Hyper-heuristics Reversed: Learning to Combine Solvers by Evolving Instances

Ivan Amaya*, José Carlos Ortiz-Bayliss*, Santiago Conant-Pablos* and Hugo Terashima-Marín*
*Tecnologico de Monterrey
National School of Engineering and Science
Email: {iamaya2, jcobayliss, sconant, terashima}@tec.mx

***Abstract*—It is common to find that training of selection hyper-heuristics is done perturbatively. The process usually starts with a random selection module and iterates over a set of instances until finding appropriate values for such module. In this work, however, we present a model for creating selection hyper-heuristics constructively. To achieve so, we use a set of instances evolved for such a task. For each low-level heuristic, we evolved a set of problem instances that are more easily solvable by that particular heuristic (compared to the other ones). Each group contains instances easily solvable with the corresponding heuristic but not with the remaining ones. Thus, our model creates its selector by calculating the centroid of each group. For doing so, the model defines a rule that maps said centroid to one corresponding action (in this case, a low-level heuristic). To test our approach, we select the one-dimensional Bin Packing Problem and set four target performance levels (which we refer to as deltas) for instance generation. Then, we analyze all the possible combinations of deltas. We study how performance of the generated hyper-heuristics shift when they are created using a different number of instances. Our data shows the feasibility of creating a hyper-heuristic under the stated conditions. Effectiveness of the model depends on the deltas used though we observed that higher deltas are useful while lower deltas are not. For example, when considering a delta level of 2.0, our method produced hyper-heuristics with an accumulated average waste 12% lower than that of the best heuristic. But, for a delta level of 0.5, it became impossible to outperform the heuristics.**

***Index Terms*—Hyper-heuristics; Bin Packing Problem; Instance generation.**

I. Introduction

Since there is no single 'best' solver for all scenarios, hyper-heuristics have emerged as one of different approaches that seek to discern when is best to use each one of the available solvers. In doing so, they try to offer more flexibility when solving a wider variety of optimization problems [1]. The term 'hyper-heuristic' was initially used to describe heuristics for choosing heuristics. Nowadays, such a definition has been expanded to include automatic generation of heuristics. An important difference between hyper-heuristics and other approaches such as metaheuristics, is that the former operate over the solver space whilst the latter do it over the solution space. Hence, hyper-heuristics do not solve a problem directly. Instead, they select an appropriate solver for the current conditions of the problem. On this regard, they are similar to other approaches, such as algorithm portfolios [2]. However, the latter use the same solver for completely solving a problem instance. Hyper-heuristics, on the other hand, use a combination of solvers. More details on hyper-heuristics can be consulted in [1], where the authors present an in-depth survey about the topic.

This project was funded by Consejo Nacional de Ciencia y Tecnología (CONACyT) through the Basic Science Projects with grant numbers 241461 and 287479, and by the Research Group with Strategic Focus in Intelligent Systems at Tecnologico de Monterrey.

As indicated by Pillay and Qu, various categories of hyper-heuristics exist [3]. A broad distinction can be done between the constructive and perturbative ones. In the former, a solution is built step by step. When completed, it is returned without further modifications. In the latter, an initial (complete) solution is modified until a desired criterion is met. In this work we will focus on constructive hyper-heuristics. Moreover, we will use selection hyper-heuristics. In this category, the problem state is mapped through a set of features so the most suitable heuristic can be applied. One of the main challenges for selection hyper-heuristic models is to obtain a proper characterization of such a state. There are different ways to get such a mapping, and examples include machine learning [4] and evolutionary algorithms [5]. Nonetheless, how effective the selection is greatly depends on the predictive power of the feature set [6]. Hence, there have been efforts at improving this power by using feature transformations on selection hyper-heuristics [7], [8]. The base idea of such an approach is to expand conflicting regions in the feature domain so that hyper-heuristics become easier to train.

Another component of great relevance in hyper-heuristics is the set of instances that these methods will solve. As we mentioned above, no solver can deal with all problems in the best possible way. So, new approaches are usually focused on improving the solution of existing benchmarks. But, such reference points are seldom updated. Still, work has been carried out in this regard. For example, the OR-Library have been distributing test data for Operations Research (OR) problems since 1990 [9]. The Mixed Integer Programming Library (MIPLIB) is yet another example [10]. MIPLIB was originally proposed in 1992 but the current version dates from 2010. In a similar fashion to OR-Library, MIPLIB covers several domains, including Bin Packing. Other works of interest on this regard include [11]–[13]. Alas, we will not detail them here due to space constraints.

Generating instances this way implies first creating the problem and then verifying the performance of solvers. But, it

is also possible to do it in the opposite direction, by creating instances tailored to the attributes of a solver. A common way of doing so is to use an 'intelligent' generator, which evolves the instance for a target solver. For example, van Hemert [14], [15] used an evolutionary algorithm to detect hard to solve instances in Constraint Satisfaction Problems (CSPs). Conversely, Smith-Miles et al. [16], [17] have focused on intentionally creating instances of the Traveling Salesman Problem (TSP) that were easy/hard for certain algorithms. Also, in [18] Amaya et al. used a messy genetic algorithm for evolving instances for the 1D Bin Packing Problem (BPP). The authors laid out different objective functions that focused on favoring/hindering a selected solver. They also managed to evolve instances with high/low variation across the solvers.

Training of selection hyper-heuristics is usually done on a subset of the available instances, searching for a set of rules that works well for them. However, we have yet to see an approach where instances tailored to each solver are used. This work seeks to fill that gap. We believe this could prove useful for better understanding hyper-heuristics and for building them according to the nature of available solvers.

