Improving Hyper-heuristic Performance for Job Shop Scheduling Problems using Neural Networks

E. Lara-Cárdenas[0000−0002−6357−4680], X. Sánchez-Díaz[0000−0003−2271−439X], I. Amaya[0000−0002−8821−7137], and J. C. Ortiz-Bayliss[0000−0003−3408−2166]

School of Engineering and Sciences, Tecnologico de Monterrey
Ave. Eugenio Garza Sada 2501, Monterrey, NL 64849, Mexico
a00398510@itesm.mx, sax@tec.mx, iamaya2@tec.mx, jcobayliss@tec.mx

Abstract. Job Shop Scheduling problems have become popular because of their many industrial and practical applications. Among the many solving strategies for this problem, selection hyper-heuristics have attracted attention due to their promising results in this and similar optimization problems. A selection hyper-heuristic is a method that determines which heuristic to apply at given points of the problem throughout the solving process. Unfortunately, results from previous studies show that selection hyper-heuristics are not free from making wrong choices. Hence, this paper explores a novel way of improving selection hyper-heuristics by using neural networks that are trained with information from existing selection hyper-heuristics. These networks learn high-level patterns that result in improved performance concerning the hyper-heuristics they were generated from. At the end of the process, the neural networks work as hyper-heuristics that perform better than their original counterparts. The results presented in this paper confirm the idea that we can refine existing hyper-heuristics to the point of being able to defeat the best possible heuristic for each instance. For example, one of our experiments generated one hyper-heuristic that produced a schedule that reduced the makespan of the one obtained by a synthetic oracle by ten days.

Keywords: Job shop scheduling · Hyper-heuristics · Neural networks.

1 Introduction

When solving a job shop scheduling problem (JSSP), the task is to schedule a set of jobs on a set of machines, subject to two constraints: each machine must handle at most one job at a time, and each job must respect a specified processing order throughout the machines. Thus, solving a JSSP requires to find a schedule for the jobs that minimizes the time required to complete all of them (the makespan). Since JSSPs become quite challenging to handle in real-world applications (they belong to the NP-hard problem class), finding more reliable methods to solve JSSPs is an essential topic of study.

Many methods to solve JSSPs have been proposed [1]. Among them, exact ones produce optimal results but are limited to the instance size. As a response, metaheuristics have commonly been used for solving the JSSP. Some examples include the use of simulated annealing [2–4], tabu search [5–7] and genetic algorithms [8–10], just to mention a few.

In addition to metaheuristics, recent research has also focused on heuristics specifically designed for this problem. Heuristics work by generating approximate solutions in a short time and with few computational resources. Some examples of this approach include dispatching rules [11] and the shifting bottleneck procedure [12]. Heuristics cannot guarantee the quality of the solutions, and there is no single one that performs best on every instance of the problem. For this reason, we sometimes rely on methods that combine the strengths of such heuristics in some intelligent fashion, in order to obtain a more stable performance for a broader range of instances.

The problem of selecting the most suitable algorithm or solving strategy for one particular situation is usually referred to as the algorithm selection problem. Examples of algorithm selection strategies include, but are not limited to: algorithm portfolios [13–15], selection hyper-heuristics [16, 17] and instance-specific algorithm configuration [18]. In general, all these methods manage a set of solving strategies and apply one that is suitable for the current problem state of the instance being solved. Striving to unify terms, from this point on, we will use selection hyper-heuristic to refer to the methods proposed in this paper.

The remainder of this document is organized as follows. Section 2 presents the most relevant concepts and literature related to this work. Section 3 describes the proposed solution model. The reader will find the experiments and their corresponding analysis in Section 4. Finally, Section 5 presents the conclusion and future work derived from this investigation.

2 Background and Related Work

A JSSP is formally defined as a set of jobs $J = \{1, \ldots, n\}$ and a set of machines $M = \{1, \ldots, m\}$. The order in which each job $j \in J$ must be processed through the machines is specified by a permutation $\sigma_j = (\sigma_j^1, \ldots, \sigma_j^m)$. For each job $j \in J$, a non-negative integer $p_{j,i}$ represents the processing time of job j on machine i. The time in which job j exits the system (i.e. the makespan of such a job), is denoted by C_j, while $C_{i,j}$ denotes the completion time of job j on machine i. The objective in this work is to minimize the total makespan (i.e. the summation over the makespan of all jobs).

Recently proposed solving strategies for JSSPs include the improved shuffled complex evolution [19], tabu search/path relinking (TS/PR) [20], evolutionary computation [21, 1], and particle swarm optimization [22]. Even so, recent studies [23, 24] encourage the generalization capabilities of neural networks. So, in order to take advantage of such a capability, we propose using neural networks as a way to generalize and improve upon existing selection constructive hyper-heuristics. The current literature already contains a few works where neural networks have been used within the field of hyper-heuristics. Some works have explored the idea of learning patterns in the performance of different heuristics for constraint satisfaction problems [25]. Other authors, such as Tyasnurita et al., have focused on learning heuristic selection for vehicle routing by using time-delay neural networks [26]. Similarly, some works have combined neural networks with other techniques to produce hyper-heuristics. For example, authors have combined neural networks with logistic regression to produce hyper-heuristics for educational timetabling [27]. The evolution of neural network topologies to construct hyper-heuristics for constraint satisfaction problems has also been explored [28]. To the best of our knowledge, there is no previous study where neural networks are used to improve the behavior of existing hyper-heuristics, as depicted in this work.

2.1 Instance Features

The features considered to characterize the problem state in this investigation are divided into two types. The first one characterizes the schedule (i.e. the solution found so far), and we consider three of them:

- **Average processed times (APT)**. APT expresses the ratio between the sum of the processing times of the previously processed activities and the sum of the processing times of the complete list of activities. This feature estimates how advanced the scheduling process is with respect to the initial conditions of the instance.
- **Dispersion of processing time index for scheduled activities (DPT)**. DPT is calculated as the ratio between the standard deviation of the processing times and the mean of the processing times.
- **Percentage of slack in makespan (SLACK)**. This feature refers to the ratio between the amount of available machine time (slack) in the whole schedule and the current makespan of the schedule. The larger the slack, the fewer the activities are, but also the more space where similar activities can be allocated.

The second type describes the problem state (i.e. the remaining/unscheduled part of the instance), and we also consider three of them:

- **Dispersion of processing time index for pending jobs (DNPT)**. For unscheduled activities: the ratio between the standard deviation of processing times and the mean of processing times.
- **Average non processed times (NAPT)**. NAPT is the complement of the APT. It is calculated as the ratio between the sum of processing times of pending activities and the sum of processing times of the whole list of activities.
- **Average pending processing times per job (NJT)**. NJT calculates, for all the pending activities, the sum of the processing times normalized for each job. Then, it divides such an amount by the number of pending jobs.

2.2 Heuristics

All heuristics considered in this work are constructive. So, they build a solution iteratively, i.e. taking one decision at the time. For their definition, let U_a be the list of pending activities, i.e. the ones to be scheduled. Let $S_i = (a_{j,i}, t_{a_j})$ be a list of tuples where i represents the machine number, $a_{j,i}$ is an activity of job j that has to be processed in machine i, and t_{a_j} is the time where activity a is being scheduled in job j. Thus, the considered heuristics for this investigation are the following:

- **Shortest Processing Time (SPT)**. From U_a select the activity $a_{j,i}$ with the shortest p_{ij}.
- **Longest Processing Time (LPT)**. From U_a select the activity $a_{j,i}$ with the longest p_{ij}.
- **Maximum Job Remaining Time (MRT)**. From U_a select the job that needs the most time for it to finish. It returns the first possible activity (in precedence order) that corresponds to said job in the first available time.
- **Most Loaded Machine (MLM)**. In U_a find the machine i which has maximum total processing time. The heuristic will return the activity a_j that has the lowest possible t_{a_j} if scheduled in machine i. If no activity is possible, then for the set of machines minus machine i, it selects the next machine with maximum total processing time until a suitable activity is found.
- **Least Loaded Machine (LLM)**. In U_a find the machine i which has minimum total processing time. The heuristic will return the activity a_j that has the lowest possible t_{a_j} if scheduled in machine i. If no activity is possible, then for the set of machines minus machine i, select the next machine with minimum total processing time until a suitable activity is found.
- **Earliest Start Time (EST)**. For U_a, get the activities which can be scheduled at a given state, this represents the possible activities. And, find the job that has the earliest possible starting time at the current problem state, and select the activity that corresponds to said job from the list of possible activities.

2.3 JSSP Instances

All the instances considered for this investigation were synthetically generated by using one algorithm taken from the literature [29]. The algorithm produces scheduling problems with a random distribution of the numbers for the machines and the completion times for each job. All the instances generated contain 15 machines and 15 jobs. In total, we generated 60 instances, where half of them are considered for training and the remaining half, for testing purposes.

3 Solution Model

The solution model proposed takes one selection constructive hyper-heuristic as input. This hyper-heuristic can be generated by using any selection hyper-heuristic generation method available. To clarify the terminology, from now on, we will refer to the hyper-heuristics used as input simply as input hyper-heuristics. By using these input hyper-heuristics, we solve the instances used for training and all the points in the instance space visited throughout the solving process (with their respective recommended heuristics) are recorded for further use. Then, the model focuses on refining the decisions made by the input hyper-heuristics by using neural networks.

The solution model first goes through a training stage where the neural network learns to behave as an improved version of the input hyper-heuristic. Information from the input hyper-heuristic is used to create a set of training examples for the neural network. In the second stage, the one devoted to testing, the neural network (the improved hyper-heuristic) is used to solve a set of unseen JSSP instances. The neural network receives a JSSP instance and decides, based on the problem state, the most suitable heuristic to apply to produce a good quality schedule.

The basic topology of the neural networks used in this investigation consists of at least three layers. Both the input and the output layers contain six neurons: one neuron per feature in the case of the input layer and one neuron per heuristic in the case of the output one. For these networks to be able to work as hyper-heuristics, they receive the features that characterize a JSSP instance, and only one of the output neurons must fire (the one that corresponds to the heuristic to apply). The number of hidden layers, as well as the number of neurons in each of these layers, is defined for each particular experiment.

4 Experiments

This section presents the experiments conducted in this investigation, where we use the makespan as a performance indicator (the lower, the better). Bear in mind that the makespan on a given instance serves the purpose of identifying its best solver, and that total makespan (i.e. the summation across all instances) indicates overall performance. All data are given for the test set.

4.1 Improving Hyper-heuristics Through Neural Networks: A Preliminary Approach

All the input hyper-heuristics used in this work were generated by using a recently proposed method that relies on simulated annealing for producing hyper-heuristics for JSSPs [30]. The hyper-heuristics produced by this method consists of a series of rules in the form (condition $\leftarrow$ action). The condition of these rules contains the values for each feature that make a heuristic (condition) more suitable than others at a particular moment in the construction of the schedule. Given one problem instance, the hyper-heuristic calculates the values of the features that characterize the current problem state. Then, it calculates the Euclidean distance between the condition of every rule in the hyper-heuristic and the current problem state. The rule which condition is closest to the problem state fires and, as a consequence, its corresponding heuristic is applied to the problem. The parameters used for running the simulated annealing hyper-heuristic generator include a minimum and maximum temperature of 1 and 100, respectively, and 150 steps.

As stated before, the result of the hyper-heuristic generation process is a collection of heuristic rules that map some regions of the instance space to specific heuristics. When a hyper-heuristic is used to solve an instance, it must decide which heuristic to use at given steps throughout the process. We refer to such steps as decision points. If we use a hyper-heuristic to solve an instance and record, for each decision point, the recommended heuristic, we can get a more detailed overview of the solving process. We call the set of decision points and their corresponding recommended heuristic, extended heuristic rules.

As a first experiment, we produced four hyper-heuristics by using the simulated annealing generation model (SAHH01 to SAHH04) on the training set. We then used the heuristic rules from each of these hyper-heuristics to train, for each SAHH, three improved hyper-heuristics by using neural networks (NNHH). Each one of the NNHHs incorporated slight changes on its topology, resulting in 12 different NNHHs. As aforementioned, the SAHHs make their decisions based on the Euclidean distances between the problem states and the conditions in the heuristic rules. On the other hand, the NNHHs decide which heuristics to apply at a given moment by using the weights in the network.

The topologies used for these experiments are defined by the number of layers and neurons in such layers (in the form 1 Input-N Hidden-1 Output). Then, topologies A, B, and C are, 6-64-64-18-6, 6-64-48-32-6, and 6-64-48-32-16-6, respectively. In all cases, a learning rate of 0.014 and a momentum of 0.87 were used. These parameters were set based on preliminary experimentation.

Table 1 shows the total makespan of the four hyper-heuristics produced through the simulated annealing method (second column). Time differences (expressed in hours) of the total makespan between each NNHHs and their corresponding SAHH are also shown (columns three to five). Here, a negative result indicates that a reduction in the total makespan was achieved by the improved hyper-heuristic. On the contrary, a positive result indicates that no improvement was obtained (and the schedule produced by the corresponding NNHH increases the makespan of the input hyper-heuristic). For simplicity, the cells in this table are referred to by a name resulting from the combination of the row and column. Hence, NNHH02B corresponds to the value 3,221. This indicates that using SAHH02 as input while considering topology B for the network, increases the schedule by 3,221 hours.

In the case of SAHH01, NNHH01A was able to reduce the makespan in just 30% of the instances of the test set. NNHH01B and NNHH01C showed better performance, reducing the makespan of SAHH01 in 63% of the instances. Similarly, NNHH02A and NNHH02B reduced the makespan produced by SAHH02 in 23.33% of the cases, while NNHH02C did so in 73.33% of them.

A similar analysis was conducted on SAHH03. Here, all topologies yielded the same performance improvement: 33.33% of the cases. Nonetheless, NNHHs

Table 1. Comparison of four SAHHs versus the NNHHs trained with heuristic rules. Best results are highlighted in bold.

Input hyper-heuristic	Total makespan	Topology A	Topology B	Topology C
SAHH01	40,538	1,596	**-1,387**	**-1,387**
SAHH02	40,748	3,221	3,221	**-2,597**
SAHH03	**41,299**	2,670	2,670	2,670
SAHH04	42,297	**-163**	1,672	**-3,146**

generated from SAHH04 improved on 50%, 30%, and 73.33% of the cases, respectively.

The most relevant information obtained from this first experiment is the evidence that a considerable reduction of hours in the total makespan of the schedules can be obtained by some of the neural network hyper-heuristics, concerning the original hyper-heuristics provided as input. For example, in cases like NNHH01C, the total time reduction (1,387 hours) accounts for almost two months. Similarly, NNHH04C offered an improvement of almost four months (3,146 hours). Of course, results were not always satisfactory and approaches like NNHH01A and NNHH04B actually require more time (1,596 and 1,672 additional hours, respectively).

4.2 Extended Heuristic Rules

In this experiment, we repeated the experimental methodology from Section 4.1 but this time, we used the extended heuristic rules to train the neural networks. With this experiment, we want to explore the behavior of the neural network hyper-heuristics when the number of training cases increases. We used SAHH01 to SAHH04, along with the three topologies used in the previous experiment to produce three neural network hyper-heuristics for each SAHH. The results from this experiment are depicted in Table 2.

Table 2. Comparison of four SAHHs versus the NNHHs trained with extended heuristic rules. Best results are highlighted in bold.

Input hyper-heuristic	Total makespan	Topology A	Topology B	Topology C
SAHH01	40,538	**-938**	1,759	1,759
SAHH02	40,748	**-191**	**-1,597**	1,386
SAHH03	**41,299**	967	835	835
SAHH04	42,297	325	**-163**	**-163**

As shown in Table 2, in all the cases, except for SAHH03, at least one of the neural network hyper-heuristics created using the extended heuristic rules reduced the total makespan achieved by the corresponding SAHH from which they were created. In the case of NNHH01A-EXT, it reduced the total makespan obtained by SAHH01 by 938 hours and performed better or equal than SAHH01 in 22/30 instances. However, NNHH01B-EXT and NNHH01C-EXT were unable to reduce the total makespan and were capable of improving the solution obtained by SAHH01 in only 36.66% of the instances.

Similarly, NNHH02A-EXT improved the schedules generated by SAHH02 in 191 hours (70% of the instances). Even so, NAHH02B obtained a more significant reduction in the total makespan (1,597 hours), while slightly improving the ratio of instances (73.33%). Hence, the reason falls to extensive individual improvements per instance for those instances where it is better than SAHH02. However, NNHH02C-EXT required 1,386 more hours to solve the test set, even when it performed better than SAHH02 in 33.33% of the instances.

Unfortunately, the neural network model was unable to improve the total makespan obtained from the schedules produced with SAHH03. Although no improvement in the overall makespan was obtained, NNHH03A-EXT outperformed SAHH03 in 43.33% of the instances, while NNHH03B-EXT and NNHH03C-EXT performed better than SAHH03 in 40% of the instances.

Lastly, NNHH04A-EXT performed at least as well as SAHH04 in 53.33% of the instances, but it was not enough to reduce the total makespan obtained by SAHH04. In contrast, NNHH04B-EXT and NNHH04C-EXT reduced the total makespan obtained by SAHH04 in 163 hours, demonstrating a performance improvement in most of the cases when compared to SAHH04.

4.3 Confirmatory Experiments with Extended Heuristic Rules

The rationale behind this experiment is to explore if using a better input hyper-heuristic can also improve the performance of the neural network hyper-heuristics. Thus, we produced 40 new SAHHs (by using the methodology described in Section 4.2) and ranked them according to their overall performance in the training set. We then selected the best SAHH and used it for generating three NNHHs (by using the same topologies described in Section 4.1).

Table 3 shows the resulting data. NNHHBestA-EXT and NNHHBestB-EXT were able to reduce total makespan by 286 and 806 hours, respectively. Also, they improved the solution of 67% and 63% of the instances, respectively. Nonetheless, and even though NNHHBestC-EXT improved the results for 80% of the instances, it worsened the overall result by 121 hours.

So far, we have compared the performance of the improved heuristic versus their original counterparts. For this last experiment, we aim at exploring the performance of the hyper-heuristic when compared against the best result the heuristics can produce: a synthetic oracle. Hence, ORACLE represents the makespan of the schedule obtained by the best heuristic for each particular instance in the test set.

ORACLE performs better than SAHHBest and NNHHBestC-EXT, by 43 and 164 hours, respectively. On the other hand, NNHHBestA-EXT outperforms both SAHHBest and ORACLE in 50% of the instances, resulting in a total reduction of the makespan by 286 and 243 hours, respectively. NNHHBestB-EXT significantly reduces the total makespan of both SAHHBest and ORACLE, since it requires 806 and 763 fewer hours to solve the test set, respectively. The reduction in the overall makespan corresponds, on average, to 25 less hours per instance (more than a day saved per instance).

Table 3. Comparison of the SAHH-B, and the NNHHs trained with its extended heuristic rules.

Method	Total makespan	Topology A	Topology B	Topology C
SAHHBest	38,288	**-286**	**-806**	121
ORACLE	38,245	**-243**	**-763**	164

4.4 Analysis of the Hyper-heuristics

When the 40 SAHHs generated in Section 4.3 were analyzed in a more detailed fashion, we observed that they produced contrasting results. For example, the difference in the total makespan between the best and worst SAHHs is 8,714 hours, which exposes the variation of simulated annealing for producing hyper-heuristics, even under similar conditions.

On this regard, the proposed improvement method is independent of the strategy used to produce the input hyper-heuristics to be improved. The only requirement for our model to work is that the hyper-heuristics used as input can be represented as heuristic rules or that they can be used to produce the extended heuristic rules.

To support the previous statement, a Principal Component Analysis (PCA) was conducted to map the decisions taken by two selected hyper-heuristics into a 2D space, as illustrated in Fig. 1.

We obtained some relevant findings from this analysis. If the bottom right regions of the figures are compared, it can be appreciated in Fig. 1 (b) that the action area of the heuristics is delimited by an almost straight line. Similarly, the central sections of Fig. 1 (b) show the order given to the heuristic rules (straight lines) in comparison with those present in Fig. 1 (a).

5 Conclusion

This paper presents the first ideas on how to improve the performance of existing heuristics by using neural networks. Although hyper-heuristics, in general, produce competent results, the differences in performance between two hyper-heuristics (generated with the same model and parameters) might sometimes be contrasting. This may happen because of the many decisions in the generation process. Then, by using the proposed approach, we can try to improve

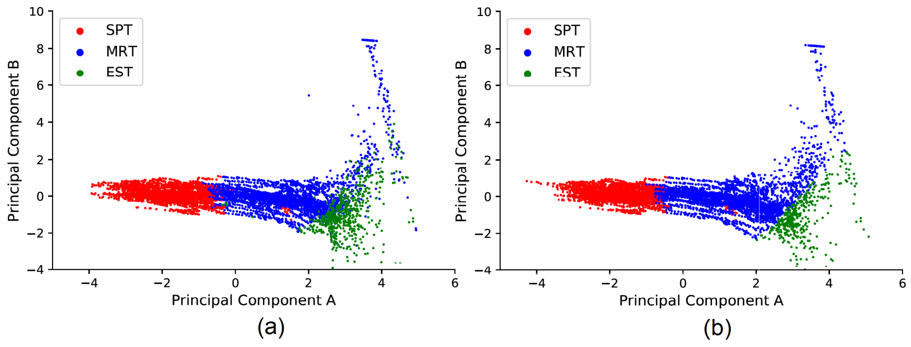

Fig. 1. PCA for extended heuristic rules and selected heuristic on each point by (a) the best generated SAHHBest and (b) NNHHBestB-EXT.

such hyper-heuristics to reduce the performance gap between most of them. It is important to mention that, in some cases, the little time investment to try to improve an existing heuristic (a few minutes) can represent several weeks of time savings.

The hyper-heuristic improvement model described in this document seems to be independent of the way the input hyper-heuristics are produced as long as we can extract the training data from the input hyper-heuristic. The fact that the model is independent of the way the input hyper-heuristics are produced has another significant benefit: the model is also domain independent. Then, we expect that applying this improvement strategy on hyper-heuristics produced for other problem domains is likely to succeed. Of course, further experimentation that involves more problem domains is a must for future work. Some other ideas remain unexplored and should also be addressed as future work. For example, more extensive experimentation that includes different neural network topologies as well as the potential exploitation of past decisions (the use of a memory module) to decide the heuristic to apply.

Acknowledgments

This research was partially supported by CONACyT Basic Science Project under grant 287479 and ITESM Research Group with Strategic Focus on Intelligent Systems.

References

1. Kurdi, M.: An effective new island model genetic algorithm for job shop scheduling problem. Computers & Operations Research **67** (2016) 132–142
2. Hernández-Ramírez, L., Frausto Solís, J., Castilla-Valdez, G., González-Barbosa, J.J., Terán-Villanueva, D., Morales-Rodríguez, M.L.: A hybrid simulated annealing

for job shop scheduling problem. International Journal of Combinatorial Optimization Problems and Informatics **10** (2018) 6–15
3. van Laarhoven, P.J.M., Aarts, E.H.L., Lenstra, J.K.: Job shop scheduling by simulated annealing. Operations Research **40** (1992) 113–125
4. Satake, T., Morikawa, K., Takahashi, K., Nakamura, N.: Simulated annealing approach for minimizing the makespan of the general job-shop. International Journal of Production Economics **60-61** (1999) 515–522
5. Bozejko, W., Gnatowski, A., Pempera, J., Wodecki, M.: Parallel tabu search for the cyclic job shop scheduling problem. Computers & Industrial Engineering **113** (2017) 512–524
6. Nowicki, E., Smutnicki, C.: A fast taboo search algorithm for the job shop problem. Management Science **42** (1996) 797–813
7. Zhang, C., Li, P., Guan, Z., Rao, Y.: A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. Computers & Operations Research **34** (2007) 3229–3242
8. Bhatt, N., Chauhan, N.R.: Genetic algorithm applications on job shop scheduling problem: A review. In: International Conference on Soft Computing Techniques and Implementations (ICSCTI). (2015) 7–14
9. Ghedjati, F.: Genetic algorithms for the job-shop scheduling problem with unrelated parallel constraints: Heuristic mixing method machines and precedence. Computers & Industrial Engineering **37** (1999) 39–42
10. Hou, S., Liu, Y., Wen, H., Chen, Y.: A self-crossover genetic algorithm for job shop scheduling problem. In: IEEE International Conference on Industrial Engineering and Engineering Management. (2011) 549–554
11. Blackstone, J.H., Phillips, D.T., Hogg, G.L.: A state-of-the-art survey of dispatching rules for manufacturing job shop operations. International Journal of Production Research **20** (1982) 27–45
12. Adams, J., Balas, E., Zawack, D.: The shifting bottleneck procedure for job shop scheduling. Management Science **34** (1988) 391–401
13. Epstein, S.L., Freuder, E.C., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming. CP '02, London, UK, UK, Springer-Verlag (2002) 525–542
14. Petrovic, S., Qu, R.: Case-based reasoning as a heuristic selector in a hyper-heuristic for course timetabling problems. In: Proceedings of the 6th International Conference on Knowledge-Based Intelligent Information Engineering Systems and Applied Technologies (KES'02). Volume 82. (2002) 336–340
15. OMahony, E., Hebrard, E., Holland, A., Nugent, C., OSullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish conference on artificial intelligence and cognitive science. (2008) 210–216
16. Ortiz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: Combine and conquer: an evolutionary hyper-heuristic approach for solving constraint satisfaction problems. Artificial Intelligence Review **46** (2016) 327–349
17. Sim, K., Hart, E., Paechter, B.: A lifelong learning hyper-heuristic method for bin packing. Evol. Comput. **23** (2015) 37–67
18. Malitsky, Y.: Evolving instance-specific algorithm configuration. In: Instance-Specific Algorithm Configuration. Springer International Publishing (2014) 93–105
19. Zhao, F., Zhang, J., Zhang, C., Wang, J.: An improved shuffled complex evolution algorithm with sequence mapping mechanism for job shop scheduling problems. Expert Systems with Applications **42** (2015) 3953–3966

20. Peng, B., Lü, Z., Cheng, T.: A tabu search/path relinking algorithm to solve the job shop scheduling problem. Computers & Operations Research **53** (2015) 154–164
21. Cheng, T.C., E., Peng, B., L, Z.: A hybrid evolutionary algorithm to solve the job shop scheduling problem. Annals of Operations Research **242** (2016) 223–237
22. Gao, L., Li, X., Wen, X., Lu, C., Wen, F.: A hybrid algorithm based on a new neighborhood structure evaluation method for job shop scheduling problem. Computers & Industrial Engineering **88** (2015) 417 – 429
23. Neyshabur, B., Bhojanapalli, S., McAllester, D., Srebro, N.: Exploring generalization in deep learning. In: Advances in Neural Information Processing Systems. (2017) 5947–5956
24. Olson, M., Wyner, A., Berk, R.: Modern neural networks generalize on small data sets. In: Advances in Neural Information Processing Systems. (2018) 3619–3628
25. Ortiz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: Neural networks to guide the selection of heuristics within constraint satisfaction problems. In Martínez-Trinidad, J.F., Carrasco-Ochoa, J.A., Ben-Youssef Brants, C., Hancock, E.R., eds.: Pattern Recognition, Springer Berlin Heidelberg (2011) 250–259
26. Tyasnurita, R., Özcan, E., John, R.: Learning heuristic selection using a time delay neural network for open vehicle routing. In: IEEE Congress on Evolutionary Computation (CEC). (2017) 1474–1481
27. Li, J., Burke, E.K., Qu, R.: Integrating neural networks and logistic regression to underpin hyper-heuristic search. Knowledge-Based Systems **24** (2011) 322–330
28. Ortiz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: A neuro-evolutionary hyper-heuristic approach for constraint satisfaction problems. Cognitive Computation **8** (2016) 429–441
29. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64** (1993) 278–285
30. Garza-Santisteban, F., Snchez-Pmanes, R., Puente-Rodrguez, L.A., Amaya, I., Ortiz-Bayliss, J.C., Conant-Pablos, S., Terashima-Marn, H.: A simulated annealing hyper-heuristic for job shop scheduling problems. In: 2019 IEEE Congress on Evolutionary Computation (CEC). (2019) 57–64