IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 28, 2022, accepted April 8, 2022, date of publication April 22, 2022, date of current version April 29, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3169503

Beyond Hyper-Heuristics: A Squared Hyper-Heuristic Model for Solving Job Shop Scheduling Problems

ALONSO VELA, JORGE M. CRUZ-DUARTE, (Member, IEEE), JOSÉ CARLOS ORTIZ-BAYLISS, (Member, IEEE), AND IVAN AMAYA, (Senior Member, IEEE)

School of Engineering and Sciences, Tecnologico de Monterrey, Monterrey, Nuevo Leon 64849, Mexico

Corresponding author: Ivan Amaya (iamaya2@tec.mx)

This work was supported in part by the Consejo Nacional de Ciencia y Tecnología (CONACyT) Basic Science Project 287479, and in part by the Instituto Tecnológico y de Estudios Superiores de Monterrey (ITESM) Research Group with Strategic Focus in Intelligent Systems.

ABSTRACT Hyper-heuristics (HHs) stand as a relatively recent approach to solving optimization problems. There are different kinds of HHs. One of them deals with how low-level heuristics must be combined to deliver an improved solution to a set of problem instances. Literature commonly refers to them as *selection hyper-heuristics*. One of their advantages is that the strengths of each heuristic can be fused into a high-level solver. However, one of their drawbacks is that sometimes this generalization scheme does not suffice. Additionally, it is not easy to reuse these HHs since the model cannot be easily tweaked. So, in this work, we develop a hyper-heuristic model with an additional layer of generalization. The rationale behind it is to preserve the general structure of selecting an adequate solver for a particular situation but to use HHs instead of low-level heuristics. We call this model a Squared Hyper-Heuristic (SHH). To validate our proposal, we pursue a four-stage methodology that covers several testing scenarios. Our data reveal that, under proper conditions, our model can outperform the base HHs. Moreover, it is flexible enough to allow for an increased number of layers so that the complexity of the final model can be tuned. Additionally, different kinds of instances can be used to train each stage of the model, thus setting the groundwork for developing a transfer learning approach for hyper-heuristics.

INDEX TERMS Job shop scheduling problem, heuristics, hyper-heuristics, optimization, problem features, squared hyper-heuristics.

NOMENCLATURE

SYMBOLS

c	Machine makespan
$\mathfrak{F}$	Objective Function
F	Feature value
$\vec{g}$	Global velocity vector
H	Heuristic
j	Job
$\vec{l}$	Local velocity vector
m	Machine
M	Set of machines
N	Number
o	Operation
O	Set of operations
OP	Optimizer

SYMBOLS

p	Processing time
PD	Problem domain
PM	Performance metric
q	Iteration counter
s	Schedule
S	Set of schedules
SL	Set of available solvers
TR	Set of training instances
u	Unification Factor
$\mathcal{U}$	Uniform Distribution
$\vec{r}$	Random vector
$\vec{v}^t$	Current total velocity
$\vec{x}$	Position

The associate editor coordinating the review of this manuscript and approving it for publication was Sotirios Goudos.

X	Population (swarm of agents)
X_a	Neighborhood topology
$\vec{x}^t$	Current position

SYMBOLS

Δ	Performance gap
ϕ_1	Self confidence coefficient
ϕ_2	Global confidence coefficient
χ	Constriction factor

SUBSCRIPTS

a	Agent or particle
d	Dimension
e	Expanded value
f	Feature index
i	Element index (machine, heuristic, or number)
j	Job
k	Job index
l	Operation index
m	Machine
max	Maximal value
n	neighbor
o	Operation
r	Rule
tr	Training instance
W	Most expensive
$*$	Best value

ACRONYMS

COP	Combinatorial Optimization Problem
HH	Hyper-Heuristic
JSSP	Job Shop Scheduling Problem
LPA	Least Pending Activities
LPT	Longest Processing Time
MPA	Most Pending Activities
OSCOMBA	Operation Slots COMBined with processing times Amplified
OSRMA	Operation Slots Repeated Machines Amplified
PSO	Particle Swarm Optimization
SHH	Squared Hyper-heuristic
SPT	Shortest Processing Time
UPSO	Unified Particle Swarm Optimization

I. INTRODUCTION

Hyper-heuristics (HHs) stand as a contemporary method for tackling optimization problems [1]. Although there are different kinds of HHs, the overall idea is straightforward: to combine the strengths of a set of available solvers to improve one or more performance metrics. This usually allows finding better solutions than by using the solvers in a standalone fashion.

There are several ways to classify a HH. Some of the earliest approaches considered the nature of the heuristic (constructive or perturbative) and the way the available solvers are used (selection or generation) [2], [3]. Nowadays, researchers have developed a wider set of hyper-heuristics, and so it has become necessary to extend the existing criteria, as Drake *et al.* discuss [4]. Such elements include the kind of feedback used by the HH, *e.g.,* online or offline learning, and the nature of its parameters, *e.g.,* static or dynamic, among others.

Because of the plethora of hyper-heuristic models available in the literature, it is unfeasible to cover all variants in this brief introduction. Nonetheless, we feel it is essential to provide some recent examples. In terms of HHs with online learning capabilities one can find the works of Wei *et al.*, and those of Yu *et al.* and of Majeed and Naz. In the first one, the authors developed a model for planning paths in marine environments, being able to achieve good-quality solutions in real-time [5]. In the second one, the authors focused on controlling swarm robots and found that hyper-heuristics are a good choice for their decentralized control [6]. In the remaining approach, the authors combined this online learning capability with a memory of already solved solutions, achieving promising results in different clustering datasets [7]. In a somewhat related application, one finds the work of Adnan *et al.* where the authors considered a semi-supervised hyper-heuristic model for classification, outperforming state-of-the-art alternatives [8]. Finally, the work by Cui *et al.* dealing with multi-objective hyper-heuristics could also prove of interest to the reader since the authors develop a framework for scheduling flight deck operations [9].

Hyper-heuristics have been successfully applied to different combinatorial problems. For example, Zhong *et al.* tackled the generation of schedules for heat sinks in networks of wireless sensors [10]. Similarly, Toledo *et al.* targeted the Orienteering Problem with Hotel Selection, where an optimum tour including hotels and points of interest must be found [11]. Moreover, Bai *et al.* developed a general-purpose hyper-heuristic for solving bin packing and course timetabling problems [12]. In all of these cases, the authors achieved better performance levels than when using human-made choices or existing alternatives.

Notwithstanding, researchers have also explored the continuous domain, although mostly focused on traditional benchmarks. For example, Miranda *et al.* fused selection and generation hyper-heuristics to improve upon the solution of continuous benchmark problems [13]. Additionally, Cruz-Duarte *et al.* explored how to automatically develop new metaheuristics through a selection hyper-heuristic approach [14], [15]. The works of Choong *et al.* and Sabar *et al.* also focus on the design of hyper-heuristics. The first one uses reinforcement learning for automatically finding a heuristic selection approach, achieving results comparable to the top HHs found in literature [16]. The latter uses a genetic programming technique for deciding the next heuristic that shall be used [17].

Another challenging task where HHs have proved useful is to solve Job Shop Scheduling Problems (JSSPs). This problem has multiple applications to real-life scenarios,

especially those related to manufacturing [18]–[23]. So, it is natural that the JSSP has been tackled with different methods. Some efforts include genetic programming [24]. Others include those based on data mining and machine learning [25], [26]. However, yet another group focuses on using hyper-heuristics. The work by Pickardt *et al.*, where the authors developed a two-stage hyper-heuristic for developing and assigning dispatching rules, is an exemplar of this category [27]. Another relevant work is the one from Hart and Sim, where the authors used genetic programming for creating heuristics [28]. In their proposal, the model distributes instances to a set of solvers, thus pursuing a divide and conquer approach. Moreover, Lara-Cardenas *et al.* developed a reward-based model so that rules can keep on evolving [29]. Other recent works include the analysis of problems with scenario-based processing times [30] and with more complex constraints [31], as well as problems closer to reality [32].

