Lifelong Learning Selection Hyper-heuristics for Constraint Satisfaction Problems

José Carlos Ortiz-Bayliss(✉), Hugo Terashima-Marín, and Santiago Enrique Conant-Pablos

Tecnológico de Monterrey National School of Engineering and Sciences, Av. Eugenio Garza Sada 2501 Sur Col. Tecnológico, 64849 Monterrey, Nuevo Leon, Mexico
{jcobayliss,terashima,sconant}@itesm.mx

Abstract. Selection hyper-heuristics are methods that manage the use of different heuristics and recommend one of them that is suitable for the current problem space under exploration. In this paper we describe a hyper-heuristic model for constraint satisfaction that is inspired in the idea of a lifelong learning process that allows the hyper-heuristic to continually improve the quality of its decisions by incorporating information from every instance it solves. The learning takes place in a transparent way because the learning process is executed in parallel in a different thread than the one that deals with the user's requests. We tested the model on various constraint satisfaction problem instances and obtained promising results, specially when tested on unseen instances from different classes.

Keywords: Heuristics · Selection hyper-heuristics · Constraint satisfaction · Lifelong learning

1 Introduction

A constraint satisfaction problem (CSP) contains a set of variables V, each with a domain D_v of possible values and a set of constraints C that restricts the values that can be assigned to those variables. When using backtracking-based algorithms to solve CSPs [1], the ordering in which the variables are considered for instantiation affects the way the solution space is explored and also, the cost of the search. If this ordering is poor, the risk of taking the search into long unpromising branches increases and, as a consequence, the time required to find one solution also increases. Then, bad choices in the ordering in which the variables are instantiated represent a huge amount of unnecessary work. Various heuristics have been proposed to tackle the variable ordering problem in CSPs and they have proven to be efficient for specific classes of instances, but usually to fail when tested on distinct classes. As a consequence, applying the same heuristic to every instance rarely produces good results. This drawback in the performance of such methods arises mainly from the vast range of parameters or algorithm

G. Sidorov and S.N. Galicia-Haro (Eds.): MICAI 2015, Part I, LNAI 9413, pp. 190–201, 2015.
DOI: 10.1007/978-3-319-27060-9_15

choices involved, and the lack of guidance on how to properly tune them when the problems change. Also, the understanding of how heuristics work on different situations is not fully understood yet, making it difficult to decide, only based on what we currently know about them, which one is the best option for a certain instance. For these reasons it seems reasonable to rely on an automatic method to produce the mapping between features and heuristics.

The idea of selecting from a set of algorithms the most suitable one to solve one specific problem is usually referred to as the algorithm selection problem [18]. This problem has been addressed in the literature by using different strategies, but two of the most used terms include algorithm portfolios and selection hyper-heuristics. Algorithm portfolios attempt to allocate a period for running a chosen algorithm from a set of algorithms in a time-sharing environment [8,12], while selection hyper-heuristics [5,19] are high-level methodologies that select among different heuristics given the properties of the instance at hand.

This paper is organized as follows. In Sect. 2 we present related works on hyper-heuristics for CSPs. Section 3 describes the features used to characterize the instances and the ordering heuristics considered for this investigation. The main contribution of this investigation, the lifelong learning selection hyper-heuristic model for CSP, is described in Sect. 4. Section 5 presents the experiments conducted, their analysis and the discussion of the results. Finally, in Sect. 6 we present our conclusion and future work.

2 Related Work

Regarding algorithm portfolios for CSPs, the work conducted by O'Mahony et al. [15] collects a set of solvers and decides which solver is the most suitable one according to the features of the instance to solve. Their approach aims at solving the instances as well as the best possible solver from the set does. More recent studies on the combination of heuristics for CSPs include the work done by Petrovic and Epstein [17], who studied the idea of combining various heuristics to produce mixtures that work well on particular classes of CSP instances. Their approach bases their decisions on random sets of what they call advisers, which are basically the criteria used by the variable and value ordering heuristics to make their decisions. The advisers are weighted according to their previous decisions: good decisions increase their weights, while bad ones decrease them. This approach has proven to be able to adapt to a wide range of instances but it requires to define the size of the sets of advisers. There is a trade-off that must be considered: the larger the set of advisers, the larger the amount of computational resources required to evaluate the criteria of the different advisers but the fewer the number of instances to train the system.

With regard to selection hyper-heuristics, Bittle and Fox [2] worked on a symbolic cognitive architecture to produce variable ordering hyper-heuristics for map colouring and job shop scheduling problems represented as CSPs. Their approach produces small 'chunks' of code that serve as the components of rules for variable ordering. As a result, hyper-heuristics composed by a large set of rules

operate to solve the instances by selectively applying the most suitable heuristic for the instance being solved. The last two years include some important developments regarding hyper-heuristics for CSPs. Autonomous search was applied to replace bad performing strategies by more promising ones during the search, producing competent variable ordering strategies for CSPs [23]. The authors used a choice function that evaluates some indicators of the search progress and dynamically ranks the ordering heuristics according to their quality to exclude those that exhibit a low performance. Ortiz-Bayliss et al. [16] proposed a learning vector quantization framework to tackle the dynamic selection of heuristics on different sets of CSP instances. Although their approach proved to be useful for the instances where it was tested, the model requires the expertise of the user for tuning the parameters in the framework in order to produce reliable hyper-heuristics.

An area of opportunity regarding all the hyper-heuristic strategies proposed in the past for CSP involves the learning approach. Most of the hyper-heuristic methods previously proposed for CSP require a training phase and, only after the training process is over, the hyper-heuristic can be used to solve as many instances as wished. Once the hyper-heuristic is trained, no further learning is done and then, the hyper-heuristics are unable to incorporate additional information to improve their decisions. The main limitation derived from this situation is that such hyper-heuristics usually fail to generalize well on instances from unseen classes of instances.

Lifelong learning [20,21] was recently introduced as an alternative learning approach that responds to the constant changes in the nature of the instances being solved and the available solvers. As Silver suggests [21], systems that use lifelong learning have the ability to learn, retain and use knowledge over a life time. Lifelong learning is more than just extending the learning phase or executing it various times: it requires the use of a suitable knowledge representation that allows the system to modify only small parts of what it knows to improve its further performance. The concept of lifelong learning was recently introduced to the field of hyper-heuristics, specifically for the Bin Packing problem with exceptional results [11,13,22]. The lifelong learning mechanism in those models was implemented by using an artificial immune system [6].

3 Problem Characterization and Ordering Heuristics

In this section we describe the features used to characterize the instances and the variable ordering heuristics considered for this investigation.

3.1 Problem State Characterization

Four commonly used features are considered to characterize the instances in this investigation:

– **Problem size (N).** The problem size is defined as the number of bits required to represent the whole solution space. N is calculated as $\sum_{v \in V} \log_2(|D_v|)$, where $|D_v|$ is the domain size of variable v.

- **Constraint density** (p_1). The constraint density of an instance is defined as the number of constraints in which the variables participate divided by the maximum number of possible constraints in the instance.
- **Constraint tightness** (p_2). A conflict is a pair of values $\langle a, b\rangle$ that is not allowed by a particular constraint. Thus, the tightness of a constraint is the number of forbidden pairs over the total number of pairs that may occur between the two variables involved in the constraint. The tightness of a CSP instance is calculated as the average tightness over all the constraints within the instance.
- **Clustering coefficient** (C). The clustering coefficient estimates how close the neighbours of a variable are to being a complete graph. Thus, the clustering coefficient of a CSP instance is calculated as the average of the clustering coefficient of all the variables within the instance.

All the previous features lie in the range $[0, 1]$ (the values of N have been normalized to lie in the same range as the rest of the features).

3.2 Ordering Heuristics

Although various heuristics for variable ordering are available in the literature, we have selected a representative set of five of them for this investigation:

- **Domain (DOM).** DOM [10] prefers the variable with the fewest values in its domain.
- **Degree (DEG).** The degree of a variable is calculated as the number of edges incident to that variable. Thus, DEG selects the variable with the largest degree [7].
- **Domain Over Degree (DOM/DEG)**. DOM/DEG tries first the variable with that minimizes the quotient of the domain size over the degree of the variable [4].
- **Kappa (K).** The value of κ is suggested in the literature as a general measure of how restricted a combinatorial problem is. If κ is small, the instances usually have many solutions with respect to their size. When κ is large, instead, the instances often have few solutions or do not have any at all [9]. κ is defined as $\kappa = \frac{-\sum_{c\in C} \log_2(1-p_c)}{\sum_{v\in V} \log_2(|D_v|)}$, where p_c is the fraction of unfeasible tuples on constraint c. K prefers the variable whose instantiation produces the less constrained subproblem (the subproblem with the smallest value of κ).
- Weighted degree (WDEG). WDEG assigns a counter to each constraint, and every time such constraint is unsatisfied, the corresponding counter is increased by one [3]. WDEG prefers the variable with the largest weighted degree. This is, the variable with the largest sum of weights over its constraints.

In all cases, ties are broken by using the lexical ordering of the variables. Once a variable is selected, the value that participates in the fewest conflicts (forbidden pairs of values between two variables) will be tried before the others [14]. In case of ties, the minimum value will always be preferred.

4 A Lifelong Learning Selection Hyper-heuristic Model for CSPs

The motivation behind the lifelong learning selection hyper-heuristic (LLSHH) model for CSP is the lack of continuous learning in current hyper-heuristic models for CSP. The proposed hyper-heuristic model keeps learning with every instance it solves, improving its decisions. The system uses the information from previous instances solved with different heuristics to predict one suitable heuristic to apply once a similar instance appears.

In order to describe the LLSHH model for CSP we will introduce two important concepts: scenarios and predictors. A scenario attempts to answer the question "How is the performance of heuristic h on instance p?". Then, a scenario is the time required by a specific heuristic to solve a CSP instance. Every time an instance is solved, a new scenario is created and, during the training, the scenario is analyzed –instance p is solved by using heuristic h and the time consumed by the solving process is recorded. A predictor, on the other hand, gathers information from different scenarios to allow the system to predict the performance of a given heuristic on a new instance. A predictor contains a vector of features that characterize a point in the problem space, and a collection of scenarios associated to such a point. A scenario cannot belong to more than one predictor.

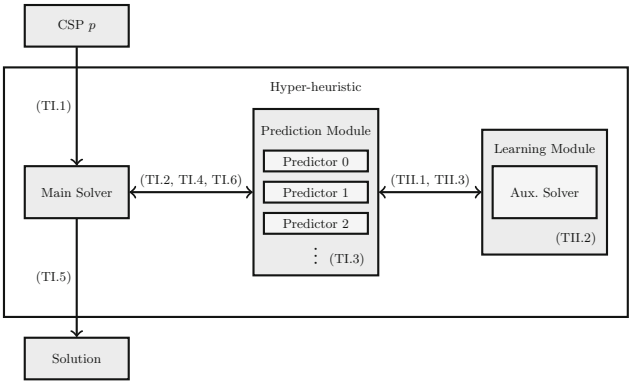

Fig. 1. The lifelong learning selection hyper-heuristic model for solving CSPs.

As shown in Fig. 1, the LLSHH model consists of three main components: a solver, a prediction module and a learning module. The solver is exclusively devoted to solving the instances presented to the system. It requests the prediction module for one suitable heuristic given the properties of the current instance under exploration. The prediction module has two tasks: it first recommends a heuristic and then, it communicates with the learning module to improve further decisions. The solver and the learning module run in parallel, each on its own

thread. Because both the solver and the prediction module can read from and write to the prediction module, the access to the prediction module is synchronized to avoid problems due to concurrency. A detailed view of the model is described in the following lines. Please note that two descriptions are provided as the solver and learning module run in parallel in different threads.

The main thread of the model (TI) runs as follows:

(TI.1) The user requests the hyper-heuristic to solve instance p.
(TI.2) The main solver requests the prediction module one suitable heuristic to apply for instance p.
(TI.3) The prediction module analyzes instance p by using the features described in Sect. 3.1. The characterization of instance p locates it on a specific point of the problem space. The prediction module analyzes all the available predictors and selects the one whose problem state is closest to the characterization of p. The closeness of the problem state to the characterization of instance p is calculated by using the Euclidean distance. For a predictor to be considered, there is a minimum acceptance distance d_{min}. This minimum distance is needed to allow the system to create new predictors and avoid using predictors with little relation to the current characterization of p.
  (TI.3a) The predictor r with the closest problem state to the characterization of p (if the distance is smaller than d_{min}) is selected. By using the information from the scenarios in predictor r, the system selects the heuristic with the smallest median recorded time among these scenarios. Let the selected heuristic be referred to as h.
  (TI.3b) If the prediction module cannot find a predictor with distance smaller than d_{min}, the module creates a new predictor r with a problem state equal to the characterization of p. One heuristic h from the ones described in Sect. 3.2 is randomly chosen.
(TI.4) The prediction module returns the heuristic selected from the previous step, h, and instance p is solved with that heuristic.
(TI.5) A solution is returned to the user.
(TI.6) A new scenario is created to register the performance of h on instance p. The scenario is assigned to predictor r.

As we already mentioned, the learning process is executed in a different thread. The learning process runs as follows:

(TII.1) The prediction module sends instance p, the selected predictor r and the selected heuristic h to the learning module.
(TII.2) With a probability α, each available heuristic (except for h, which is used and evaluated in thread T1 when the system solves the instance for the user) may be selected for the generation of new scenarios. For all the heuristics selected, a new scenario is created and analyzed.
(TII.3) The scenarios analyzed are sent to the prediction module to incorporate them to their respective predictors.

4.1 Benchmark Instances

In this investigation we have incorporated instances taken from a public repository available at http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html. All the instances in this work are binary CSPs coded in XCSP 2.1 format (http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf). In the following lines we briefly describe the classes of instances considered for this investigation:

- **RAND**. This class contains 50 random instances from the set `frb30-15.tgz` listed in the public repository.
- **QRND**. This class contains 140 random instances containing a small structure. The instances correspond to the sets `geom.tgz`, `composed-25-10-20.tgz` and `composed-75-1-2.tgz` in the public repository.
- **PATT**. The 52 instances in this class follow a regular pattern and involve a random generation. The instances were taken from sets `coloring.tgz`, `QCP-10.tgz` and `QCP-15.tgz` from the public repository.
- **REAL**. This class contains 18 instances taken from real world applications. The instances correspond to the sets `driver.tgz` and `rlfapScens.tgz` in the repository.

Although the proposed model is able to start from scratch (with no previous information), we recommend to conduct a training phase to provide some initial information about the heuristics and the instances that may be updated later when the test instances are solved. In this investigation, around 50 % of the instances of each class were used for training and the rest exclusively for testing.

5 Experiments

We conducted two main experiments in this investigation. In total, five hyper-heuristics were produced and tested. The first experiment is related to the ability of the hyper-heuristics to solve specific classes of instances. The second experiment explores the idea of a more general use of the hyper-heuristic where only one general method is capable of overcoming the use of a single heuristic for all the classes.

In both experiments we used a maximum running time of 25 seconds per instance, a minimum acceptance distance (d_{min}) equal to 0.1 and a probability of scenario generation (α) equal to 0.75.

