrix to the previous configuration.
5. Repeat from step 3 until the maximum number of cycles is reached.

3 Experiments and Results

In order to test the model proposed we developed three experiments. In the first experiment we focus on the generation of hyper-heuristics using a small hard training set (Set A) and later, we test those hyper-heuristics on a larger set of hard unseen instances (Set B) with similar properties than the instances inside the training set. The second experiment tries to obtain new hyper-heuristics using the larger set (Set B) and analysing the results to see whether the training on those specific instances can improve the results of the hyper-heuristics obtained in the first experiment. The third and final experiment tests the performance of the hyper-heuristics trained with Set B on unseen instances from Set C which includes hard and not so hard instances.

3.1 Experiment I

In this experiment we produced four hyper-heuristics, each one using a different default ordering heuristic. The size of the matrices was set to 10 and for each hyper-heuristic the local improvement process ran for 30 cycles. In this experiment, the instances from Set A were used as input for the process. Table 1 presents the average number of consistency checks that any of the four hyper-heuristics uses at the start of the process, at the end of the improvement process and the reduction achieved through the process with respect to the initial hyper-heuristic. One conclusion derived from Table 1 is that hyper-heuristics HH01-04 have the same performance on Set A in terms of consistency checks. It is important to mention that the decision matrices from these hyper-heuristics are not equal, but some decision points share the same ordering heuristics. This means that even with a different combination of heuristics we can achieve the same performance. Because each hyper-heuristic is using a different default ordering heuristic to initialize the decision matrix at the beginning of the process, it is clear to see that the average consistency checks that MXC/MINC requires to solve each instance in Set A is the the same that the average obtained by the hyper-heuristic that uses it as default heuristic at the beginning of the process, it is, 15834. Following the same reasoning, to solve each instance in Set A, SD/MINC, MXC/MXC and SD/MXC require 9709, 16032 and 9995 consistency checks, respectively. In all the cases the hyper-heuristics are able to overcome the average consistency checks required by the pure ordering heuristics.

Table 1. Performance of HH01-04 on Set A

HH	Default H	Avg. at start	Avg. at end	Reduction
HH01	MXC/MINC	15834	7947	50%
HH02	SD/MINC	9709	7947	19%
HH03	MXC/MXC	16032	7947	51%
HH04	SD/MXC	9995	7947	21%

Once we trained hyper-heuristics HH01-04, it is time to test whether they are able to perform well on a wider set of instances with similar features to those used during the training. We used HH01-04 to solve Set B without any additional training. The results presented in Table 2 are still competitive, but not as good as in the previous test. Even though the hyper-heuristics are able to overcome the results of MXC/MINC and MXC/MXC, the hyper-heuristics are not able to beat SD/MINC and SD/MXC. This decrease in the performance is justified because the hyper-heuristics were not trained for those specific instances. Even though no additional training was performed, HH01-04 provide good results when compared to the simple ordering heuristics: the average consistency checks required by HH02 to solve each instance of Set B is 11.57% above the average of the best ordering heuristic (which in this case is SD/MINC with 7989 consistency checks). The results from this test are presented in Table 2.

Table 2. Performance of HH01-04 on Set B

H	Avg(Checks)	HH	Avg(Checks)
MXC/MINC	20308	HH01	8913
SD/MINC	7989	HH02	8914
MXC/MXC	19170	HH03	8913
SD/MXC	8553	HH04	8913

3.2 Experiment II

We have already argued that one possible cause of the decrease in the performance of HH01-04 when tested on Set B is the fact that they were not trained for such set. In an attempt to confirm this idea, we produced four new hyper-heuristics (HH05-08) as described in Experiment I, but this time we used Set B as training set during the process. The results shown in Table 3 confirm the idea that the difference in the performance of HH01-04 with respect to the two sets was not because of the model itself, but the training instances used during the process. Table 3 presents the results of hyper-heuristics HH05-08 on Set B. As we can observe, these hyper-heuristics provide better results than the previous HH01-04. The difference in the performance is notorious, and three of the four hyper-heuristics are able to perform better than each one of the simple ordering heuristics (even though in the case of SD/MINC the difference is not significant). HH07 does not provide results as good as the other hyper-heuristics but it is still better than any of the previous HH01-04.

Table 3. Performance of HH05-08 on Set B

HH	Default H	Avg. at start	Avg. at end	Reduction
HH05	MXC/MINC	20308	7959	61%
HH06	SD/MINC	7989	7959	1%
HH07	MXC/MXC	19170	8856	54%
HH08	SD/MXC	8553	7960	7%

3.3 Experiment III

In the previous experiments we trained and tested our hyper-heuristics using only hard instances. We also need to consider the performance of these hyper-heuristics on other instances with similar features, but not as difficult as the ones presented before. The instances from Set C were solved with HH05-08 and compared against the performance of the corresponding default ordering heuristic (Figure 2).

In Figure 2 we can observe that HH05 and HH08 clearly outperform their respective default ordering heuristics. Specially at the peak located at the transition phase, each of these two heuristics dominate the ordering heuristics. In the case of HH06 the performance is not very clear. Both the behaviour of HH06 and SD/MINC describe a very similar curve for values of $p_2 < 0.6$. For larger values of p_2, HH06 is dominated by SD/MINC. Finally, without any additional training, HH07 was not able to compete with MXC/MXC. This ordering heuristic looks very robust for the instances contained in Set C. It is important to recall that the instances in Set C were never seen before by the hyper-heuristics and also, they have features different to those used during the training.

4 Conclusions

The solution model presented in this paper produces good quality hyper-heuristics which are able to compete with the simple ordering heuristics on the different sets used in this investigation.

The hyper-heuristics produced for one specific set reduce their performance when applied to unseen instances with similar features but they are still competitive. Even though the hyper-heuristics presented achieved promising results

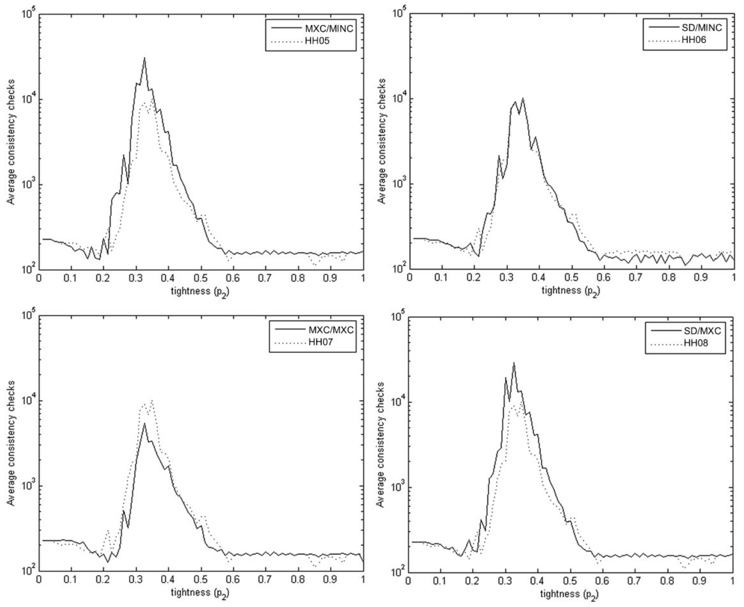

Fig. 2. Performance of HH05-08 and their corresponding default ordering heuristic on Set C.

we still need to work more in order to obtain better quality hyper-heuristics that can be applied to a wider range of instances without having to retrain them.

As future work we have considered the inclusion of new ordering heuristics and new ways to test the performance of the hyper-heuristics, not only the average consistency checks on the sets. We also know it is a common practice to use randomly generated instances in CSP studies but we are aware that it is not enough to test our approach. We want to test our hyper-heuristics on real instances taken from existing CSP repositories.

5 Acknowledgments

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

References

1. Bittle, S.A., Fox, M.S.: 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. pp. 2209–2212. GECCO '09, ACM, New York, NY, USA (2009)

2. Brelaz, D.: New methods to colour the vertices of a graph. Communications of the ACM 22 (1979)
3. Burke, E., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Rong, Q.: A survey of hyper-heuristics. Tech. Rep. NOTTCS-TR-SUB-0906241418-2747, School of Computer Science, University of Nottingham (2009)
4. Burke, E., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.R.: A classification of hyper-heuristic approaches. Tech. Rep. NOTTCS-TR-SUB-0907061259-5808, School of Computer Science, University of Nottingham (2009)
5. Chakhlevitch, K., Cowling, P.: Hyperheuristics: Recent developments. In: Cotta, C., Sevaux, M., Srensen, K. (eds.) Adaptive and Multilevel Metaheuristics, Studies in Computational Intelligence, vol. 136, pp. 3–29. Springer Berlin / Heidelberg (2008)
6. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: Proceedings of IJCAI-91. pp. 331–337 (1991)
7. Denzinger, J., Fuchs, M., Fuchs, M., Informatik, F.F., Munchen, T.: High performance atp systems by combining several ai methods. In: In Proc. Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 97. pp. 102–107. Morgan Kaufmann (1997)
8. Fisher, H., Thompson, G.L.: Probabilistic learning combinations of local job-shop scheduling rules. In: Factory Scheduling Conference. Carnegie Institute of Technology (1961)
9. Fukunaga, A.S.: Automated discovery of local search heuristics for satisfiability testing. Evolutionary Computation 16, 31–61 (March 2008)
10. Gent, I., MacIntyre, E., Prosser, P., Smith, B., T.Walsh.: An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In: Proceedings of CP-96. pp. 179–193 (1996)
11. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 8(1), 99–118 (1977)
12. Minton, S., Johnston, M.D., Phillips, A., Laird, P.: Minimizing conflicts: A heuristic repair method for csp and scheduling problems. Artificial Intellgence 58, 161–205 (1992)
13. Ortiz-Bayliss, J.C., Terashima-Marín, H., Özcan, E., Parkes, A.J.: Mapping the performance of heuristics for constraint satisfaction. In: IEEE Congress on Evolutionary Computation (CEC). pp. 1–8 (july 2010)
14. Ortiz-Bayliss, J.C., Terashima-Marín, H., Özcan, E., Parkes, A.J.: On the idea of evolving decision matrix hyper-heuristics for solving constraint satisfaction problems. In: Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. pp. 255–256. GECCO '11, ACM, New York, NY, USA (2011)
15. Prosser, P.: Binary constraint satisfaction problems: Some are harder than others. In: Proceedings of the European Conference in Artificial Intelligence. pp. 95–99. Amsterdam, Holland (1994)
16. Smith, B.M.: Locating the phase transition in binary constraint satisfaction problems. Artificial Intelligence 81, 155–181 (1996)
17. Terashima-Marín, H., Ortiz-Bayliss, J.C., Ross, P., 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. ACM (2008)
18. Tsang, E., Kwan, A.: Mapping constraint satisfaction problems to algorithms and heuristics. Tech. Rep. CSM-198, Department of Computer Sciences, University of Essex (1993)