Despite the existing hyper-heuristic models and their success when tackling JSSPs, it is customary to find problem instances that are not properly solved by the same hyper-heuristic. This comment is supported by previous work [33], where HHs containing low-level solvers were not able to properly solve different types of instances at once. We suggest we can alleviate this by improving upon the generalization capabilities of HHs. However, to the best of our knowledge, such a higher-order model that selects HHs is a field yet unexplored. So, in this work, we aim to fill such a knowledge gap by proposing a more general selection hyper-heuristic model, which can accommodate other hyper-heuristics. We then exploit a previously proposed instance generator that allows for the creation of instances where simple heuristics excel or fail [34]. The idea of such an instance generator was adopted from [35]. In this way, our work has a two-fold contribution:

1) The proposal of a more general hyper-heuristic model that can accommodate other high-level solvers and its extensive testing across different sets of instances
2) A complexity analysis about how the computational cost of the proposed model grows

Consequently, our work strives to answer the following research questions:

1) How can a higher-order hyper-heuristic model be implemented?
2) What is the performance of such a higher-order hyper-heuristic when solving JSSPs?
3) How does the computational complexity grow for such a model?

The rest of this document is organized as follows. Section II presents the core definitions and ideas that represent the foundations of our work. Then, Section III describes our proposed approach and its components. Section IV and Section V lay out the steps we followed in our work and the resulting data, respectively. Section VI wraps up this work by providing some insights and by declaring some paths for pursuing in future works.

II. FUNDAMENTALS

A. JOB SHOP SCHEDULING PROBLEM

The Job Shop Scheduling Problem (JSSP) stands as a relevant Combinatorial Optimization Problem (COP) with a plethora of applications [18]. Even so, most of them relate to the manufacturing industry [19]–[23]. In simple terms, solving a JSSP requires to organize (schedule) the execution of some tasks (jobs) to improve processing performance.

The traditional JSSP seeks to minimize the time taken to fulfill all tasks, known as the makespan. However, it is also customary to find JSSP variants in the literature [36]. One of them is the cyclic flexible JSSP, where jobs can be performed by more than one machine and which considers the effect of the number of assigned workers [37]. Similarly, one may find variants that consider external factors like breakdowns and others [38]. Another common variant is the just-in-time JSSP, where early and late completions are penalized [39]. Of course, there are lots more variants, but it is unfeasible to cover them all here [40]–[43].

The JSSP is an NP-hard problem in which using exact methods is out of the question [29]. Because of this, approximated solvers have evolved and adapted. This process has included the development of decision rules for scheduling jobs, as with the approach followed by Nguyen *et al.* where the authors used information from completed schedules through a Genetic Programming approach [24]. Similarly, other authors have used data mining techniques for developing such rules [26], [44].

A similar avenue befalls the development of simple heuristics that build feasible solutions quickly. Alas, these do not guarantee optimality. Nevertheless, they usually perform properly when tackling sets of instances with specific natures. Thus, it is only reasonable to try and combine them. One way of doing so is to select a solver from a predefined pool to deal with a complete problem instance, which is commonly known as an algorithm portfolio [45], [46]. However, this approach bounds the best performance to that of the best available heuristic. So, it is customary to create a synthetic oracle based on the best solutions obtained from isolated heuristics.

Another approach pursues a similar path. However, instead of using a single solver per instance, it combines them. This is usually achieved by limiting heuristic usage to a single solution step so that a new heuristic selection can be carried out. Literature depicts this strategy as a selection Hyper-Heuristic (HH). However, bear in mind that there are different kinds of hyper-heuristics [3], [4]. Moreover, their usage is not limited to COPs, as they can also be applied in continuous optimization problems [47]. Hyper-heuristics have been useful in tackling COPs, including the JSSP [38], [48]–[50].

As aforementioned, there are several kinds of JSSPs [51]. In this work, we restrict ourselves to the traditional JSSP. In such version, the problem assumes that operations are ordered and that they must be performed in a fixed machine. Moreover, only one operation of a given job can be performed simultaneously. We also adopt the definition of a JSSP that considers an operation as the building block of a job [34].

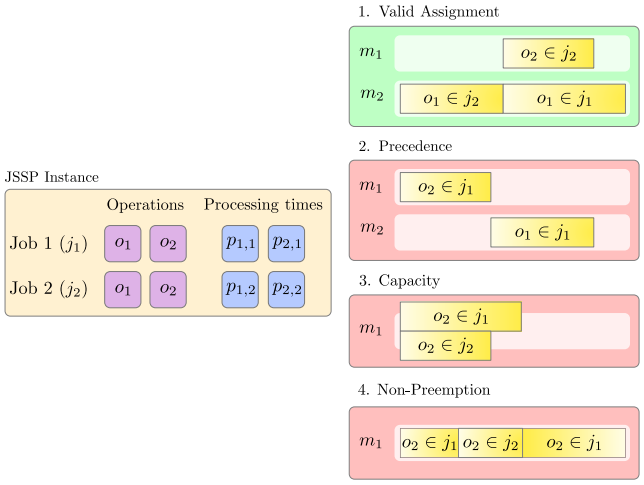


FIGURE 1. Constraints associated with the Job Shop Scheduling Problem (JSSP) when assigning operations. 1. Valid assignment. 2. The assignment does not comply with the precedence of operations for the given job. 3. Invalid assignment where one machine is processing two operations simultaneously. 4. The assignment violates the preemption constraint because one operation is split into two parts.

Besides, and to make things easier for the reader, we have summarized the main symbols required for dealing with a JSSP in Table 1.

TABLE 1. Relation of symbols associated with the JSSP and their meaning, based on the definitions coined in [34].

Symbol	Description
o_l	An operation from the set of operations $\mathcal{O}$ (also known as activity)
j_k	A job given by an ordered sequence of N_o operations
m_i	A machine from the set of machines $\mathcal{M}$ that executes operations
$p_{l,k}$	The processing time of operation l from j_k
s	A schedule from the space of feasible schedules $\mathcal{S}$ (complies with all constraints and process all jobs)
$c_{\max}(s)$	Makespan of schedule s (time taken to process all jobs)

Bear in mind that the traditional JSSP also includes some restrictions about machine usage [52]. Such restrictions are displayed in Figure 1. Briefly speaking, they imply that machines must respect the order of operations within a job (precedence), can only perform a single operation at a time (capacity), and cannot partially process operations (non-preemption).

B. UNIFIED PARTICLE SWARM OPTIMIZATION (UPSO)

UPSO is a popular variant of the traditional Particle Swarm Optimization (PSO) algorithm. The PSO algorithm was pioneered by Eberhart and Kennedy back in 1995, while UPSO was presented almost ten years later by Parsopolous and Vrahatis [53]. Both versions take advantage of the swarm intelligence and have found diverse applications, especially in engineering [54]. We detail the logic behind UPSO in Pseudocode 1.

As with the original proposal, UPSO defines a population $X \in \mathbb{R}^{N_d \times N_a}$ of N_a search agents (particles) that scout a problem with N_d dimensions while interacting with each other. However, unlike PSO, UPSO divides the particles into neighborhoods and provides a parameter for tuning the balance between exploration and exploitation. This is achieved by defining a global ($\vec{g}_a$) and a local ($\vec{l}_a$) velocity for each particle a, which are obtained using Equation (1) and Equation (2).

$$\vec{g}_a = \chi\left(\vec{v}_a^t + \phi_1\vec{r}_1 \odot (\vec{x}_{a,*} - \vec{x}_a^t) + \phi_2\vec{r}_2 \odot (\vec{x}_* - \vec{x}_a^t)\right) \quad (1)$$

$$\vec{l}_a = \chi\left(\vec{v}_a^t + \phi_1\vec{r}_3 \odot (\vec{x}_{a,*} - \vec{x}_a^t) + \phi_2\vec{r}_4 \odot (\vec{x}_{n,*} - \vec{x}_a^t)\right) \quad (2)$$

Here, ϕ_1 and ϕ_2 are the self and global confidence coefficients, respectively; and $\vec{r}_i \in \mathbb{R}^{N_d}$ are i.i.d. random vectors with uniform distribution between zero and one, *i.e.*, $\vec{r}_i \ni r_{i,j} \sim \mathcal{U}(0,1), \forall\, i \in \{1,2,3,4\}$. N_a represents the total number of search agents, which is commonly known as the population size. Additionally, $\odot$ is the Hadamard product and χ is the constriction factor. The latter should be defined as shown in Equation (3), based on the recommendations from Clerc [55], where $\kappa = 1$ is customary.

$$\chi = \frac{2\kappa}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|}, \text{ with } \phi = \phi_1 + \phi_2. \quad (3)$$

The total velocity of the particle $\vec{v}_a^t$ is calculated as a weighted sum between $\vec{g}_a$ and $\vec{l}_a$, using an unification factor $u \in [0,1]$, as shown:

$$\vec{v}_a^{t+1} = (1-u)\vec{l}_a + u\vec{g}_a \quad (4)$$

Additionally, each a-th particle also has a current position, given by $\vec{x}_a^t \in \mathbb{R}^{N_d}$. Consequently, such a position is updated with:

$$\vec{x}_a^{t+1} = \vec{x}_a^t + \vec{v}_a^{t+1} \quad (5)$$

Moreover, there are three elements of interest: $\vec{x}_{a,*}$, $\vec{x}_{n,*}$, and $\vec{x}_*$. These represent the best position found by each particle, by each neighborhood, and by the entire swarm, respectively, which are selected based on the fitness given by the objective function $\mathfrak{F} : \mathbb{R}^{N_d} \to \mathbb{R}$. These elements can be calculated using Equations (6) to (8).

$$\vec{x}_{a,*} = \text{arginf}\{\mathfrak{F}(\vec{x}_{a,*}), \mathfrak{F}(\vec{x}_a^t)\} \quad (6)$$

$$\vec{x}_{n,*} = \text{arginf}\{\mathfrak{F}(\vec{x}_{n,*}), \cup_{\vec{x}\in X_a}\mathfrak{F}(\vec{x})\} \quad (7)$$

$$\vec{x}_* = \text{arginf}\{\mathfrak{F}(\vec{x}_{n,*}), \cup_{\vec{x}\in X}\mathfrak{F}(\vec{x})\} \quad (8)$$

Although there are several neighborhood topologies [56], in this work we have considered the most basic one for the sake of simplicity. So, each neighborhood follows a ring topology with three-nodes, such that $X_a = \{\vec{x}_{a-1}^t, \vec{x}_a^t, \vec{x}_{a+1}^t\}$.

C. HYPER-HEURISTICS

The term hyper-heuristic (HH) was coined in 1997 to describe a protocol that combines several artificial intelligence methods [1]. Some years later, the term was used to describe "heuristics to choose heuristics" [57]. This approach has

Pseudocode 1 Unified Particle Swarm Optimization (UPSO) Algorithm

Input: Objective function $\mathfrak{F}$, neighborhood topology X_a, population size N_a, parameter κ, self confidence ϕ_1, global confidence ϕ_2, unification factor u, and stopping criteria: $q_{\max}$ and others (if they exist)

Output: Best solution $\vec{x}_*$

Make $q \leftarrow 0$ and determine χ with (3)
Initialize the population positions $\vec{x}_a^t$ and velocities $\vec{v}_a^t$, $\forall\, a \in \{1, \ldots, N_a\}$
Find $\vec{x}_{a,*}$, $\vec{x}_{n,*}$, and $\vec{x}_*$ using (6), (7), and (8)
while $(q \leq q_{\max})$ & (additional stopping criterion is not met) **do**
    Determine $\vec{g}_a$ and $\vec{l}_a$ with (1) and (2), $\forall\, a \in \{1, \ldots, N_a\}$
    Update $\vec{x}_a^t$ and $\vec{v}_a^t$ with (4) and (5), $\forall\, a \in \{1, \ldots, N_a\}$
    Update $\vec{x}_{a,*}$, $\vec{x}_{n,*}$, and $\vec{x}_*$ using (6), (7), and (8)
    $q \leftarrow q + 1$
end while

been used to tackle various problems through different strategies. For example, Sim and Hart described an immune-inspired hyper-heuristic system that produces new heuristics for the bin-packing and job-shop scheduling problems [58]. Similarly, Nguyen *et al.* proposed a hyper-heuristic powered by genetic programming for three combinatorial and optimization problems [59]: Max-SAT, one dimensional bin packing and permutation flow shop. Sabar and Kendall presented a Monte Carlo tree-search hyper-heuristic that evolves heuristics [60]. They also tackled the problems considered by Nguyen *et al.* and included two additional ones: travelling salesman and personnel scheduling.

We can classify hyper-heuristics based on different criteria [2]. However, it is customary to consider the kind of solver developed by the hyper-heuristic. On the one hand, a hyper-heuristic may create an entirely new solver. On the other hand, it may select one from the available solvers. These kinds are known as generation and selection hyper-heuristics, respectively.

Another common approach for cataloguing hyper-heuristics is to consider the kind of heuristics that they have access to. Such heuristics can be either perturbative or constructive, depending on whether they alter an existing solution or create a solution from scratch. Bear in mind that such classification criteria are not exclusive. So, one may traditionally find the following kinds of hyper-heuristics: generation-constructive, generation-perturbative, selection-perturbative, and selection-constructive. More recently, Burke *et al.* added more classification criteria, including one related to the kind of feedback used by the model [61]. In this sense, one may find three alternatives:

Online Learning. The model keeps on learning with every instance it solves. This means that it solves and learns simultaneously.

Offline Learning. The model uses a set of instances for optimizing its parameters in order to maximize performance. Afterward, it can be used to solve new instances but the model is no longer updated.

No Learning. No feedback is used whatsoever to improve the model.

Although it would be interesting to deepen into all the existing models, doing so is beyond the scope of this manuscript. So, to avoid overextending this section, we stop our discussion here and invite the reader to consult [61].

III. PROPOSED APPROACH

In this section, we provide an overview of our proposed model and its implementation. First, we explain the structure and general idea of this Squared Hyper-Heuristic (SHH) model and a possible classification scheme. Then, we explain the training process of a SHH. Finally, we analyze the computational complexity of adding the extra decision layer. Bear in mind that, in this work, we use instances from the literature and instances we generated. In both cases, we assume that the number of operations of each job equals the number of available machines. We also assume that each hyper-heuristic available to the SHH has been previously trained (more details in Section IV).

A. OVERVIEW OF THE PROPOSED APPROACH

Our idea for the proposed hyper-heuristic (HH) model is straightforward: to have a HH that selects among other HHs when solving a problem. Of course, this implies that one must train the base HHs so that they select among low-level heuristics. Hence, our proposed model sits at a higher level of abstraction than traditional hyper-heuristics. For this reason, we opted for the name of squared hyper-heuristic (SHH).

Our approach still selects a solver from a pool of available alternatives, as with traditional selection hyper-heuristics. Nonetheless, such a pool is filled with already trained selection hyper-heuristics instead of the traditional approach that uses low-level heuristics. The selected hyper-heuristic ultimately chooses a low-level heuristic, but the selection changes with each step of the solution. Thus, our model retains the traditional behavior of a selection hyper-heuristic. Hence, its name.

Figure 2 exemplifies our idea. Here, the proposed model (SHH) can select among two traditional HHs (HH_1 and HH_2) when solving a problem instance. The SHH follows a traditional rule-based scheme for such a selection. Moreover, the objective remains: to optimize the values of each rule

so that the solver selects the best approach for the current conditions.

For simplicity, we use the same optimization strategy (*i.e.,* UPSO) to train SHHs and HHs. We selected this metaheuristic because of different reasons. For starters, it organizes search agents (particles) into neighborhoods and so the search is not focused on a single direction. Moreover, the information of such neighborhoods is combined through a couple of constants that allow balancing the local and global search behaviours. Additionally, we have worked with UPSO in the past [34] and so it was readily available. Notwithstanding, bear in mind that UPSO could be replaced by any other metaheuristic or optimization algorithm, with virtually no changes to the proposed model.

The training process consists of minimizing an objective function. For this work, said function is related to the makespan of the set of instances. To this end, we used UPSO to adjust, via the training phase, the parameters of the hyper-heuristic model, until reaching a set of values that offer good performance, as we explain in III-D. Moreover, since HHs use heuristics for altering the problem states, each rule of the SHH ultimately translates into using a given heuristic. In this way, a region of influence for each heuristic can be defined for the SHH. These regions represent the combinations of feature values required for a given heuristic to be selected by the high-level solvers. So, with the proposed model, we might generate more complex maps than with traditional HHs. These can be achieved since the regions of influence derived from the SHH overlap the regions belonging to the base HHs, based on the rules from the former.

The right-most part of Figure 2 presents an example of two different maps using the same base HHs. Such HHs contain three rules each and share three low-level heuristics, represented as H_1, H_2, and H_3. As one may notice, one SHH leads to a more complex distribution of solvers (top right) by using two rules for selecting among the base hyper-heuristics. However, our model is not limited to combining all the available solvers, or providing complex regions of influence. If needed, it can disregard one or more solvers, providing simpler regions of influence, as we show in the bottom right part of Figure 2. Do note that heuristic H_3 does not appear in the final model.

As aforementioned, the actions taken through a SHH ultimately translate into heuristics. This hints at the idea that the proposed layer might be unnecessary. Notwithstanding, we believe that it allows finding more complex regions of influence more easily than training a HH that directly achieves these patterns. Additionally, it may prove fruitful to try and develop some translation algorithm in the future, seeking to transform a high-order hyper-heuristic model into one with a lower order.

Since our proposed model can use high-level solvers, we consider an attribute based on the kind of solver the SHH incorporates. Moreover, we attempt to build upon the classification given by Burke *et al.* [61]. So, we consider that a SHH may be either *closed* or *flexible*. The former exclusively uses traditional hyper-heuristics. Moreover, this particular model selects a traditional hyper-heuristic to alter the problem instance, based on the current features values of said instance. The flexible SHH, theoretically, can access other kinds of solvers that fit the problem, *e.g.,* a single heuristic, a neural network, etc. Bear in mind that the neural network and the low-level heuristic are only indicated as illustrative examples of feasible solvers and they are not the only alternatives. Actually, the model can accommodate any kind of algorithm that could function as a solver for the SHH, *i.e.,* a solver that returns a modified version of a problem instance. Figure 3 shows an example of each category. This leads to the following categories:

1) Closed generation constructive
2) Closed generation perturbative
3) Closed selection perturbative
4) Closed selection constructive
5) Flexible generation constructive
6) Flexible generation perturbative
7) Flexible selection perturbative
8) Flexible selection constructive

Since it would be unfeasible to cover all categories in this work, we limit ourselves to analyzing the closed selection constructive squared hyper-heuristic model shown in Figure 3.

B. HEURISTICS CONSIDERED FOR THIS WORK

Throughout this work, we consider four different heuristics. Bear in mind that all heuristics select an activity that complies with problem restrictions. This means that they only select among upcoming activities for each job. We include two heuristics based on the processing times of the activities. While one of them focuses on scheduling jobs that take longer to complete (*i.e.,* Largest Processing Time, LPT), the other targets small jobs first (*i.e.,* Shortest Processing Time, SPT). Similarly, we include two heuristics that seek to schedule a whole job quickly or evenly. To achieve this, we target the number of pending activities for each job. Thus, heuristic MPA strives to follow a breadth-first approach since it selects the job with the Most Pending Activities. Conversely, heuristic LPA follows a depth-first approach, seeking to first complete jobs with the Least Pending Activities. It is essential to highlight that in the case of ties, *i.e.,* if two or more activities are valid options for the heuristic, the one belonging to the job with the lowest ID is selected.

C. FEATURES CONSIDERED FOR THIS WORK

In this work, we considered a total of five features. It is essential to highlight that these features are taken from the work of Mirshekarian *et al.* and that their definitions can become relatively complex [62]. We refer to them by a five-letter code associated with the first author and a number corresponding to its position within the original list of features. Bear in mind that we only consider those features related to the problem instance and unrelated to specific heuristics.

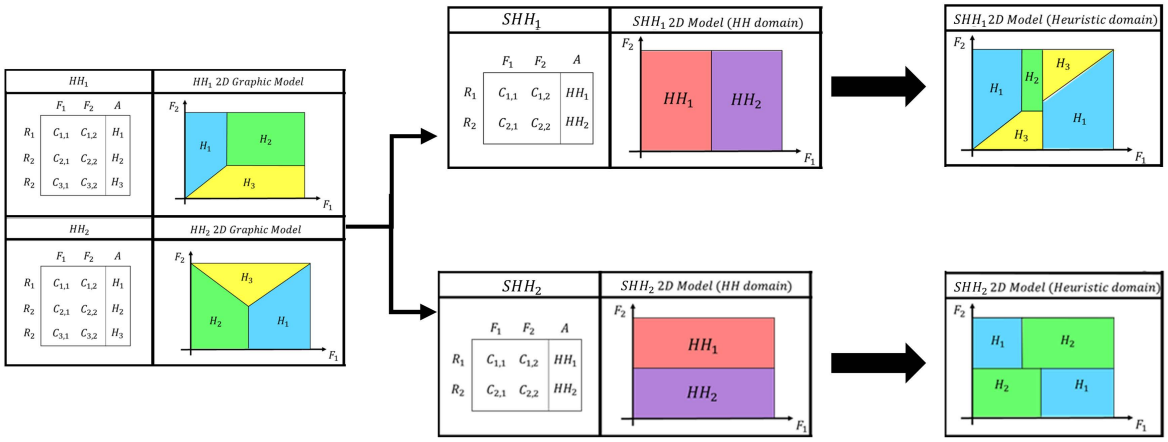


FIGURE 2. An example of two different squared hyper-heuristics that can be created with the same base solvers and the way heuristic selection changes.

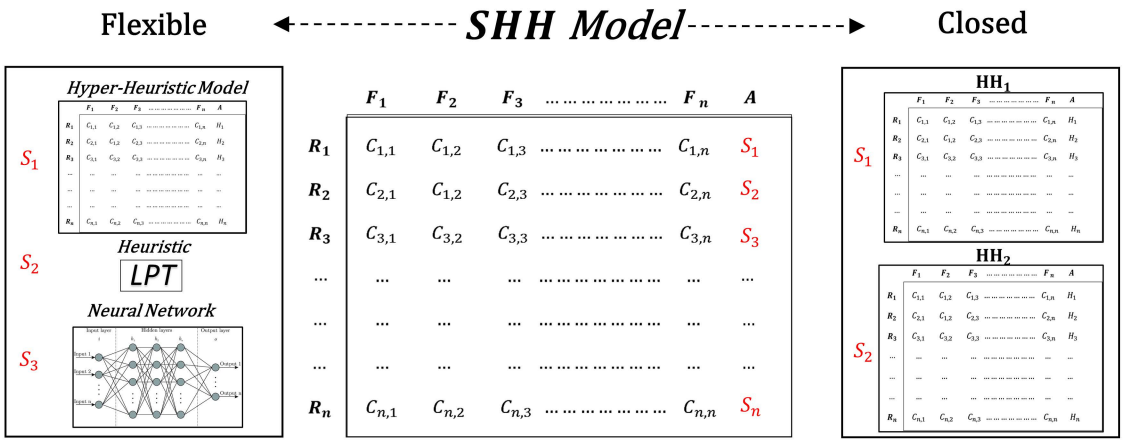


FIGURE 3. Overview of the Squared Hyper-Heuristic (SHH) model. The SHH evaluates the Job Shop Scheduling Problem features and selects a solver to perform the next action, based on the action defined in the closest rule. In a closed SHH (right) such actions are restricted to hyper-heuristic models. Conversely, a flexible SHH (left) has access to a variety of methods.

So, we do not necessarily use features with consecutive IDs. Similarly, we sought to select some of the most representative features the authors found, as long as they are unrelated to heuristics. The rationale behind this decision is to avoid biasing information towards a single heuristic. Additionally, some of these features use an expanded metric, calculated as shown in:

$$F_e = \frac{F_f(F_f + 1)}{2} \tag{9}$$

where F_e is the expanded metric, and F_f is the original feature value. Such features are thoroughly explained in previous works [34], [62]. Hence, for the sake of brevity we only provide an overview of them:

`Mirsh15`. Standard Deviation of Job Processing Times divided by the mean of Job Processing Times. A rather straightforward feature based on the accumulated processing times of all operations within each job.

`Mirsh29`. Standard Deviation of Machine Processing Times divided by mean of Machine Processing Times. Similar to `Mirsh15` but it considers processing times across machines instead of jobs.

`Mirsh95`. Mean of OSRMA divided by the number of machines. This feature relates the number of machines that are repeated in each operation (across jobs) and the total number of available machines.

`Mirsh222`. Mean of OSCOMBA divided by the product between the number of machines and the mean of Operation Processing Times. This feature relates upcoming activities within the instance with their processing times and with the number of times that each machine is used.

`Mirsh282`. Mean of Machine Load Voids Amplified divided by the number of machines. This feature analyzes machine usage but considering the number of zeros (voids) per operation. Such a zero is given by a machine that is not used at all in an operation.

D. MODEL TRAINING

Our training process for a SHH (Algorithm 2) requires several inputs, such as the number of rules N_r and the number of features N_f. Do note that this code can be generalized for a SHH model with more layers with few changes, but it escapes the scope of this work. Algorithm 2 assumes that one has access to a population-based optimizer for training the HH model. In this work, we use UPSO. Other inputs required for the process are the problem domain PD (JSSP in this case), the set of available solvers *SL* (*e.g.,* the set of HHs), the set of training instances *TR*, the optimizer OP, the performance metric PM for evaluating candidate solutions, and the number of search agents N_a. Do note that the trained SHH is the one that yields the best PM when solving all instances in *TR*, and it may or not correspond to the final value of a search agent, depending on OP. The algorithm begins by initializing the set of agents, *i.e.,* the candidate SHHs, with random values. These agents will explore the solution space based on the policies set by the optimizer. Then, the algorithm assesses the quality of each agent by evaluating its PM when solving *TR*. This implies using all SHHs to solve every training instance. A loop is then started, where the values of each SHH are updated based on the policy given by OP for up to $q_{\max}$ iterations. Note that the process may stop earlier if a stop criterion is reached, such as performance stagnation. Additionally, a record of the best agent is updated with each iteration, so that the best solver is returned to the user.

This process is quite similar to the training of a traditional HH. The only difference is that each search agent in our model represents a SHH and that whenever an action is used, the corresponding HH must be analyzed to select a heuristic ultimately. In this way, the process can be extended to higher orders. This requires, however, some kind of recursion for traversing the decision layers.

E. COMPLEXITY ANALYSIS

Before ending this section, we feel that it is important to provide a complexity analysis of our proposed approach (Algorithm 2). The initialization step has a multivariate linear complexity that depends on the number of rules N_r, the number of features N_f and the number of agents N_a, *i.e.,* $\mathcal{O}(N_a * N_f * N_r)$. The second step requires constant time, and so we can omit it.

The core of the complexity lies in the evaluation phase (steps 3 and 6), which requires solving all training instances ($N_{tr} = |TR|$) with each one of the N_a agents. Thus, agents and instances increase complexity in a linear fashion. Moreover, when evaluating an instance $tr \in TR$ we must calculate the current feature values. The time complexity for this operation is given by Equation (10), where $\mathcal{O}(F_f(tr))$ is the complexity of calculating a given feature.

$$\sum_{f=1}^{N_f} \mathcal{O}(F_f(tr)) \tag{10}$$

Afterward, we need to calculate the distances between each rule and the current feature values. The complexity of this step is given by Equation (11), where $K(N_f)$ is the complexity of calculating the chosen distance metric for one rule, which depends on the number of features.

$$\mathcal{O}(N_r * K(N_f)) \tag{11}$$

The closest rule must now be selected, which implies finding the minimum distance. This adds a linear complexity given by $\mathcal{O}(N_r)$. Now, we need to add the complexity of using the selected heuristic H for moving the solution one step further, *i.e.,* $\mathcal{O}(H)$. In this way, we complete one step of the solution process. So, we must multiply such complexity for the number of steps required to solve the problem, which is given by the number of jobs (N_j) and machines (N_m).

In this work, we use the Euclidean distance metric. So, the cost of the distance metric grows linearly w.r.t. N_f. Moreover, Equation (10) is bounded by the worst cost of all features, *i.e.,* $\mathcal{O}(F_W)$. Therefore, the complexity of evaluating the set of search agents grows as Equation (12) shows.

$$\mathcal{O}(N_{tr} * N_a * N_j * N_m * (F_W + N_r * N_f + N_r + H)) \tag{12}$$

We must now analyze the cost of the other operations within the loop. Step five has a complexity that is directly linked to the OP, *i.e.,* $\mathcal{O}(\text{OP})$. Step six has the same cost that we just analyzed for step three, *i.e.,* Equation (12). The next step performs a search within a list, and so its complexity grows as $\mathcal{O}(N_a)$. The remaining steps represent fixed costs, and so their complexity can be disregarded. Moreover, each loop requires evaluating the stopping criteria, which can be associated with a constant value. Additionally, we need to include the effect of the number of training iterations. In the worst-case scenario, the process will be repeated for up to $q_{\max}$ iterations. So, the final complexity of the training stage rises as shown in Equation (13).

$$\mathcal{O}\left(q_{\max}\left(\text{OP} + N_{tr} * N_a * N_j * N_m * (F_W + N_r * N_f + H)\right)\right) \tag{13}$$

Let us now see what happens when we train a SHH. For the sake of simplicity, we analyze a closed SHH model, *i.e.,* where all solvers are already trained HHs, since it is the kind of solver we use in this work. Moreover, we assume that all HHs have the same number of features N_f and rules N_r.

The main difference w.r.t. the previous analysis rests on the way a solver given by the selected rule is used. So, the addition of this layer only alters the cost of steps 3 and 6 from Algorithm 2. Since the SHH selects a trained HH, we must use the current feature values to calculate new distance metrics to the corresponding set of rules. This has the same computing cost from the previous analysis (minus the current feature values since we already have them). In this way, we finally arrive at a heuristic that can be used to tackle the problem. Hence, the complexity of evaluating a set of SHHs is represented by Equation (14).

$$\mathcal{O}(N_{tr} * N_a * N_j * N_m * (F_W + 2 * N_r * N_f + 2 * N_r + H)) \tag{14}$$

Pseudocode 2 Squared Hyper-Heuristic Training Algorithm

Input: Number of rules N_r, problem domain PD, solvers *SL*, training instances *TR*, optimizer OP, number of features N_f, performance metric PM, number of search agents N_a

Output: Best solution SHH_q

```
Initialize a set of N_a SHHs with N_r rules, N_f features, and a problem PD, using random values
Make q ← 0
Evaluate each SHH by calculating its PM value when solving TR
while (q ≤ q_max) & (additional stopping criterion is not met) do
    Update the values of each SHH using the policy given by OP
    Evaluate each SHH by calculating its PM value when solving TR
    SHH_q ← SHH with best PM value between the current set of SHHs and SHH_q
    q ← q + 1
end while
```

As one may notice, the only difference when evaluating a set of HHs or SHHs is a constant with the value of two, which is negligible for this kind of analysis. So, the overall computational complexity of training both models grow in the same fashion, *i.e.,* following Equation (13). Notwithstanding, and by recursion, a model with more layers would increase linearly as the number of layers go up.

IV. METHODOLOGY

This work explores the performance of the proposed SHH model through a four-stage methodology, as indicated in Figure 4. For doing so, we analyze the method's behavior when subject to different training and testing scenarios. For the sake of simplicity, we use the same optimization method, *i.e.,* UPSO, for training all HHs and SHHs.

Due to the amount of experiments, using a single machine will require way too much time. Therefore, we used four different machines across all experiments: a MacBook Pro 2019 (Intel Core i5-8257U @ 1.4 GHz with 8 GB RAM), two Dell Inc desktops (one with an Intel Core i5-6200U @ 2.30 GHz with 8 GB RAM, and one with an Intel Core i7-6700 @ 3.40 GHz with 16 GB RAM), and an Alienware desktop (Intel Core i7-8700 @ 3.20 GHz with 16 GB RAM). Since mixing the computational times reported by different machines could bias conclusions, we opt for not providing such data.

It is important to remember that the metaheuristic and the SHH model require different sets of parameters. However, our goal is to focus on the SHH model and not on how the metaheuristic parameters affect training performance. So, we seek to assess the functionality of our proposal for some arbitrarily selected parameters. However, we know that future work should analyze their effect to properly tune the model and improve its performance. We use the following fixed parameters, based on preliminary tests: $\phi_1 = 2.00$, $\phi_2 = 2.50$, and $u = 0.25$. Besides, we consider a fixed limit of 100 iterations for most tests and vary the number of particles between 15, 30, and 50 agents, as we detail below.

We use several sets of training and testing instances that we generate using a previously proposed instance generator that allows for a fixed performance gap (Δ) [34]. Additionally, we consider some randomly created instances and some publicly available instances that Taillard reported.[1] These represent larger instances that we mainly use to validate our findings. Table 2 details the sets of training instances and Table 3 contains the testing ones.

TABLE 2. Training sets used in this work with different performance gaps (Δ). Instances are equally distributed and they all have the same size (three jobs and four machines), unless otherwise stated.

ID	#Instances	Δ	Instance Description
TR01	15	N/A	Random
TR02	15	Free	LPTvsSPT
TR03	15	Free	SPTvsLPT
TR04	15	Free	MPAvsLPA
TR05	15	Free	LPAvsMPA
TR06	15	Free	AllvsMPA
TR07	30	20	LPTvsAll + SPTvsAll
TR08	30	20	MPAvsAll + LPAvsAll
TR09	60	20	LPTvsAll + SPTvsAll + MPAvsAll + LPAvsAll
TR10	30	20	LPTvsAll + MPAvsAll
TR11	30	20	SPTvsAll + LPAvsAll
TR12	30	20	LPTvsAll + LPAvsAll
TR13	30	20	SPTvsAll + MPAvsAll
TR14	100	N/A	Random
TR15	30	20	LPTvsAll + AllvsLPT
TR16	30	20	SPTvsAll + AllvsSPT
TR17	30	20	MPAvsAll + AllvsMPA
TR18	30	20	LPAvsAll + AllvsLPA
TR19	60	20	LPTvsAll (10) + SPTvsAll (20) + MPAvsAll (5) + LPAvsAll (25)
Tai1515	10	N/A	Taillard set (15 jobs, 15 machines)

Throughout all our tests, we use a performance metric (PM) given by the total makespan. This means that each time the model is tested (*e.g.,* during the training process) we sum the makespan achieved across all instances (*e.g.,* from the training set). We incorporate this metric because it is a straightforward way of obtaining information about the performance achieved across a set of instances while incurring in a low computing burden. Moreover, in this work we are interested in analyzing data variation. So, we provide a set of violin plots that contain the distribution of the makespans achieved for each instance.

A. FEASIBILITY TESTS

Our first approach seeks to determine the feasibility of the proposed model. So, we try out different scenarios and ana-

[1]Taillard's publicly available instances: http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html

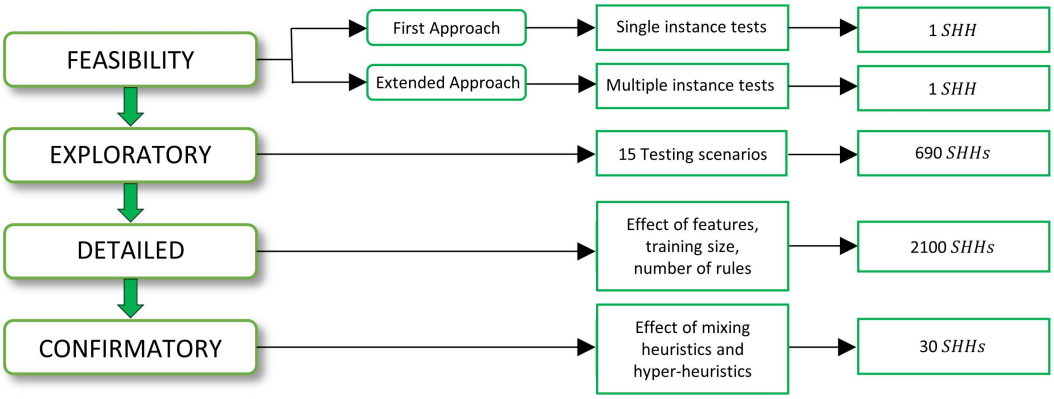


FIGURE 4. Four-stage methodology followed throughout this work.

TABLE 3. Testing sets used in this work with different performance gaps (Δ). Instances are equally distributed and they all have the same size (three jobs and four machines), unless otherwise stated.

ID	#Instances	Δ	Instance Description
TE01	30	N/A	Random
TE02	30	Free	MPAvsLPA + LPAvsMPA
TE03	30	Free	LPTvsSPT + SPTvsLPT
TE04	30	Free	AllvsMPA
TE05	30	Free	LPTvsSPT
TE06	30	Free	LPAvsMPA + SPTvsLPT
TE07	30	Free	MPAvsLPA + Random
TE08	30	Free	SPTvsLPT
TE09	60	20	LPTvsAll + SPTvsAll + MPAvsAll + LPAvsAll
Tai2015	10	N/A	Taillard set (20 jobs, 15 machines)
Tai2020	10	N/A	Taillard set (20 jobs, 20 machines)

lyze whether the SHH obtains better performance than the base models. First, we focus on assessing whether the proposed SHH adequately performs when solving a single JSSP. Then, we analyze if the performance gain also applies when solving a whole set of instances. For this stage, we use a single SHH and two trained HHs: HH_1 and HH_2. Each HH has access to four heuristics (see Section III-B for details): LPT, SPT, LPA, and MPA. Moreover, we train HH_1 with 30 LPTvsSPT instances, *i.e.,* favoring LPT, and HH_2 with 30 SPTvsLPT instances, *i.e.,* favoring SPT. Additionally, we train the SHH with a set of 30 random instances of the same size, *i.e.,* three jobs, and four machines.

1) FIRST APPROACH

Our approach for this experimental step is relatively straightforward. We solved a total of four instances. The first two are random instances. The others are specialized to a particular heuristic. We consider one instance favoring LPT while hindering SPT, *i.e.,* one instance from the LPTvsSPT class. The remaining instance has the opposite nature, thus favoring SPT while hindering LPT; the SPTvsLPT class. Then, we compare the performance of the three solvers while solving these instances. Note that the four testing instances are different from those used to train the solvers.

2) EXTENDED APPROACH

Afterward, we test the SHH model on whole instance sets, striving to detect if the patterns observed with single instances can be generalized. So, we perform a total of four tests that use sets TE01, TE05, TE08, and TE03, respectively (see Table 3). In each one, we solve the dataset with all the available solvers.

The first test might give us an overview of the generalization capabilities since it contains random instances. The second and third ones seek to determine if our proposed model can learn to focus on a particular solver, as instances preserve the nature of those we used to train HH_1 and HH_2, respectively. The final test mixes both kinds of instances. Consequently, each HH should exhibit an average performance, as we trained them as specialized solvers. Conversely, if the SHH properly learns when to use each solver, it should outperform both HHs.

B. EXPLORATORY TESTS

The exploratory stage focuses on varying the parameters and solvers of the model. This stage aims to find a set of values that allow training a SHH with good performance w.r.t. the HH models and low-level heuristics. We train several SHHs using the sets from Table 2 and test them on the sets from Table 3. We define a total of 15 tests with different foci, as Table 4 shows. For example, we use random instances for tests 1-3. In other tests, we vary the parameters of the SHH, such as the number of rules. Also, we alter training parameters in a few cases, such as the number of instances and the number of search agents for UPSO.

For the first three sets, we combine specialized and non-specialized HHs while training the SHH with random instances. Then, we use specialized training instances and remove the best heuristic from the list of available solvers

(tests 4 and 5). In tests 6-11, we use arbitrary combinations of HHs and training instances.

Test 12 represents our first test where the SHH has access to more than two solvers. Afterward, in test 13, we expand upon the testing conditions by providing the SHH with one more solver and raising the number of rules in the model to 10. Bear in mind that whenever the model has more rules than solvers, some will appear more than once. The optimization process will also determine the specific values for each rule and the number of repetitions for each solver. In the following test (test 14), we seek to favor the model by training it with instances exhibiting mixed behaviors and including more search agents throughout the optimization procedure. Nonetheless, we also include test 15 to determine the model's behavior when using instances from the literature. However, such instances are larger than the ones we tailored, restricting the number of tests we can carry out with the available computing resources.

TABLE 4. Experiments performed in the exploratory stage. The definition of each instance set is given in Table 2. These tests use the following UPSO parameters: $\phi_1 = 2.00$, $\phi_2 = 2.50$, $u = 0.25$, and 100 iterations. Parameter N_a changes as indicated. All available solvers are hyper-heuristics trained with different sets, as indicated by their names.

ID	Solvers	Training	#Rules	N_a
1	$HH_{\text{TR01\_1}}$ and $HH_{\text{TR01\_2}}$	TR01	4	15
2	HH_{TR02} and HH_{TR03}	TR01	4	15
3	HH_{TR04} and HH_{TR05}	TR01	4	15
4	HH_{TR04} and HH_{TR05}	TR06	4	15
5	HH_{TR02} and HH_{TR03}	TR06	4	15
6	HH_{TR04} and HH_{TR04}	TR01	4	15
7	HH_{TR01} and HH_{TR03}	TR01	4	15
8	HH_{TR01} and HH_{TR04}	TR01	4	15
9	HH_{TR02} and HH_{TR02}	TR02	4	15
10	HH_{TR03} and HH_{TR04}	TR03	4	15
11	HH_{TR05} and HH_{TR01}	TR05	4	15
12	$HH_{\text{TR02}} - HH_{\text{TR05}}$	TR01	4	15
13	$HH_{\text{TR01}} - HH_{\text{TR05}}$	TR14	10	15
14	HH_{TR07} and HH_{TR08}	TR09	2	30
15	$HH_{\text{Tai1515\_1}}$ and $HH_{\text{Tai1515\_2}}$	Tai1515	10	30

C. DETAILED TESTS

As we shall discuss further on, test 14 provides the conditions where SHHs perform best. Hence, in this section, we exploit this scenario feature-wise, instance-wise, and rule-wise. To do so, we select the best performing run from each kind of hyper-heuristic. Hence, the analysis for this stage considers that SHHs always have access to the same set of four hyper-heuristics. Thus, we can focus the discussion on the effect of the training process of the SHH rather than on the variability of HHs.

Our first scenario explores the 31 combinations of the five available features (Section III-C), which Table 5 summarizes.

We train 30 SHHs for each combination while preserving the remaining testing conditions, and thus we analyze the performance of 930 SHHs. We rank them all based on their performance for the set of instances. We use such a ranking to analyze the variability of the data through a set of violin plots. With this information, we select the subset of features that seems to perform best.

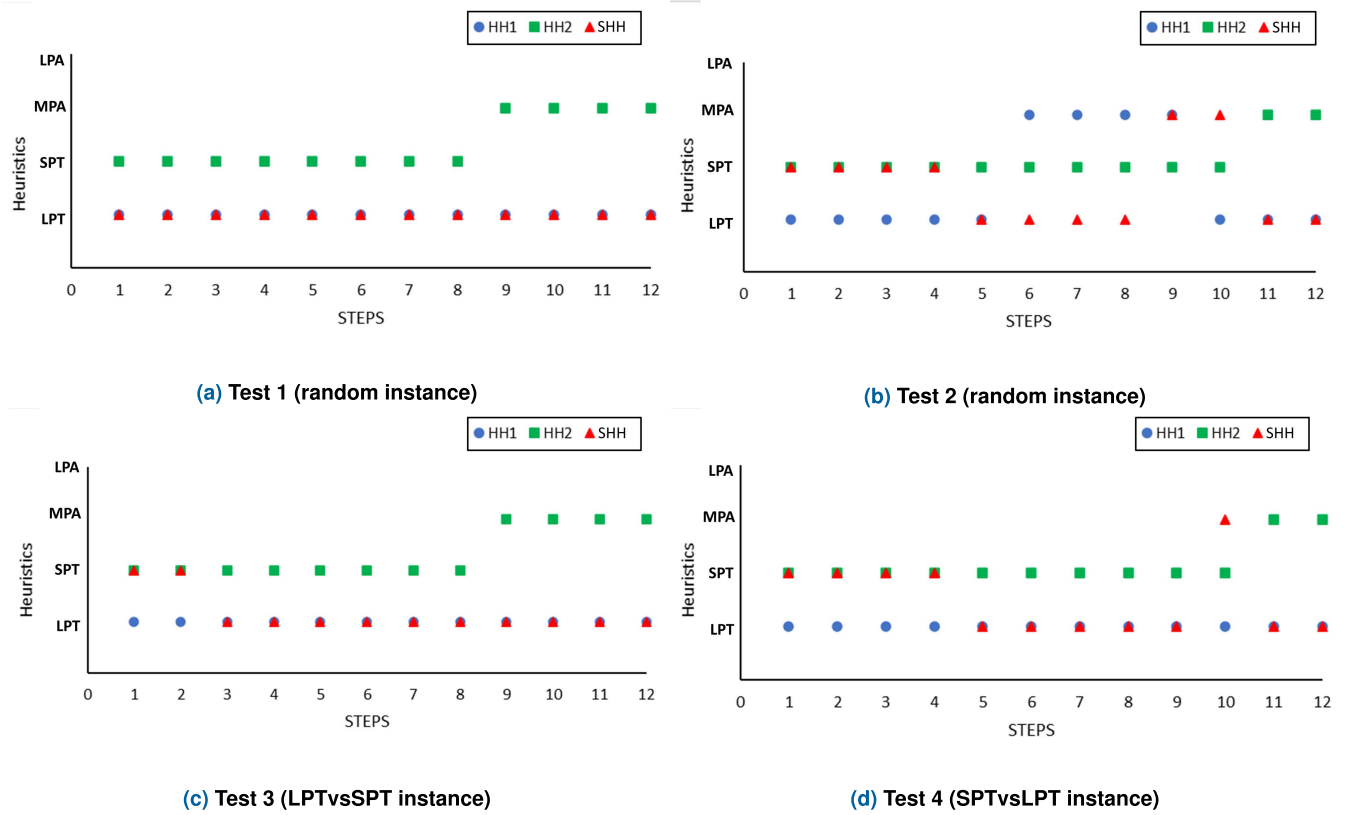
TABLE 5. Feature combinations used in detailed stage. The "x" represents the features that appear in the respective combination.

ID	Features				
	Mirsh222	Mirsh15	Mirsh29	Mirsh282	Mirsh95
1	×				
2	×	×			
3	×	×	×		
4	×	×	×	×	
5	×	×	×	×	×
6	×	×	×		×
7	×	×		×	
8	×	×		×	×
9	×	×			×
10	×		×		
11	×		×	×	
12	×		×	×	×
13	×		×		×
14	×			×	
15	×			×	×
16	×				×
17		×			
18		×	×		
19		×	×	×	
20		×	×	×	×
21		×	×		×
22		×		×	
23		×		×	×
24		×			×
25			×		
26			×	×	
27			×	×	×
28			×		×
29				×	
30				×	×
31					×

Afterward, we tackle the remaining parameters to detect the effect of having larger or smaller sets of training instances and considering more or fewer rules for the model. We create a total of nine training subsets with a different amount of instances from set TR09. All the subsets have the same number of instances from each kind (LPTvsAll, SPTvsAll, LPAvsAll, MPAvsAll). The smallest one includes 12 instances, and subsequent subsets increase their size linearly until they arrive at 108 instances. These subsets are labeled from IT1 to IT9, where IT1 is the smallest one. We also vary the number of rules of the SHH between 1 and 10. By merging both ranges, we obtain a total of 90 combinations. We repeat each experiment 30 times, and so we analyze 2700 SHHs in this stage.

D. CONFIRMATORY TESTS

In our final testing stage, we explore the performance of a simple flexible SHH. So, we train SHHs with simultaneous access to hyper-heuristics and heuristics. The training and testing sets used in this section are TR09 and TE09, respectively. Moreover, we use the following



(a) Test 1 (random instance)

(b) Test 2 (random instance)

(c) Test 3 (LPTvsSPT instance)

(d) Test 4 (SPTvsLPT instance)

FIGURE 5. Heuristic distribution for all tests related to the first approach stage.

conditions: Four rules, 30 repetitions, four available HHs (trained on TR07, TR08, TR10, and TR11, respectively), and four available heuristics (LPT, SPT, MPA, and LPA). We also use a subset of features for the models, given by `Mirsh222`, and `Mirsh95`. We make this decision based on the data achieved for the previous stages, as we discuss further on.

V. RESULTS AND DISCUSSION

We now move on to presenting the most relevant data from our experiments. For the sake of readability, we preserve the structure from Section IV.

A. FEASIBILITY TESTS

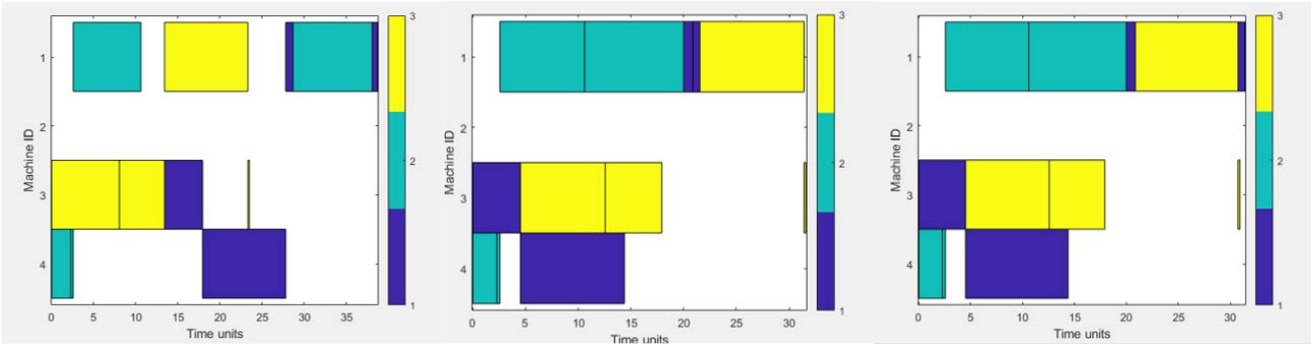
Remember that our first experimental stage is split into two phases. First, we tackle single instances. Then, we use whole sets.

1) FIRST APPROACH

For the first random instance (test 1), all solvers (HH_1, HH_2, and *SHH*) reach the same makespan value, *i.e.*, $c_{\max}(s) = 27.0784$. This is rather interesting since HH_2 actually leads to a different sequence of heuristics, as shown in Figure 5a. Nonetheless, with another random instance (test 2) HH_1, HH_2, and *SHH* lead to $c_{\max}$ values of 38.7128, 31.6054, and 31.4236, respectively. Hence, the SHH outperforms the HHs. We may explain this because we trained the SHH with a set of random instances. Therefore, it may be easier for *SHH* to solve this instance correctly. Nonetheless, bear in mind that *SHH* has access to only HH_1 and HH_2, which represent solvers trained for other kinds of instances. So, such an outcome is noteworthy.

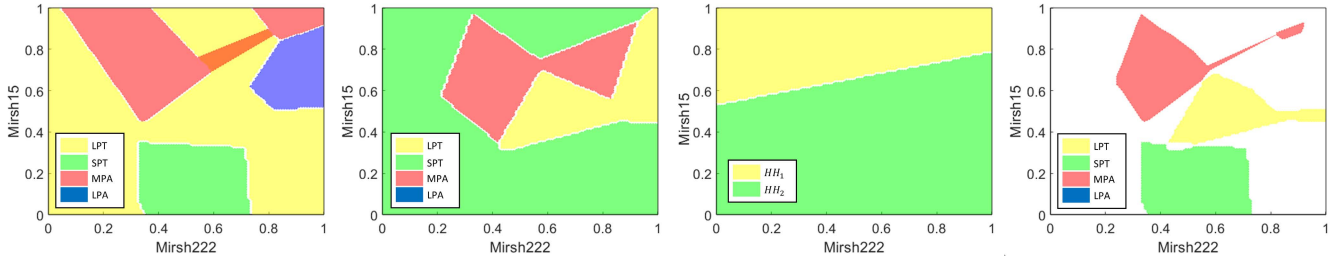
Let us now analyze the solutions yielded by each solver (Figure 6). Note that there is a substantial difference between the schedules yielded by HH_1 and those given by the other solvers. Besides, the schedules from HH_2 and *SHH* only differ in step nine: while the former schedules job three, the latter schedules job one. Therefore, a slight difference can lead to improved performance. Moreover, all solvers lead to a different sequence of heuristics (Figure 5b), which is unexpected due to the similarities between HH_2 and *SHH*.

The two final tests use specialized instances. For test 3, HH_1, HH_2, and *SHH* yielded a $c_{\max}$ of 39.9648, 88.8860, and 39.9648, respectively. As expected, HH_1 performs well since we trained it with instances of the same kind (LPTvsSPT). Interestingly enough, *SHH* yields the same makespan. Similarly, HH_2 performs poorly since we trained it with instances of the opposite kind (SPTvsLPT). The sequence of heuristics used by HH_1 and *SHH* are quite similar, as Figure 5c shows. They only differ in the first two steps, where *SHH* used HH_2. This suggests that it does not matter whether one uses SPT or LPT for the two first steps of this instance. Conversely, the heuristics used by HH_2 shows a very different, yet expected, distribution.



(a) HH_1 (b) HH_2 (c) *SHH*

FIGURE 6. Schedule of Test 2 with different high-level solvers.



(a) HH_1 (b) HH_2 (c) *SHH* (d) Common Zones (HH_1 & HH_2)

FIGURE 7. Action zones of different high-level solvers.

In the case of test 4, the makespan values were 79.3781, 39.3781, and 39.3781 for HH_1, HH_2, and *SHH*, respectively. As one may see, the performance of the HHs inverted since we used an instance of the opposite nature (SPTvsLPT). