Influence of Instance Size on Selection Hyper-Heuristics for Job Shop Scheduling Problems

Fernando Garza-Santisteban, Jorge M. Cruz-Duarte, Ivan Amaya, José Carlos Ortiz-Bayliss, Santiago Enrique Conant-Pablos, and Hugo Terashima-Marín
Department of Computation, Instituto Tecnologico de Monterrey, Monterrey (NL), Mexico
E-mails: A00918788@itesm.mx, {jorge.cruz,iamaya2,jcobayliss,sconant,terashima}@tec.mx

***Abstract*—Hyper-heuristics stand as a novel tool that combines low-level heuristics into robust solvers. However, training cost is a drawback that hinders their applicability. In this work, we analyze the effect of training with different problem sizes, to determine whether an effective simplification can be made. We train selection hyper-heuristics for the Job Shop Scheduling problem through Simulated Annealing (SA). Results from preliminary experiments suggest that the aforementioned simplification is feasible. To better understand such an effect, we carry out experiments training on two different instance sizes, 5×5 and 15×15, while testing on instances of size 15×15. Our data show that hyper-heuristics trained in small-sized instances perform similarly to those trained in larger problems. Thus, we discuss a possible explanation for this effect.**

***Keywords*—Job Shop Scheduling; Hyper-heuristic; Simulated Annealing; Combinatorial Optimization; Instance size.**

I. Introduction

Job Shop Scheduling (JSS) is a kind of combinatorial optimization problem. A JSS arises whenever it is required to schedule a series of machines and jobs, subject to some restrictions, and with a defined sequence of operations and processing times. Although its description is simple, the JSS problem is quite challenging to handle in real-world applications since it is NP-hard. This optimization problem is inherent to any industrial process, where it is imperative to produce a job plan (*i.e.,* a schedule) for resources and machines. Therefore, appropriate and efficient methods to solve the JSS problem, and its multiple variations, have been highly demanded since 1956 [1]. For example, some authors have tackled the Dynamic Flexible JSS problem, where there is uncertainty in the processing times [2], [3], [4]. Others have considered the Deterministic No-Wait JSS version, whose main goal is to find a schedule that minimizes the makespan (the time at which the last activity of the whole schedule finishes) [5], [6], [7]. This type of problem has been addressed through several solution techniques. One of them has been to develop dispatching rules (or low-level heuristics) that can be used to construct or modify a schedule. Some examples include the dispatching rules proposed by Blackstone *et al.* in [8], and the *Shifting Bottleneck* procedure employed by Adams *et al.* in [9]. These methods are computationally efficient, but the quality of their solutions is uncertain.

One way of surmounting such a drawback is to use more robust search techniques. There are several works following this path, including strategies such as Tabu Search [10], [11] and Guided Local Search with Branch-and-Bound [12]. However, there are also approaches based on Genetic Algorithms [13], [14], as well as on Genetic Search [15] and Genetic Programming [16]. Of course, this kind of strategies also include hybrids [17] and other approaches [18], [19], [20].

Nonetheless, the avenue mentioned above has a noticeable handicap: the algorithm needs to run from scratch for every problem in the dataset, which increases the computational cost. But, a new way of solving this kind of problems has recently emerged: hyper-heuristics. The idea with hyper-heuristics (HHs) is to provide robustness to the solver without excessively increasing computational cost. To achieve such a goal, hyper-heuristics combine the strengths of different low-level strategies by learning when to use each one. For instance, Chaurasia *et al.* proposed a HH based on Evolutionary Algorithms and a guided heuristic for JSS instances [21]. Similarly, Garza-Santisteban *et al.* studied the feasibility of using Simulated Annealing (SA) to train HHs for the JSS problem [22]. In fact, they noticed that there seemed to be an effect associating feature values and instance sizes, but were unable to delve deeper into the issue. Such combination of heuristics has also been explored following the idea of strictly using information about instance features, instead of using a combination of features and performance data [23]. Other studies have presented HHs based on evolutionary approaches [24], [25], [26] and have also explored the effect of transforming features [27], [28].

Notwithstanding, it is clear that competent strategies have been proposed in the literature to solve JSS problems. But they are exposed to be implemented in instances with a size different to which they were conceived (or designed) for. In real applications, the number of resources and machines varies. Thus, instance sizes are seldom preserved. To the best of our knowledge, there is no evidence about how the performance of a search strategy is affected when it is trained in instances that differ in size to the ones used for testing.

So, this work aims at revealing how instance size impacts hyper-heuristic training, and if such a difference affects hyper-heuristic performance when tested on a set of larger problems. In addition, we propose a selection hyper-heuristic approach based on Simulated Annealing, using a defined set of dispatching rules as heuristics. It is worth mentioning that our method

differs from the ones in literature because the latter employ evolutionary algorithms to evolve heuristics directly.

This manuscript is organized as follows: Section II presents the fundamental concepts related to the Job Shop Scheduling problem. Section III details the proposed strategy. Section IV describes the procedure carried out. Section V presents and discusses the resulting data. Finally, Section VI summarizes our most relevant findings and outlines future works.

II. Problem Definition

Let $\{j_1, j_2, ..., j_n\}$ be a set of n jobs given on a set of m machines $\{m_1, m_2, ..., m_m\}$, with the order sequence for each job on the set of machines. Each job has to be processed sequentially on one or more machines, having different processing times for each case. Such problems can be defined as tasks or activities $a_{i,k}$ for each job j_i at machine m_k. Thus, a job j_i comprises a set of at most m activities $\{a_{i,1}, a_{i,2}, ..., a_{i,m}\}$, where the processing time $t_{a_{i,k}}$ of the activity $a_{i,k}$ is different for each machine m_k. Therefore, the JSS problem consists in arranging each job j_i at each machine m_k to generate a timetable that minimizes the makespan. The following restrictions are also considered: (i) Each machine can process only one activity at a time; (ii) No activity can be scheduled to start before the preceding one ends; (iii) If any activity starts at a machine, it has to finish before another one can be scheduled on the same machine; and (iv) Any activity is always scheduled at the earliest possible time (the no-wait policy).

III. Solution Approach

Low-level heuristics are useful to perform an action based on certain information about the problem while keeping computational costs low. Another approach is to use a high-level solver that finds the best heuristic for a given problem state. With a JSS problem, a state S_t is defined as the specific schedule at any time t, considering the set of activities that have not been scheduled yet, and is represented by a set of features. Hence, this high-level solver provides a specific low-level heuristic (h) to be used at a given state S_t of the scheduling process. In this work, such a high-level solver refers to a constructive selection hyper-heuristic (HH).

A. Hyper-heuristics

Our hyper-heuristic model can be defined as a rule collection $\{r_1, r_2, \ldots, r_q\}$ that triggers predefined actions (heuristics to apply). Each rule is composed by an ordered feature set $\{f_{i,1}, f_{i,2}, ..., f_{i,q}\}$ that represents its corresponding activation condition. Our HH operates by calculating feature values of a current problem state S_t and its Euclidean distance to each defined rule. Then, the heuristic associated to the rule that minimizes such a distance is applied to the problem state.

The training process of a HH (see Section III-G for more details) implies that, in theory, the rules must replicate the best heuristic for a specific problem state. This statement implies several assumptions, such as:

1) The feature set used in the condition of a HH can properly describe distinct states of the problem.
2) Since a meta-heuristic, say Simulated Annealing (SA), generates the parameters for each HH in a perturbative randomized fashion, it can generate HHs with a set of rules sufficiently spread throughout the feature domain and identify the conditions that favor each heuristic.
3) When solving a problem, heuristics produce results sufficiently different from each other such that rules in the HH can capture these differences.

Now, let us suppose that such assumptions do not hold. Then, similar feature values can be rendered when solving an instance with each heuristic, throughout the whole search procedure. Therefore, all heuristics follow similar paths, which lead them to barely different solutions. It is then evident that disregarding the prior assumptions negatively impacts the meta-heuristic ability to produce a good hyper-heuristic.

Hence, it becomes paramount to determine how diverse such paths can be. One way of doing so is to measure how many solution paths remain available after a decision is taken. Let u_i^h be the number of distinct jobs available for the set of all heuristics (H) when heuristic h assigns an activity at step i. Also, let p_i^h be the number of pending jobs at step i when solving with heuristic h. Then, the average ratio u_i^h/p_i^h for every h in H shows job diversity across the solution of an instance. The smaller the measure, the less diverse paths are for that step. Conversely, the larger the measure, the more diverse they are. Figure 1 shows the result of conducting the former analysis on a 15×15 instance. It is clear that the diversity of solution paths tends to increase as the process of solving the instance goes forward.

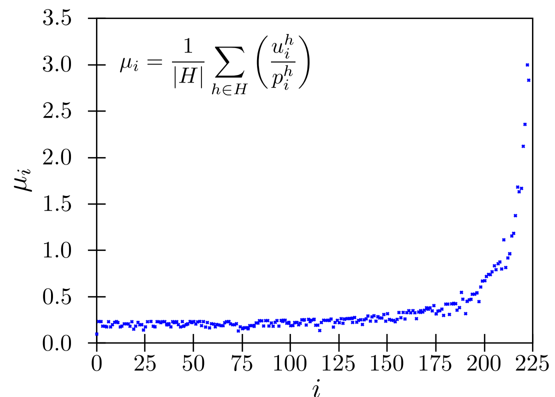

Fig. 1. Average μ_i of u_i^h/p_i^h for all $h \in H$ over a instance of size 15×15.

Note that u_i^h/p_i^h is different than the ratio of unique and pending jobs. Should they be equivalent, the measure would range between $[0, 1]$. Instead, what we are measuring here is the impact on heuristic diversity whenever a heuristic tries to solve the problem.

This result implies that the decision of scheduling a job greatly restricts other possibilities until almost the end of the solution process. Hence, the possibility of achieving a better result is also restricted. The former affects the training process

of the HH. During the first stages of the solution process, possible solution paths are small relative to the final phase of the solution. This directly affects assumption (2): no matter how good the training process is, the extent to which the HH will produce a better solution is greatly limited by the heuristic it chooses to apply at the first steps of the solving process.

B. Instances

In this work, the instances used for training are generated by following the approach described by Taillard [29] for creating random scheduling problems. The main characteristic of this generator is that it produces a random distribution of numbers for the machines and times of each job. Instances have the following features: fixed processing times, no set-up times, and no due dates nor release dates. For testing purposes, we have used the instances generated and published by Taillard [29]. Specifically, instances can be of size $n \times m$, where n is the number of jobs and m is the number of machines. Thus, the larger the instance size, the more jobs and machines it has. We use test sets with instances of size 15×15 to assess the performance of hyper-heuristics. Training sets contain instances of either size 5×5 or size 15×15, depending on the configuration of the experiment, and were generated accordingly.

C. Heuristics

For a set of activities $U_a = \{a_{1,1}, a_{1,2}, ..., a_{n,m}\}$, let U_t and U_s be the list of pending activities and of already scheduled ones, respectively. Also, let $t_{a_{i,k}}$ be the processing time of activity $a_{i,k}$ for job j_i at machine m_k. A heuristic returns an activity $a_{i,k} \in U_t$ following a specific rule r_q. In case that a rule produces more than one activity, a random activity is chosen. Table I summarizes the heuristics considered in this work, which depend either the pending (U_t) or scheduled (U_s) activities.

TABLE I
HEURISTICS CONSIDERED IN THIS WORK.

Heuristic	Definition
RND	Select a random activity $a_{i,k}$ from U_a that complies with precedence and insert it into the first available time.
SPT	Select $a_{i,k} \in U_t$ that corresponds to the shorstest processing time $t_{a_{i,k}} = \min\{t_{a_{1,1}}, ..., t_{a_{n,m}}\}$.
LPT	Select $a_{i,k} \in U_t$ that corresponds to the longest processing time $t_{a_{i,k}} = \max\{t_{a_{1,1}}, ..., t_{a_{n,m}}\}$.
MRT	Select $a_{i,k} \in U_t$ such that i corresponds to the job with the most remaining time in the set.
MLM	Select $a_{i,k} \in U_s$ such that k corresponds to the machine with the most load of activities in terms of processing time.
LLM	Select $a_{i,k} \in U_s$ such that k corresponds to the machine with the least load of activities in terms of processing time.
EST	Select $a_{i,j} \in U_t$ that can be scheduled at the earliest possible time.

D. Problem State Features

Two kinds of features are used to describe the state of a JSS problem, during its solving procedure. One set describes the schedule (solution), while the other one depicts the problem state (remaining jobs to be scheduled). A total of six features are employed, as Table II details.

TABLE II
FEATURES USED IN THIS WORK TO REPRESENT THE STATE OF A JSS PROBLEM.

Feature	Description	Related to
APT	Ratio between the sums of processing times of the already processed activities and the whole list of activities.	Solution
DPT	Ratio between the standard deviation and the mean of processing times from the scheduled activities.	Solution
SLACK	Ratio between the amount of unused machine time in the whole schedule and the current make-span.	Solution
DNPT	Ratio between the standard deviation and the mean processing times from the pending activities.	Problem
NAPT	Complement of APT.	Solution
NJT	Amount of processing times normalized per job divided by the number of pending jobs.	Problem

E. Objective Function

An objective function ($of : H \rightarrow \mathbb{R}$) is implemented to relate performances of the hyper-heuristic and the best single heuristic (for the same instance), which is accumulated over all the instances. The main reason for this is that their difference can represent the performance gain. Hence, the objective function $of(\text{HH})$ is defined as

$$of(\text{HH}) = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{C_{s_i}^{\text{HH}}}{C_{b_i}} - 1 \right) \tag{1}$$

where $C_{s_i}^{\text{HH}}$ and C_{b_i} are the makespan achieved by the hyper-heuristic and the best makespan achieved by the heuristics (the Oracle) on the i-th instance, respectively. n is the number of instances that each candidate hyper-heuristic seeks to solve.

F. Oracle

The Oracle is a Utopian strategy capable of selecting the best heuristic for each and every instance. Hence, it can be synthetically built by selecting the best solutions given by the set of heuristics. But, this also makes it unfeasible from a practical point of view as it implies running all solvers for each problem that requires solving. Thus, it represents a brute-force approach. Even so, it is feasible for using it as a reference value for comparison purposes. An illustrative example is now discussed. Table III contains synthetic data for four heuristics (h_1, h_2, h_3 and h_4), two instances, and assuming that lower values are better. In the first instance, h_4 is the best heuristic. So, the Oracle yields 80. In the second one, h_3 is the best heuristic and, therefore, the Oracle yields 100. By analyzing all the instances it becomes evident that the Oracle performs better than any single heuristic. Thus, it represents a lower bound for heuristics.

TABLE III
EXAMPLE OF ORACLE CALCULATION BY ASSUMING THAT LOWER VALUES ARE BETTER AND CONSIDERING FOUR HEURISTICS (h_1, h_2, h_3 AND h_4).

Instance	h_1	h_2	h_3	h_4	Oracle
1	100	150	200	80	80 (h_4)
2	120	140	100	200	100 (h_3)
Total	220	290	300	280	180

G. Hyper-heuristic Trainer

Training a hyper-heuristic for solving the JSS problem implies iterating over the feature values and the action of each rule, until finding suitable values. In this work, we use the well-known Simulated Annealing (SA) meta-heuristic to perform such a process [30]. Thence, the output of SA is the hyper-heuristic that minimizes the fitness function, evaluated over the training instances of the problem. It is worth mentioning that we chose SA since we wanted to explore the performance of a meta-heuristic that allowed worsening the best solution at some iterations.

Simulated Annealing requires some tuning parameters and basic operations that must be defined. One of the most distinctive steps of SA involves the Metropolis criterion for calculating the Acceptance Probability AP. In this case, such a probability is calculated for a couple of hyper-heuristics HH_1 and HH_2, as shown in (2). Here, T is the current temperature of the cooling process emulated by SA, and $of(\text{HH})$ is the objective function (or fitness) defined for the JSS problem in (1). Pseudocode 1 describes the SA implementation for training a hyper-heuristic.

$$AP(\text{HH}_1, \text{HH}_2) = \exp\left(-\frac{of(\text{HH}_2) - of(\text{HH}_1)}{T}\right) \quad (2)$$

It is important to remark that a neighbor hyper-heuristic HH_n (Line 6 in Pseudocode 1) is generated by randomly choosing among three different modifications of the current hyper-heuristic HH (Figure 2):

- *Adding a new rule*—the HH size is increased by one rule, including a new action with random feature values.
- *Removing a random rule*—the HH size is decreased by one rule, eliminating one randomly selected rule from HH. This option is omitted if current HH has one rule.
- *Perturbating a feature*—the HH size is unmodified, but features are randomly selected from an also randomly selected rule and replaced with new random values.

IV. METHODOLOGY

This work is aimed at assessing how the instance size of a JSS problem and the number of iterations performed by the optimizer affect the performance of a hyper-heuristic. Experiments were divided into three stages, as Table IV details. Simulated Annealing (SA) was implemented as optimizer. All the experiments (training and testing procedures) were run 30 times, for guaranteeing statistical significance, on a machine with an 8-core Intel Xeon Scalable processor at 2.7 GHz.

Pseudocode 1 Simulated Annealing used for HH training.

Require: An initial temperature value $T_0 \in \mathbb{R}_+$, a cooling rate $c \in]0,1[$, a temperature threshold $T_t \in]0, T_0[$.
Ensure: HH_*.
1: Set a random number generator $u \sim \mathcal{U}(0,1)$.
2: Set the temperature to its initial value $T = T_0$.
3: Generate an initial hyper-heuristic HH.
4: Set the best hyper-heuristic $\text{HH}_* = \text{HH}$.
5: **while** The system is not cool $(T > T_t)$ **do**
6: Create a neighbor of HH, HH_n, by modifying HH.
7: Calculate fitness $of(\text{HH})$ and $of(\text{HH}_n)$ with (1).
8: **if** $AP(\text{HH}, \text{HH}_n) > u$, **then** $\text{HH} \leftarrow \text{HH}_n$, **end if**
9: **if** $of(\text{HH}) < of(\text{HH}_*)$, **then** $\text{HH}_* = \text{HH}$, **end if**
10: Update temperature $T = T(1 - c)$.
11: **end while**

Moreover, three sets of instances with the characteristics specified in Table V were generated according to Taillard [29]. It is noteworthy that the testing set (Set 3) comprises unseen instances. We now detail each experimental stage.

TABLE IV
SUMMARY OF THE EXPERIMENTS PERFORMED.

Stage	Train on	Test on	Iterate up to	Repeat
A	Set 1 and 2	Set 3	100	30 times
B	Set 1	Set 3	10, 100 and 1000	30 times
C	Set 1 and 2	Set 3	10, 100 and 1000	30 times

TABLE V
SETS OF JSS PROBLEMS EMPLOYED IN THIS WORK.

	Instances	Size	Purpose
Set 1	30	15×15	Training
Set 2	30	5×5	Training
Set 3	5	15×15	Testing

Stage A: Instance size—A hyper-heuristic is trained with Set 1 and tested on Set 3 (cf. Table V). Subsequently, another hyper-heuristic is trained with Set 2 and also tested on Set 3. To achieve these HHs, 100 iterations were employed.

Stage B: Number of iterations—A hyper-heuristic is trained and tested with Sets 1 and 3, respectively. Both sets comprise instances of the same size, but the number of iterations for training changes, *i.e.,* 10, 100 and 1000.

Stage C: Instance size and number of iterations—Hyper-heuristics are trained with Sets 1 and 2 (instances of different sizes), and for different number of iterations: 10, 100 and 100. Each hyper-heuristic is tested on Set 3 (Table V).

It is important to highlight that the number of iterations refer to those carried out with SA, which consists on building neighboring solutions and sometimes allowing the preservation of a worse solution. We believe the selected values are good enough for analyzing the behavior at different phases of the solving process. Also, we are limited to 1000 iterations due to computational resources.

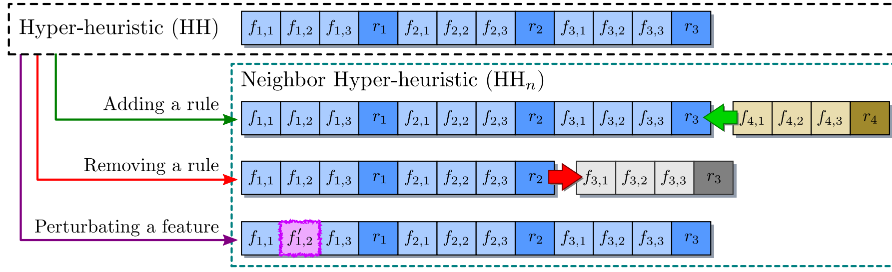

Fig. 2. Possible operations that Simulated Annealing (SA) performs to generate a neighbor hyper-heuristic (HH$_n$) from a current one (HH): adding a rule, removing a rule, and perturbating a feature within an existing rule.

V. Results and Discussion

For the sake of clarity, this section is structured according to the experimental stages described in the Methodology.

A. Instance size

Figure 3 presents the performance of hyper-heuristics trained on Sets 1 (left) and 2 (right) when testing them on Set 3. It is easy to infer that both distributions are right-skewed. Nonetheless, data range for Set 1 (big instances) is tighter than for Set 2 (small instances). Similarly, and though median values are quite similar to each other, it is slightly higher for small instances. Hence, it seems that the benefit of using larger instances may stem from the perspective of stability. Even so, both scenarios tend to yield the same performance. In this way, a Wilcoxon's statistical test with $\alpha = 0.05$ was executed, and it revealed that there is no significant difference between both approaches.

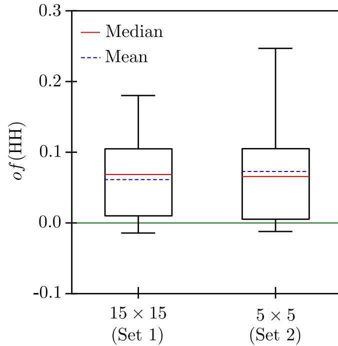

Fig. 3. Performance on Set 3 (15×15) of hyper-heuristics trained with Sets 1 and 2, and repeating 30 times.

This is an interesting and rather unexpected behaviour. It would be natural that a hyper-heuristic trained on simpler problems performs poorly on complex ones. But, it would seem as if smaller instances contribute to a better (or at least equal) distinction of heuristics during the training process. To illustrate that, we compared feature values for instances of different sizes. Let us consider, for the sake of simplicity, features related to the problem state (*i.e.,* DNPT and NJT, cf. Table II) prior to creating the solution, as Figure 4 displays. Instances of all three sizes seem to revolve around the point $(0.55, 0.50)$. Moreover, smaller instances are the most spread out, which supports the aforementioned effect.

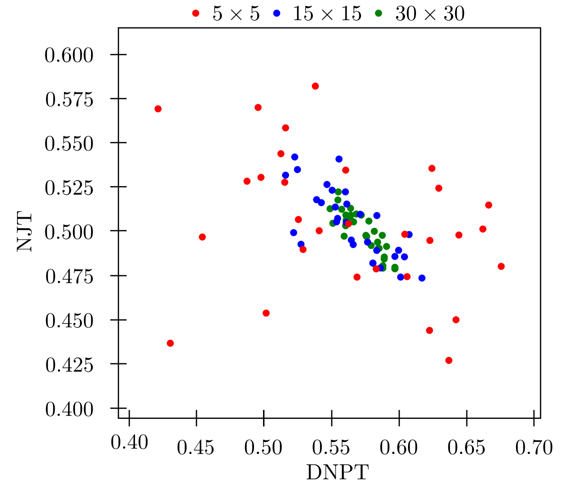

Fig. 4. Instance distribution based on features DNPT and NJT with sets of 30 instances for three different sizes.

B. Number of iterations

Using a larger number of iterations yields better results for both, training and testing, as Figure 5 shows. This is an expected behavior. Thence, a small number of iterations yields poorly-performing hyper-heuristics with low stability. Nonetheless, it is noteworthy that migrating from 100 to 1000 iterations allows hyper-heuristics to outperform the Oracle, *i.e.,* $of(\mathrm{HH}) < 0$, in most experiments.