This manuscript is organized as follows: Section II presents the main idea behind our hyper-heuristic model, while sections III and IV provide the organization of tests and the resulting data (respectively); Section V summarizes the main conclusions about our work.

II. Our proposed approach

A. An overview of the hyper-heuristic model

Our hyper-heuristic model can be represented by a matrix with R rows and $F+1$ columns, where R is the number of rules and F stands for the number of features. Hence, in a general sense, the model maps the problem to a feature domain given by features $F_1, F_2, ..., F_F$, which can be changed through a set of solvers (usually low-level heuristics) given by $h_1, h_2, ..., h_h$. This mapping is used to locate the current state of the problem, given by vector F_T, and the decision points of the hyper-heuristic that shape the regions where each solver is used. The decision about which action (i.e. solver) to use given a problem state is taken based on the nearest neighboring rule, measured through a Euclidean distance.

For the combinatorial problem being analyzed in this work, using an action implies packing an item into a bin. This process does not solve the instance completely. Instead, it changes the current state of the problem since the instance now has one less element to pack. Thus, a new decision can be taken, which may fall under the purview of another action. It is precisely this behavior what allows a hyper-heuristic to arrive at a solution different from those yielded by each solver (e.g. low-level heuristic) on their own.

Figure 1 shows an example of the hyper-heuristic model with three rules and three features. Actions are represented by integers corresponding to the heuristic IDs. Note that the current state of the problem, $F_T = [0.4, 0.7, 0.8]$ is compared against the selector by calculating the distance to each rule. Since the first rule is the closest one, action 1 will be used

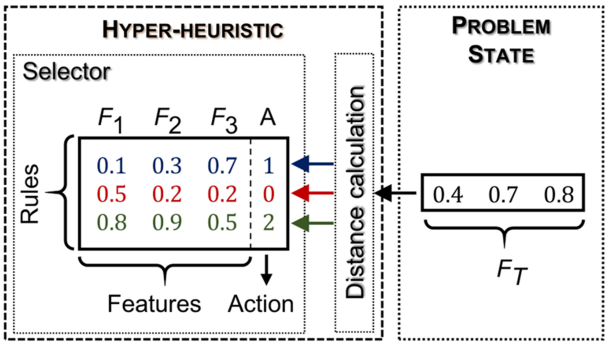


Fig. 1. An example of the hyper-heuristic model used in this work, considering three features (F_1, F_2, F_3) and three rules. Numbers in the action column correspond to the ID of the available solvers.

to decide the bin in which the next item will be packed. But, afterwards it may be the case that the second rule becomes closest to the problem state, so action 0 would be used next.

B. Available solvers

In this work, we considered four low-level heuristics that pack the first item in what remains of the instance:

- **First Fit Heuristic (FF):** Place the item into the lowest numbered bin where the item fits.
- **Best Fit Heuristic (BF):** Find all the bins that can store the item and place it into the one that leaves the least free space.
- **Worst Fit Heuristic (WF):** Find the bins that can store the item and place it into the one with the most available space.
- **Almost Worst Fit Heuristic (AWF):** Similar to WF, but instead uses the bin with the second most available space.

Should an item not fit in any of the existing bins, it must be placed into a new one. Also, our choice of heuristics does not hinder the scope of our work, as instances can be generated for a solver as long as its performance for the instance can be assessed.

C. Available features

For this work, we considered the same three features used in [18] for analyzing the generated instances: Average item length (AL), Standard deviation of the item lengths (SL), and Ratio of big pieces (RBP, i.e., the ratio of items whose length is above half of the bin capacity). No solution-related features are considered in this work.

D. Meaning of the delta level

In this work, we refer to the delta level (Δ) as the value given by eq. (1), where $Perf_{one}$ is the performance achieved by a heuristic of interest and $Perf_{others}$ is the performance of the remaining heuristics. We selected average waste as the performance metric. Hence, a negative delta implies that the heuristic of interest is performing as desired.

$$\Delta = Perf_{one} - \min\left(Perf_{others}\right) \quad (1)$$

Let us suppose that the heuristic of interest is FF and it yields an average waste of 1.5, whilst heuristics BF, WF, and AWF provide average wastes of 1.9, 1.8, and 2.0. Hence, the delta level for this scenario is $1.5 - 1.8 = -0.3$.

E. Instances used in this work

Since we need instances with specific delta levels, we followed the approach shown in [18] to evolve them. However, we changed the objective function to eq. (2), where Δ_d is the desired delta level (i.e. 0.5, 1.0, 1.5, or 2.0). This way, the global optimum (zero) represents an instance where the average waste given by the solver of interest is Δ_d units better than the one given by the best of the remaining solvers.

$$F_{obj} = (\Delta + \Delta_d)^2 \tag{2}$$

We generated a dataset of 150 easy-to-solve instances for each solver on every scenario, and with the following configuration per instance: 100 items, with a bin capacity of 32 and a maximum length per item of 32.

F. Creation and usage of a hyper-heuristic

Other selection hyper-heuristic models available in literature train their selectors in a perturbative way, e.g. by creating an initial random set of rules and optimizing them over a given training set [8]. However, in this work we follow a different approach. We believe that a hyper-heuristic can be derived in a constructive way by using information from the instances, i.e. without optimizing the selector. The rationale behind this idea is that previous work has shown that instances are mapped to different regions of the feature space depending on the heuristic that solves them most easily [19]. In order to achieve our goal, we need a set of instances where each solver excels. Also, we need a way for separating the data of each solver. So, we use Principal Component Analysis (PCA) over the set of instances to reduce the number of dimensions from three to two. This has the additional benefit of allowing us to easily inspect the location of each cluster of instances. Thus, the general process required for building a hyper-heuristic can be summarized as:

1) Define an empty selector, S, given by an empty matrix.
2) Randomly select a subset of n instances for every solver and every delta level from the set of N available instances, where $n \leq N$. In this work, $N = 150$ was used. Please note that N is also given per solver and per delta level.
3) Map each instance by calculating the initial feature values and storing them into a matrix F_I. This implies calculating the features from Sect. II-C on each unsolved instance, which yields a feature vector per instance, representing each row of F_I. Hence, matrix F_I contains $n \times N_s \times N_d$ rows, where N_s is the number of solvers and N_d is the number of delta levels.
4) Transform the mapping space using PCA:
   - Normalize each column of the instances by subtracting the respective mean μ and dividing in the corresponding standard deviation σ, obtaining the normalized matrix F_{I_N}.
   - Calculate the covariance matrix (Σ) of the normalized data (F_{I_N}).
   - Calculate the first two eigenvectors of Σ and store them in U.
   - Multiply the normalized data (F_{I_N}) by U.
5) For each subset of transformed instances (i.e. for each solver and each delta level):
   - Calculate the centroid, as the average of the transformed feature values.
   - Define a rule given by such a centroid and the action of the selected solver, and add it to the selector (S).
6) Return S, U, μ, σ.

The reason for storing U, μ, σ is that they are required for mapping the current problem state, F_T, to the space in which the selector was built. Therefore, to take a decision with the hyper-heuristic model the following steps must be covered:

1) Calculate the current state of the problem, F_T.
2) Use μ and σ to normalize F_T, following the element-wise operation $F_{T_N} = (F_T - \mu)/\sigma$.
3) Use U to transform the normalized values into the reduced space given by the PCA model, following the matrix operation $F_{PCA} = F_{T_N} \times U$.
4) Find the rule of the selector closest to the transformed values (F_{PCA}).
5) Take the decision given by the aforementioned rule.

This mapping is the only change required to implement our proposed hyper-heuristic model. Thus, the process described in Sect. II-A still applies.

III. Methodology

Throughout this work, we followed a two-stage methodology (Figure 2). All tests were run in an ASUS FX503 laptop with an Intel i7-7700HQ CPU and 8 GB of RAM at 2400 MHz. Moreover, we considered scenarios with four delta levels: $\Delta_d = \{0.5, 1.0, 1.5, 2.0\}$. Also, we built hyper-heuristics using 15 different n values: $10, 20, 30, 40, \ldots, 150$. Each testing stage is described in the following lines.

A. Preliminary

We began by analyzing the simplest approach: Whether it was possible to create a hyper-heuristic considering a single delta level for the solvers. Thus, we ran four different scenarios. Each one of them was further split into 15 tests. Each test used a random subset of the instances from each solver, to analyze how performance shifts when considering different number of instances for building the selector. Since we need to choose subsets, we repeated each test 30 times.

In order to assess the quality of each hyper-heuristic, we compare their performance against that of each standalone solver. Striving to preserve the same reference level at each scenario, the testing set was selected as the one formed by all available instances (i.e. 150 instances per solver, for a total of 600 instances).

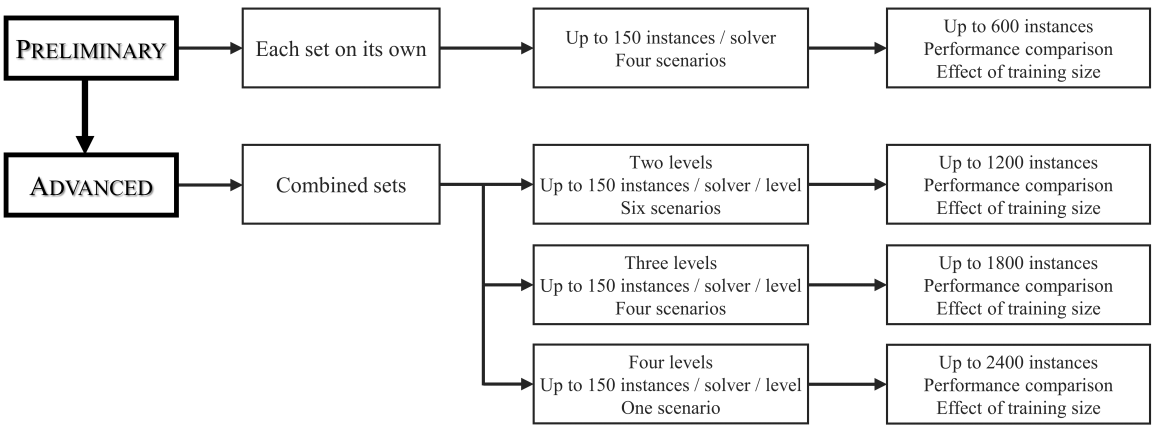


Fig. 2. Two-stage methodology followed in this work.

B. Advanced

For the second stage of this work, we moved on to analyze the performance of hyper-heuristics built from combinations of different delta levels. Since we are considering four of them, we analyze all possible combinations of two (6), three (4), and four (1) delta levels. Therefore, in this stage we analyzed a total of 11 different scenarios. Once again, we split each scenario into 15 tests with different number of instances per heuristic and per delta level, and repeated each test 30 times. As in the first stage, we compare the performance of the hyper-heuristics against that of the standalone solvers.