5.1 Producing Hyper-heuristics for Specific Classes of Instances

We used the training instances from each class to produce one hyper-heuristic for the specific class used for training. In total, four hyper-heuristics were produced in this experiment: RAND-HH, QRND-HH, PATT-HH and REAL-HH. For each class, all the training instances were solved by using the corresponding hyper-heuristic. Once the system provided a solution to all the training instances, we used the hyper-heuristics produced to solve only the test instances from their

respective class. The idea of this experiment is to observe whether it is possible to produce hyper-heuristics that perform better than the heuristics applied in isolation for specific classes of instances.

Table 1 presents the success rate of the four hyper-heuristics on their respective classes of instances. The success rate in this investigation refers to the percentage of instances in each class where the hyper-heuristic required no more time than one specific heuristic. For example, the results shown in Table 1 indicate that for instances in class RAND, the hyper-heuristic RAND-HH is almost always at least as good as DOM, DEG, DOM/DEG and WDEG, but only in 12 % of the test instances for that class it was able to perform at least as well as K.

Table 1. Success rate of RAND-HH, QRND-HH, PATT-HH and REAL-HH on their respective class of instances with respect to each heuristic.

	DOM	DEG	DOM/DEG	K	WDEG
RAND	84.00 %	84.00 %	84.00 %	12.00 %	88.00 %
QRND	88.33 %	81.67 %	80.00 %	63.33 %	66.67 %
PATT	96.00 %	100.00 %	76.00 %	88.00 %	96.00 %
REAL	100.00 %	75.00 %	50.00 %	62.50 %	62.50 %

Table 2 complements the information of the performance of the four hyper-heuristics, as it shows the average gain/loss of each hyper-heuristic on instances from their respective classes of instances. For example, we know that the hyper-heuristic RAND-HH is almost always at least as good as DOM, DEG, DOMDEG and WDEG, but with the results from Table 2 we also know that when such hyper-heuristic was used, the average time required to solve a test instance in the class RAND decreased by 32.06 %, 23.57 % and 27.68 % with respect to DOM, DEG and WDEG, respectively. The result for DOM/DEG is interesting, as it shows that although the hyper-heuristic RAND-HH is in 84.00 % of the instances at least as good as DOM/DEG, there is no real benefit from using this hyper-heuristic (in average, 0.38 % time less per instance with respect to DOM/DEG). When we compared hyper-heuristic RAND-HH against K, we observed that, in average, it requires 56.09 % more time per instance than K.

The behaviour of hyper-heuristics QRND-HH, PATT-HH and REAL-HH shows that these hyper-heuristics are good solving options for the classes of instances they were trained for. In all cases, important savings in time were obtained by using hyper-heuristics on their corresponding classes of instances.

5.2 Producing a Hyper-heuristic for Multiple Classes of Instances

In the previous experiment we produced and tested hyper-heuristics for each specific class. Now, in this experiment we were interested in showing that there

Table 2. Average time gain/loss per instance of RAND-HH, QRND-HH, PATT-HH and REAL-HH on their respective class of instances when compared against each heuristic.

	DOM	DEG	DOM/DEG	K	WDEG
RAND	−32.06 %	−23.57 %	−0.38 %	+56.09 %	−27.68 %
QRND	−45.47 %	−52.12 %	−30.96 %	−8.35 %	−40.05 %
PATT	−17.17 %	−58.84 %	−14.37 %	−14.84 %	−51.60 %
REAL	−71.61 %	−23.66 %	−15.57 %	−56.17 %	−1.24 %

is a benefit from using hyper-heuristic as we can produce only one general method and then, use it on instances from different classes with acceptable results. In this section, we focused on producing one single hyper-heuristic (MC-HH) that could perform well on the four classes of instances.

The training process for MC-HH was conducted in the same way we did for training the four hyper-heuristics from the previous experiment but this time, we constructed only one set of training instances that includes the training instances from the four classes into a single training set.

Tables 3 and 4 show the results obtained for MC-HH when compared against each heuristic.

Table 3. Success rate of MC-HH on each class of instances.

	DOM	DEG	DOM/DEG	K	WDEG
RAND	80.00 %	84.00 %	48.00 %	8.00 %	88.00 %
QRND	88.33 %	80.00 %	86.66 %	56.66 %	68.33 %
PATT	68.00 %	88.00 %	60.00 %	76.00 %	84.00 %
REAL	75.00 %	75.00 %	75.00 %	87.50 %	62.50 %

Table 4. Average time gain/loss per instance of MC-HH on each class of instances when compared against each heuristic.

	DOM	DEG	DOM/DEG	K	WDEG
RAND	−40.94 %	−34.42 %	+0.70 %	+65.96 %	−39.37 %
QRND	−45.61 %	−52.23 %	−31.14 %	−8.40 %	−40.22 %
PATT	−11.33 %	−56.57 %	−7.26 %	−7.09 %	−48.72 %
REAL	−78.54 %	−45.96 %	−41.24 %	−70.24 %	−29.59 %

In general, MC-HH is not as good as the hyper-heuristics trained for each particular class of instances with regard to the success rate. But, its ability to

reduce the average time for solving the instances improved for some classes, specially for the instances from REAL. An important consideration is that K was the best performing heuristic for classes RAND, QRND and PATT, but for REAL the best heuristic was WDEG. MC-HH is capable of performing as well as K and WDEG in each of these classes (except for RAND), showing that one hyper-heuristic can combine the strengths of single heuristics to perform well on different classes of instances.

5.3 Discussion

There is an important observation that needs to be discussed about the running time of the proposed hyper-heuristic model. First, assuming that a hyper-heuristic always selects the same heuristic h, it will always require more time than h applied directly to solve the problems. The additional time is the result of revising the predictors to find one that is close to the characterization of the current instance. The benefit of using a hyper-heuristic in this model occurs when the hyper-heuristic changes the heuristic to apply based on the features of each instance being solved. When that happens, its performance can be superior to the one of a single heuristic applied to solve the same instances.

Finally, we observed that classes QRND, PATT and REAL are suitable to be solved by the model proposed. But, RAND is difficult to solve by the hyper-heuristic because of its lack of structure. It seems that the learning strategy in the hyper-heuristic is unable to properly characterize the instances by using the features described in Sect. 3.1. For this reason, we think that more features should be considered in the future.

6 Conclusion

In this paper we have described a lifelong learning hyper-heuristic model for solving CSPs. The hyper-heuristic learns from a set of scenarios that represent the historical performance of heuristics on previously solved instances. Then, the system creates predictors that group those scenarios and, when the system deals with a new instance, it predicts one suitable heuristic for such instance based on what the system knows about the heuristics and previously solved instances. Every instance the hyper-heuristic solves can produce, with probability α, a new scenario to increase the system's knowledge about the heuristics and the instances. Those scenarios are analyzed in a different thread, making the learning process transparent for the user. With this model, the system is continually learning from new instances it solves.

Although we observed promising results with this hyper-heuristic model, it is clear for us that some elements might be improved. For example, the minimum acceptance distance was introduced as a way to allow the hyper-heuristic to discard some predictors, the ones which are"not similar enough" to the current instance being solved. Despite this idea is something we found important to include (it allows the creation of new predictors), we think that should work in a

different way. Having one d_{min} for the distance worked well for this investigation but ignores the fact that some features may have a larger effect on the time required to solve an instance. For example, two instances with the same values of p_1, p_2 and C but different N may require completely different running times, as the number of variables increases the size of the search space. For this reason, we think that d_{min} should represent a vector, where each feature has its own minimum acceptance distance. Also, this value should be automatically updated according to the scenarios analyzed for each predictor. At this moment we only have preliminary ideas on how to achieve this, but we still need to figure out how to implement it in an efficient way.