C. Instance size and number of iterations

Figure 6 exhibits the results of six cases of study obtained by combining three values for the number of iterations (10, 100, and 1000) and two training sets (Set 1 and 2). All the hereby generated hyper-heuristics were tested on Set 3. As expected, the best results were achieved with the largest number of iterations (1000) and the instances most similar to the testing set, *i.e.,* Set 1 with size 15×15. However, we observed that performance values from hyper-heuristics trained on Set 1 with 100 iterations and on Set 2 with 1000 iterations seemed similar. Such a fact is also noticed for their testing results. Indeed, the statistical test mentioned in Section V-A was

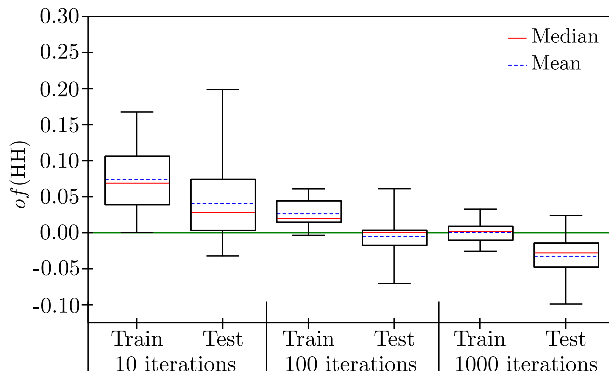

Fig. 5. Performance achieved by hyper-heuristics trained on Set 1 (15×15) for 10, 100 and 1000 iterations, and repeating 30 times. Data are shown for training and testing sets.

executed for those data and revealed no significant statistical difference between them (Table VI). Hence, both training procedures are equivalent and tend to generalize equally well.

TABLE VI
WILCOXON'S SIGNED RANK SUM TEST FOR HYPER-HEURISTICS TRAINED ON SET 1 WITH 100 ITERATIONS AND ON SET 2 WITH 1000 ITERATIONS, AND TESTED ON SET 3.

Parameter	Training	Testing
α	0.05	0.05
Tails	2	2
W	207	152
W_{crit}	120	120
Result	No difference	No difference

In addition to the statistical inferences, we analyzed the computing time for each run of every training scenario, as Table VII details. By working with small instances (Set 2), we can reduce training time in more than half (56.7%) without significantly jeopardizing hyper-heuristic performance. In fact, the median performance on the testing set with both scenarios is virtually the same (with a difference of 0.01). Actually, one of them is basically zero. Hence, they match the performance level of the Oracle (see Section III-F). Therefore, it is possible to train hyper-heuristics with sets composed of small instances to speed up the training process. Conversely, this time gain could be employed to feed more training data to the model, striving to achieve more robust hyper-heuristics.

VI. Conclusions and Future Work

In this work, we analyzed the effect of instance size on the performance of hyper-heuristics (HHs) implemented for Job Shop Scheduling (JSS) problems. For that purpose, we used Simulated Annealing (SA) to train hyper-heuristics. We conducted several experiments on two sets of 30 instances with sizes of 15×15 and 5×5, and each experiment was repeated 30 times for ensuring statistical significance. In addition, we also considered three different values for the number of iterations, 10, 100 and 1000, to figure out the influence of iterations while training. Subsequently, the generated hyper-heuristics were tested on a set with five unseen instances of size 15×15. All training instances used in this work were generated according to the scheme reported by Taillard in [29]. Testing instances were selected directly from the aforementioned work, as they represent the most difficult ones among the instances generated by Taillard [29]. Results allowed us to outline several interesting facts that are summarized next.

TABLE VII
TIME REQUIRED TO TRAIN A SINGLE HYPER-HEURISTIC WITH DIFFERENT INSTANCE SIZES AND NUMBERS OF ITERATIONS, USING THE PROCESSOR DESCRIBED IN THE METHODOLOGY. VALUES IN BOLD REPRESENT CONFIGURATIONS WITH EQUIVALENT PERFORMANCE WHEN TRAINING ON PROBLEMS OF DIFFERENT SIZE.

Instances (size)	Iterations	Time
Set 1 (15×15)	10	3 min
	100	**15 min**
	1000	2.5 h
Set 2 (5×5)	10	6 s
	100	45 s
	1000	**6.5 min**

When considering a fixed number of iterations, training with large problems has no relevant benefits. Similar performance levels were achieved for hyper-heuristics trained with small problems (5×5) and with larger ones (15×15), as shown in Figure 3. This assertion was corroborated with a Wilcoxon's ranked sum.

As expected, the best performing hyper-heuristics were achieved when training on similar problems (*i.e.,* 15×15), and for the highest number of iterations (1000), cf. Figure 6. Most of these HHs outperformed their corresponding synthetic Oracle. But, the cost of training a single hyper-heuristic was too high (about 2.5 hours per replica). We noticed that this drawback can be diminished by reducing the number of iterations, lowering training time to 15 minutes per replica. But, hyper-heuristic performance is jeopardized. However, another approach is to keep the number of iterations (1000) but to train in smaller problems (*i.e.,* 5×5). This way, performance is also worsened (to a similar degree), but a single HH can now be achieved in 6.5 minutes. In fact, a Wilcoxon's ranked sum test ($\alpha = 0.05$) revealed that there is no statistically significant difference between the performance of both approaches. Thus, it represents an advantageous opportunity. We believe that the reason for this behavior rests in the distribution of features, as was shown for the problem instances considered in this work. Thence, we consider that by using a set of training instances with small problems is a worthwhile approach to training hyper-heuristics. In doing so, the training process may be faster. Conversely, the time gain may be used for increasing the number of training samples or the number of iterations, thus leading to better solvers.

Furthermore, we realized that the instance size directly impacts how a hyper-heuristic reads the state of a problem, which affects the training process. Data suggest a link between such a phenomenon and the features used to represent

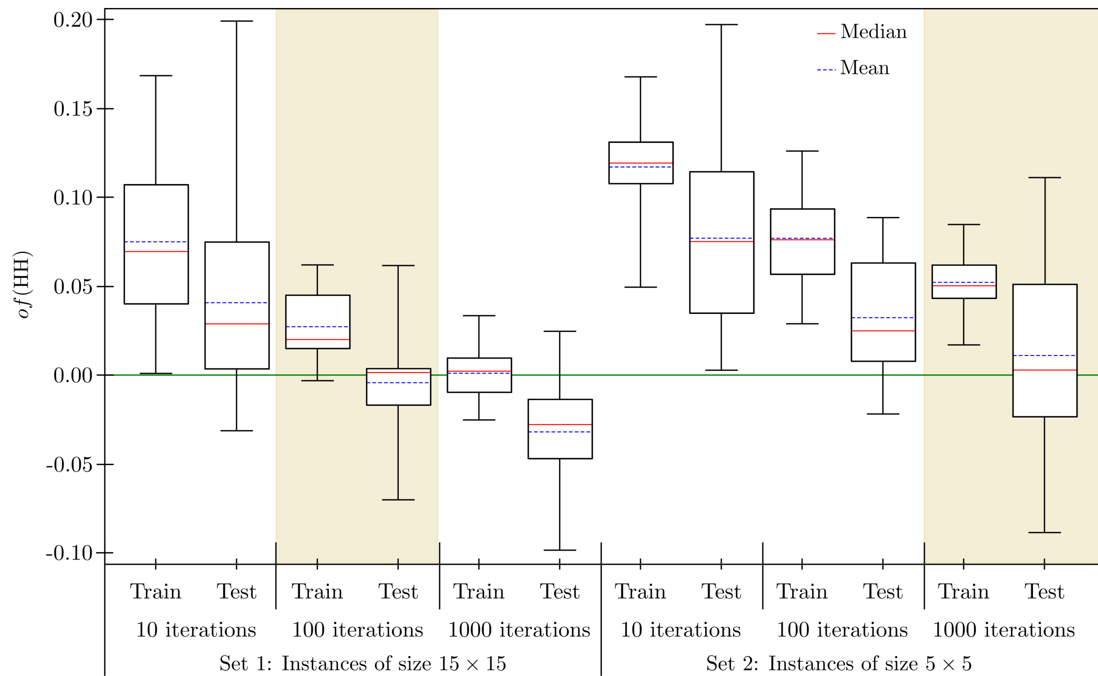

Fig. 6. Performance achieved by hyper-heuristics trained with 30 instances of size 5×5 and 15×15 for 10, 100 and 1000 iterations, and repeating 30 times. Highlighted boxes represent two configurations with similar performance.

the problem. Then, when mapping the feature space, small instances become more disperse than larger ones. This makes it easier for the optimizer to find a feasible set of rules and actions, *i.e.,* a hyper-heuristic. However, this effect may not be exclusive to JSS problems, so we plan to explore other domains while focusing on this phenomenon. Finally, this work naturally branches into two additional lines worthy of future studies. The first one is to continue exploring the influence of instance size over hyper-heuristic training; the other one is to study the implications of using a different set of features to characterize the problem state.

ACKNOWLEDGEMENTS

This research was supported by ITESM Research Group with Strategic Focus on Intelligent Systems.

REFERENCES

[1] J. Zhang, G. Ding, Y. Zou, S. Qin, and J. Fu, "Review of job shop scheduling research and its new perspectives under industry 4.0," *Journal of Intelligent Manufacturing*, vol. 30, no. 4, pp. 1809–1830, Apr 2019.

[2] D. Yska, Y. Mei, and M. Zhang, "Feature construction in genetic programming hyper-heuristic for dynamic flexible job shop scheduling," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '18*, no. 1. New York, New York, USA: ACM Press, 2018, pp. 149–150.

[3] J. Lin, "Backtracking search based hyper-heuristic for the flexible job-shop scheduling problem with fuzzy processing time," *Engineering Applications of Artificial Intelligence*, vol. 77, no. October 2016, pp. 186–196, 2019.

[4] Y. Zhou, J.-J. Yang, and L.-Y. Zheng, "Hyper-Heuristic Coevolution of Machine Assignment and Job Sequencing Rules for Multi-Objective Dynamic Flexible Job Shop Scheduling," *IEEE Access*, vol. 7, pp. 68–88, 2019.

[5] K. Miyashita, "Job-shop scheduling with genetic programming," in *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 505–512.

[6] A. Masood, Y. Mei, G. Chen, and M. Zhang, "Many-objective genetic programming for job-shop scheduling," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. Vancouver, Canada: IEEE, July 2016, pp. 209–216.

[7] W. Bozejko, A. Gnatowski, J. Pempera, and M. Wodecki, "Parallel tabu search for the cyclic job shop scheduling problem," *Computers & Industrial Engineering*, vol. 113, pp. 512 – 524, 2017.

[8] J. H. Blackstone, D. T. Phillips, and G. Hogg, "A state-of-the-art survey of dispatching rules for manufacturing job shop operations," *International Journal of Production Research*, vol. 20, pp. 27–45, 01 1982.

[9] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management Science*, vol. 34, no. 3, pp. 391–401, 1988.

[10] E. Nowicki and C. Smutnicki, "A fast taboo search algorithm for the job shop problem," *Management science*, vol. 42, no. 6, pp. 797–813, 1996.

[11] P. Fattahi, M. Messi Bidgoli, and P. Samouei, "An improved tabu search algorithm for job shop scheduling problem trough hybrid solution representations," *Journal of Quality Engineering and Production Optimization*, vol. 3, no. 1, pp. 13–26, 2018.

[12] E. Balas and A. Vazacopoulos, "Guided local search with shifting bottleneck for job shop scheduling," *Management science*, vol. 44, no. 2, pp. 262–275, 1998.

[13] R. Q. dao-er ji and Y. Wang, "A new hybrid genetic algorithm for job shop scheduling problem," *Computers & Operations Research*, vol. 39, no. 10, pp. 2291 – 2299, 2012.

[14] L. Wang, J.-C. Cai, and M. Li, "An adaptive multi-population genetic algorithm for job-shop scheduling problem," *Advances in Manufacturing*, vol. 4, no. 2, pp. 142–149, 2016.

[15] S. Uckun, S. Bagchi, K. Kawamura, and Y. Miyabe, "Managing genetic search in job shop scheduling," *IEEE Intelligent Systems*, vol. 8, no. 5, pp. 15–24, 1993.

[16] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Genetic programming for job shop scheduling," in *Evolutionary and Swarm Intelligence Algorithms*. Cham, Switzerland: Springer, 2019, pp. 143–167.

[17] L. Wang and D.-Z. Zheng, "An effective hybrid optimization strategy for job-shop scheduling problems," *Computers & Operations Research*, vol. 28, no. 6, pp. 585–596, 2001.

[18] C. S. Chong, M. Y. H. Low, A. I. Sivakumar, and K. L. Gay, "A bee colony optimization algorithm to job shop scheduling," in *Proceedings of the 2006 Winter Simulation Conference*. Monterey,California: Winter Simulatrion Conference, Dec 2006, pp. 1954–1961.

[19] D. Sha and C.-Y. Hsu, "A hybrid particle swarm optimization for job shop scheduling problem," *Computers & Industrial Engineering*, vol. 51, no. 4, pp. 791–808, 2006.

[20] K.-L. Huang and C.-J. Liao, "Ant colony optimization combined with taboo search for the job shop scheduling problem," *Computers & operations research*, vol. 35, no. 4, pp. 1030–1046, 2008.

[21] S. N. Chaurasia, S. Sundar, D. Jung, H. M. Lee, and J. H. Kim, "An Evolutionary Algorithm Based Hyper-heuristic for the Job-Shop Scheduling Problem with No-Wait Constraint," in *Harmony Search and Nature Inspired Optimization Algorithms*. Singapore: Springer Singapore, 2019, vol. 741, pp. 249–257.

[22] F. Garza-Santisteban, R. Sanchez-Pamanes, L. A. Puente-Rodriguez, I. Amaya, J. C. Ortiz-Bayliss, S. Conant-Pablos, and H. Terashima-Marin, "A Simulated Annealing Hyper-heuristic for Job Shop Scheduling Problems," in *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, jun 2019, pp. 57–64. [Online]. Available: https://ieeexplore.ieee.org/document/8790296/

[23] I. Amaya, J. C. Ortiz-Bayliss, S. Conant-Pablos, and H. Terashima-Marin, "Hyper-heuristics Reversed: Learning to Combine Solvers by Evolving Instances," in *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, jun 2019, pp. 1790–1797. [Online]. Available: https://ieeexplore.ieee.org/document/8789928/

[24] E. Hart and K. Sim, "A hyper-heuristic ensemble method for static job-shop scheduling," *Evolutionary computation*, vol. 24, no. 4, pp. 609–635, 2016.

[25] J. Branke, T. Hildebrandt, and B. Scholz-Reiter, "Hyper-heuristic evolution of dispatching rules: A comparison of rule representations," *Evolutionary computation*, vol. 23, no. 2, pp. 249–277, 2015.

[26] V. Kumar *et al.*, "Integration of dispatch rules for jssp: A learning approach," in *Soft Computing: Theories and Applications*. Singapore: Springer, 2019, pp. 619–627.

[27] 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)*. IEEE, jun 2017, pp. 2614–2621. [Online]. Available: http://ieeexplore.ieee.org/document/7969623/

[28] 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.

[29] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, no. 2, pp. 278 – 285, 1993, project Management anf Scheduling.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.