IV. Results

This section presents the most relevant data of our experiments. To facilitate reading we have organized this section following the structure of the previous section.

A. Preliminary

Figure 3 shows the way in which performance of hyper-heuristics change under different scenarios (boxes), as well as the behavior of low-level heuristics (dashed lines). Since the testing set is the same for each subplot, heuristics perform steadily. However, it is evident that building the hyper-heuristic with fewer instances leads to a more variable performance. Also, it is interesting to note that in all cases, the first fit (FF) and best fit (BF) heuristics performed similarly and quite better than the other two (WF and AWF), even though they all had the same number of instances (150) with the same Δ_d (0.5, 1.0, 1.5, and 2.0).

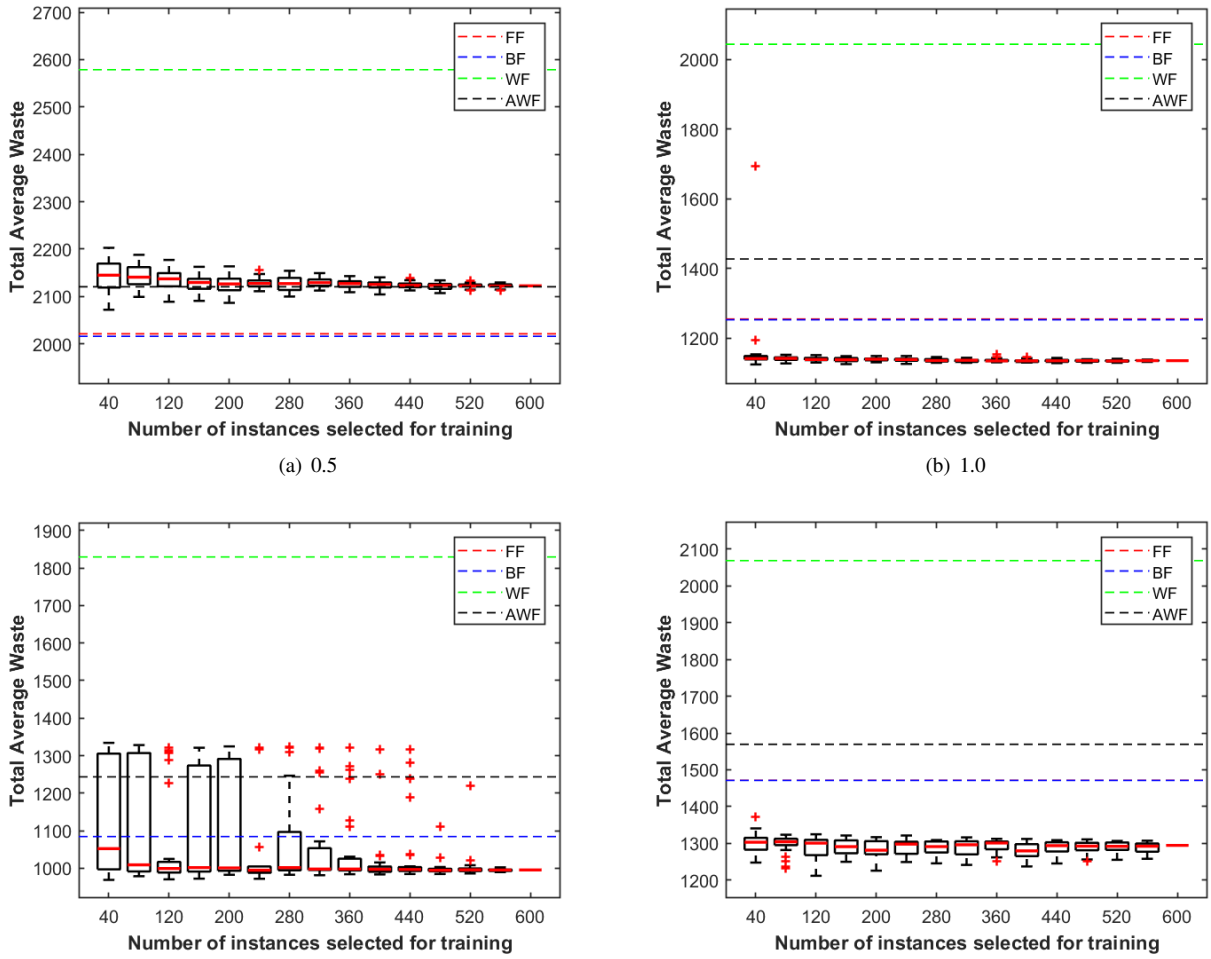
Another interesting behavior in Fig. 3 is that only for the first case (i.e. a performance gap of 0.5 as shown in Fig. 3(a)), it was impossible to detect hyper-heuristics that outperformed the low-level heuristics. We believe this is due to the small performance gap across heuristics, which may lead to overlaps in the feature domain. In all the other cases, the median performance of hyper-heuristics proved to be better than the performance of low-level heuristics, even when selecting a handful of instances for generating the rules. Please bear in mind that the hyper-heuristic model being proposed learn its rules by calculating the centroid of easy-to-solve instances for each solver (i.e. each low-level heuristic). Thus, being able to achieve a good enough hyper-heuristic with only 10 instances per solver (for a total of 40), is certainly interesting. It is also clear, of course, that using such a low number of instances leads to a high variability in performance, so it is better to use some more. Furthermore, it seems that a higher performance gap leads to higher benefits when building the hyper-heuristic. For example, in the second and third cases, using the hyper-heuristic led to an accumulated average waste about 10% smaller than the one given by the best heuristic. However, in the final case (Fig. 3(d)) the reduction rose to over 12%.