Another important aspect to consider is that the order in which the instances are solved by the hyper-heuristic may affect the way the predictors are created. This may not seem like an issue for the experiments conducted in this investigation but opens the door for a future and important development of the model: predictor segmentation and integration. With segmentation, a predictor that has evidence that two instances with similar features require considerably different solving times may be split into two different predictors. On the other hand, integration deals with the idea of creating one new predictor by merging two or more existing ones.

Finally, we would like to include more features and heuristics to our model and compare the proposed model against other existing hyper-heuristic models for CSP to better estimate its quality.

Acknowledgments. This research was supported in part by ITESM Research Group with Strategic Focus in Intelligent Systems and CONACyT Basic Science Project under grant 241461.

References

1. Bitner, J.R., Reingold, E.M.: Backtrack programming techniques. Commun. ACM **18**, 651–656 (1975)
2. 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. ACM (2009)
3. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: European Conference on Artificial Intelligence (ECAI 2004), pp. 146–150 (2004)
4. Brelaz, D.: New methods to colour the vertices of a graph. Commun. ACM **22**, 251–256 (1979)
5. Burke, E., Hart, E., Kendall, G., Newall, J., Ross, P., Shulenburg, S.: Hyper-heuristics: an emerging direction in modern research technology. In: Handbook of metaheuristics, pp. 457–474. Kluwer Academic Publishers (2003)
6. Capodieci, N., Hart, E., Cabri, G.: Artificial immune systems in the context of autonomic computing: integrating design paradigms. In: Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion, GECCO Comp 2014, pp. 21–22. ACM, New York (2014)

7. Dechter, R., Meiri, I.: Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. Artif. Intell. **38**(2), 211–242 (1994)
8. Gagliolo, M., Schmidhuber, J.: Dynamic algorithm portfolios. Ann. Math. Artif. Intell. **47**, 3–4 (2006)
9. 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 the International Conference on Principles and Practice of Constraint Programming (CP 1996), pp. 179–193 (1996)
10. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artif. Intell. **14**, 263–313 (1980)
11. Hart, E., Sim, K.: On the life-long learning capabilities of a NELLI*: a hyper-heuristic optimisation system. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 282–291. Springer, Heidelberg (2014)
12. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **27**, 51–53 (1997)
13. Sim, K.E.H., Paechter, B.: A lifelong learning hyper-heuristic method for bin packing. Evol. Comput. **23**(1), 37–67 (2015)
14. Minton, S., Johnston, M.D., Phillips, A., Laird, P.: Minimizing conflicts: a heuristic repair method for CSP and scheduling problems. Artif. Intell. **58**, 161–205 (1992)
15. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science (2008)
16. Ortiz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: Learning vector quantization for variable ordering in constraint satisfaction problems. Pattern Recogn. Lett. **34**(4), 423–432 (2013)
17. Petrovic, S., Epstein, S.L.: Random subsets support learning a mixture of heuristics. Int. J. Artif. Intell. Tools **17**, 501–520 (2008)
18. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976)
19. Ross, P., Marín-Blázquez, J.: Constructive hyper-heuristics in class timetabling. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005), vol. 2. IEEE Press (2005)
20. Silver, D.L.: Machine lifelong learning: challenges and benefits for artificial general intelligence. In: Schmidhuber, J., Thórisson, K.R., Looks, M. (eds.) AGI 2011. LNCS, vol. 6830, pp. 370–375. Springer, Heidelberg (2011)
21. Silver, D.L., Yang, Q., Li, L.: Lifelong machine learning systems: beyond learning algorithms. In: Lifelong Machine Learning, Papers from the 2013 AAAI Spring Symposium, Palo Alto, California, USA, 25–27 March 2013
22. Sim, K., Hart, E.: An improved immune inspired hyper-heuristic for combinatorial optimisation problems. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, GECCO 2014, pp. 121–128. ACM, New York (2014)
23. Soto, R., Crawford, B., Monfroy, E., Bustos, V.: Using autonomous search for generating good enumeration strategy blends in constraint programming. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part III. LNCS, vol. 7335, pp. 607–617. Springer, Heidelberg (2012)