Moreover, we wondered whether using PCA was beneficial for our approach. Thus, we ran a quick experiment without the PCA module. So, rules are given by all three features and the feature vector representing the current state of the problem, F_T, is directly compared against them. We noticed that the module seems to aid instance differentiation. For example, for the lowest delta level, removing the module worsened the median result, making three of the four heuristics better than the hyper-heuristic. In the case of the second delta, removing the module was also harmful, rendering most of the generated hyper-heuristics useless. We omit more details about these tests because of space restrictions, though we certainly plan to study this dependence more deeply in the future.

B. Advanced

We now present data achieved by repeating the hyper-heuristic building process for sets created by combining the instances of each delta level shown in the previous section.

1) Two levels: It is interesting to see that some of the elements detected in the previous stage persist (Fig. 4). For example, the BF and FF heuristic still perform virtually equally. Also, once again using instances with a small Δ_d (i.e. 0.5) is harmful for the hyper-heuristic model. This time around, however, the benefit of using a higher delta is more evident,



(a) 0.5

(b) 1.0

(c) 1.5

(d) 2.0

Fig. 3. Performance of low-level heuristics and of hyper-heuristics, over the whole set of generated instances (Preliminary tests). Each box shows the performance of 30 different hyper-heuristics. Each plot shows data for instances tailored to a given Δ_d. Note: The FF and BF heuristics perform quite similarly, so their lines tend to overlap.

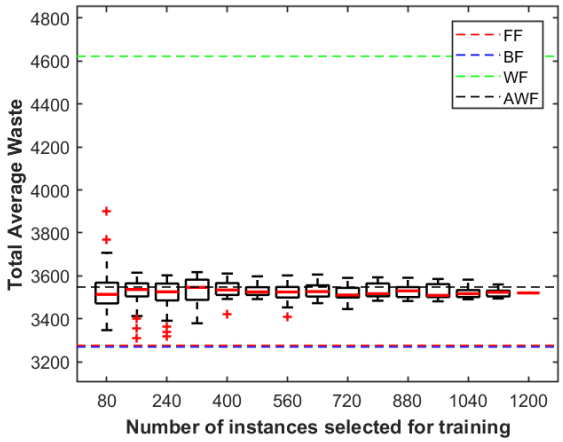
as only the scenarios that included $\Delta_d = 2.0$ yielded hyper-heuristics that outperformed the low-level heuristics. In fact, scenarios (e) and (f) show that the median performance was always better than the best heuristic, even when built with the fewest instances (10 per solver per delta level). Nonetheless, in the former the gain from using the hyper-heuristic was of about 3%, whilst in the latter it was possible to reduce the accumulated average waste by about 10%.

2) Three levels: Adding a third delta level to the dataset increases the complexity of the model, as the hyper-heuristic now contains 12 rules instead of 4 or 8 (single and dual delta levels, respectively). Moreover, and in a similar fashion to the previous scenarios, including instances with a $\Delta_d = 0.5$ is harmful for the process (Fig. 5). The hyper-heuristics with the best performance were those that did not include such a Δ_d (Fig. 5(d)). Even so, no configuration was able to outperform the low-level heuristics. We believe this can be due to the complexity of the model and that it may be alleviated by using instances with higher Δ_d values. Alas, exploring such a path was out of the scope of this manuscript.

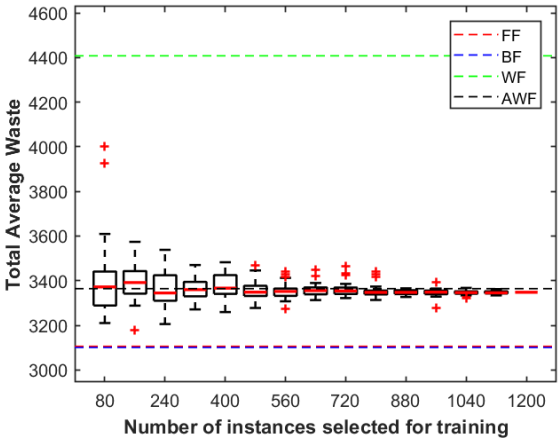
3) All levels: As expected, using all Δ_d levels was not fruitful as the instances with small delta values proved troublesome (Fig. 6). Still, considering instances with several delta values seem to help reduce variability in the data. Even so, it is clear that heuristics BF and FF are virtually tied in terms of performance, even though there is exactly the same number of instances with exactly the same Δ_d for each heuristic.

V. Conclusions and Future Work

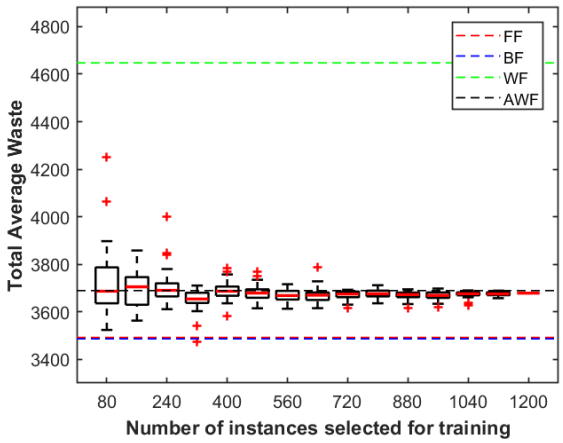
Throughout this work we presented a selection hyper-heuristic model whose rules represent the centroid of easy-to-solve instances with a given Δ_d. Such instances were generated following an evolutionary approach described in [18], and are available upon request. To validate our approach, we ran multiple tests over four different Δ_d values. We analyzed a total of 15 different scenarios, each one of them divided into 15 tests with a different number of samples. To observe stability of our approach, we repeated each test 30 times.



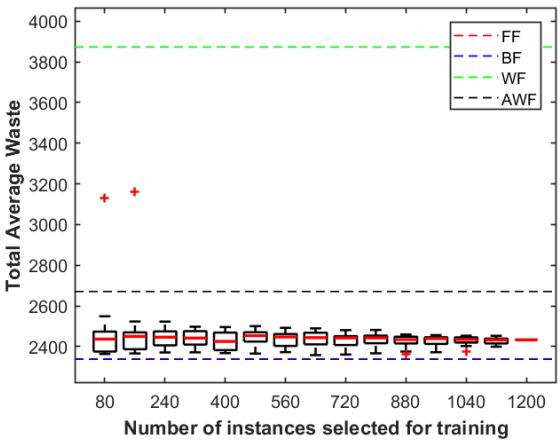
(a) 0.5 and 1.0



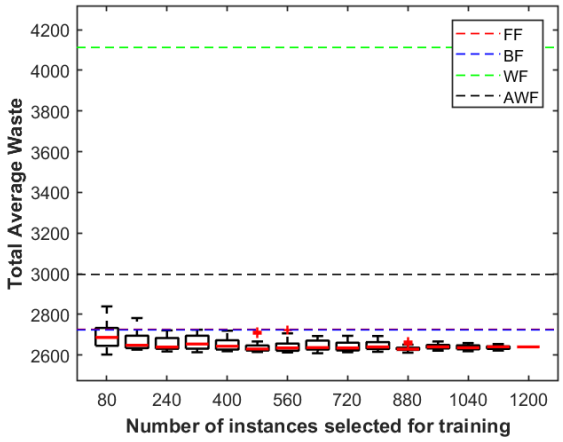
(b) 0.5 and 1.5



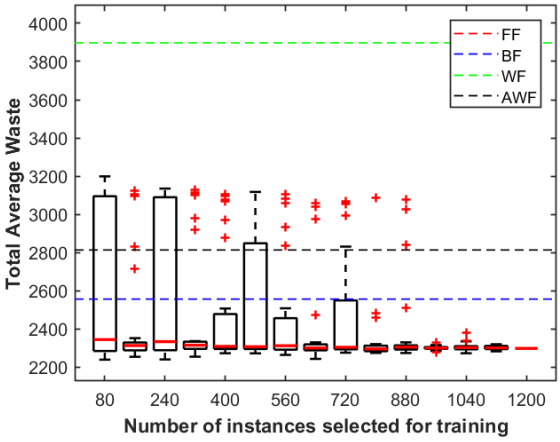
(c) 0.5 and 2.0



(d) 1.0 and 1.5

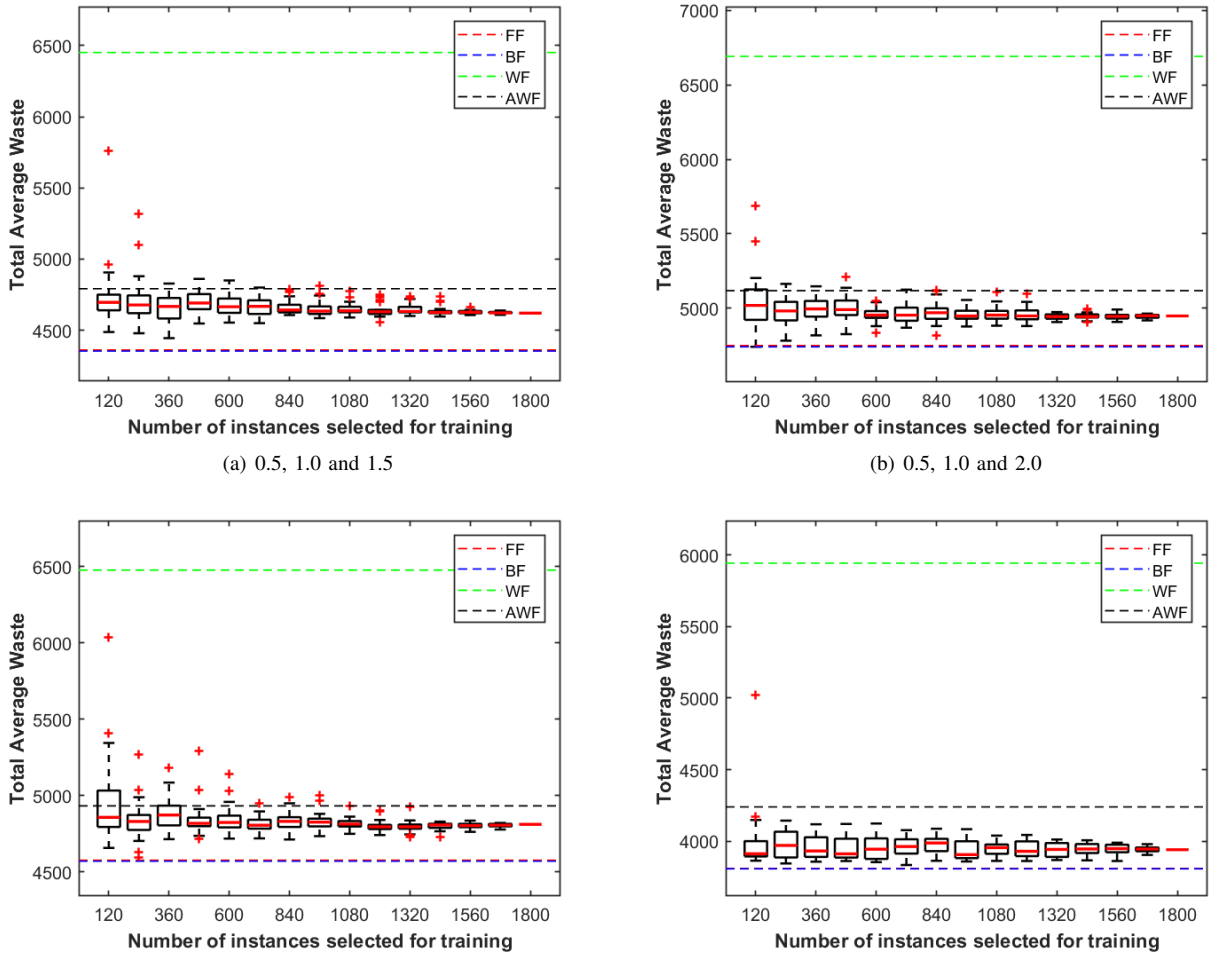


(e) 1.0 and 2.0



(f) 1.5 and 2.0

Fig. 4. Performance of low-level heuristics and of hyper-heuristics, over the whole set of generated instances (Advanced tests - 2 levels). Each box shows the performance of 30 different hyper-heuristics. Each plot shows data for instances tailored to a given combination of Δ_d. Note: The FF and BF heuristics perform quite similarly, so their lines tend to overlap.



(a) 0.5, 1.0 and 1.5

(b) 0.5, 1.0 and 2.0

(c) 0.5, 1.5 and 2.0

(d) 1.0, 1.5, and 2.0

Fig. 5. Performance of low-level heuristics and of hyper-heuristics, over the whole set of generated instances (Advanced tests - 3 levels). Each box shows the performance of 30 different hyper-heuristics. Each plot shows data for instances tailored to a given combination of Δ_d. Note: The FF and BF heuristics perform quite similarly, so their lines tend to overlap.

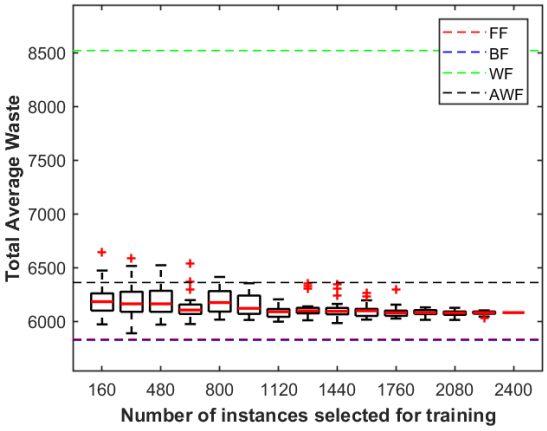


Fig. 6. Performance of low-level heuristics and of hyper-heuristics, over the whole set of generated instances (Advanced tests - 4 levels). Each box shows the performance of 30 different hyper-heuristics. Note: The FF and BF heuristics perform quite similarly, so their lines tend to overlap.

After analyzing our data, we can conclude that our proposed approach is feasible. Although, it depends on the considered Δ_d values. For example, in the simplest case using $\Delta_d = 0.5$ rendered it impossible to outperform the low-level heuristics. Conversely, using the highest value (i.e. $\Delta_d = 2.0$) led to a reduction of the accumulated average waste of over 12% with respect to the best heuristic.

Adding more Δ_d levels increased the complexity of the model. Each new level added four rules to the selection hyper-heuristic (one per heuristic). Such complexity exhibited an interesting interaction with the performance. Our analysis indicates that as the number of rules increases, instances with higher Δ_d values are required. For example, when considering two Δ_d values (i.e. eight rules) only two scenarios yielded hyper-heuristics that outperformed the low-level solvers. These scenarios represented the ones using the highest Δ_d levels (1.0 and 2.0, and 1.5 and 2.0). Even though they both achieved a satisfactory median performance, the latter achieved the biggest performance gain (10%). Even so, more complex

scenarios (with three and four delta levels) did not yield a satisfactory result. We believe this happens because such models (with 12 and 16 rules) require higher delta values. Also, we consider important to note that, even though we improved the results given by low-level heuristics, there is still a performance gap with regards to a synthetic Oracle. This may be due to the nature of the evolved instances and results could improve if using a different set of features. However, we leave such work for a future study since it goes beyond the scope of this manuscript.

With respect to the number of instances used for building a valid hyper-heuristic, we can conclude that our approach does not require too many instances. However, this works as long as the Δ_d levels are good enough. This can be seen in the cases where a single level was used. Only $\Delta_d = 0.5$ inhibited good hyper-heuristics from being created. Also, using all available instances is not always better. We achieved hyper-heuristics with a few number of instances that outperformed the one with all of them. However, such performance was not as stable since other repetitions of the same test yielded worse hyper-heuristics. This is due to the many combinations that arise from randomly selecting a small subset from a big set (e.g. 40 instances out of 600). Also, bear in mind that instance generation considered the performance of low-level heuristics and not their feature values. So, there is no guarantee that instances of the same nature are clustered. In spite of the variability, in most cases all hyper-heuristics outperformed low-level heuristics. Hence, we deem this approach as a feasible one and invite the reader to experiment with it.

Several paths lie ahead for future work. One of them is the inclusion of an analysis over a broad set of instances available in the literature and over imbalanced sets. The former would focus on analyzing if the information extracted from the instances is good at generalizing the domain. The latter would seek a better understanding about how performance changes when the number of representative instances is different for each solver. Another path leads to the analysis about how performance is affected when instances are biased towards a single solver. Throughout this work we already got a glimpse about it since two low-level heuristics (BF and FF) outperformed the other ones (WF and AWF). But, more experiments are required to validate it since our tests considered the same number of instances for each solver. We also observed that using higher Δ_d values seem to improve the gain achieved with the hyper-heuristic, so this should be studied in a third path. A final path we deem worthy of study is the effect of using sets with different number of items, bin capacity, and max length per item.

ACKNOWLEDGEMENT

This research was partially supported by CONACyT Basic Science Projects under grants 241461 and 287479, and ITESM Research Group with Strategic Focus in Intelligent Systems.

REFERENCES

[1] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: a survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, pp. 1695–1724, 12 2013.

[2] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, 2008.

[3] N. Pillay and R. Qu, *Hyper-Heuristics: Theory and Applications*. Natural Computing Series, Cham: Springer International Publishing, 2018.

[4] J. C. Ortiz-Bayliss, H. Terashima-Marín, and S. E. Conant-Pablos, "Learning vector quantization for variable ordering in constraint satisfaction problems," *Pattern Recognition Letters*, vol. 34, pp. 423–432, 3 2013.

[5] T. N. Ferreira, J. A. P. Lima, A. Strickler, J. N. Kuk, S. R. Vergilio, and A. Pozo, "Hyper-Heuristic Based Product Selection for Software Product Line Testing," *IEEE Computational Intelligence Magazine*, vol. 12, pp. 34–45, 4 2017.

[6] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 3 2003.

[7] I. Amaya, J. C. Ortiz-Bayliss, A. E. Gutierrez-Rodriguez, H. Terashima-Marin, and C. A. C. Coello, "Improving hyper-heuristic performance through feature transformation," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2614–2621, IEEE, 6 2017.

[8] I. Amaya, J. C. Ortiz-Bayliss, A. Rosales-Pérez, A. E. Gutiérrez-Rodríguez, S. E. Conant-Pablos, H. Terashima-Marín, and C. A. Coello Coello, "Enhancing Selection Hyper-Heuristics via Feature Transformations," *IEEE Computational Intelligence Magazine*, vol. 13, no. 2, pp. 30–41, 2018.

[9] J. E. Beasley, "OR-Library: Distributing Test Problems by Electronic Mail," *The Journal of the Operational Research Society*, vol. 41, no. 11, pp. 1069–1072, 1990.

[10] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter, "MIPLIB 2010," *Mathematical Programming Computation*, vol. 3, no. 2, pp. 103–163, 2011.

[11] S. Martello, D. Pisinger, and D. Vigo, "The three-dimensional bin packing problem," *Operations Research*, vol. 48, no. 2, pp. 256–267, 2000.

[12] D. Pisinger, "Where are the hard knapsack problems?," *Computers & Operations Research*, vol. 32, pp. 2271–2284, 9 2005.

[13] K. B. Petursson and T. P. Runarsson, "An evolutionary approach to the discovery of hybrid branching rules for mixed integer solvers," *Proceedings - 2015 IEEE Symposium Series on Computational Intelligence, SSCI 2015*, pp. 1436–1443, 2016.

[14] J. I. van Hemert, "Evolving binary constraint satisfaction problem instances that are difficult to solve," in *Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC'03)*, pp. 1267–1273, IEEE Press, 2003.

[15] J. I. van Hemert, "Evolving combinatorial problem instances that are difficult to solve," *Evolutionary Computation*, vol. 14, no. 4, pp. 433–462, 2006.

[16] K. Smith-Miles, J. van Hemert, and X. Lim, "Understanding TSP Difficulty by Learning from Evolved Instances," in *Learning and Intelligent Optimization* (C. Blum and R. Battiti, eds.), vol. 6073 of *Lecture Notes in Computer Science*, pp. 266–280, Springer Berlin Heidelberg, 2010.

[17] K. Smith-Miles and J. van Hemert, "Discovering the suitability of optimisation algorithms by learning from evolved instances," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 2, pp. 87–104, 2011.

[18] I. Amaya, J. C. Ortiz-Bayliss, S. E. Conant-Pablos, H. Terashima-Marín, and C. A. Coello Coello, "Tailoring Instances of the 1D Bin Packing Problem for Assessing Strengths and Weaknesses of Its Solvers," in *Parallel Problem Solving from Nature PPSN XV* (A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, and D. Whitley, eds.), vol. 11101 of *Lecture Notes in Computer Science*, pp. 373–384, Cham: Springer International Publishing, 2018.

[19] L. F. Plata-González, I. Amaya, J. C. Ortiz-Bayliss, S. E. Conant-Pablos, H. Terashima-Marín, and C. A. Coello Coello, "Evolutionary-based tailoring of synthetic instances for the Knapsack problem," *Soft Computing*, 2 2019.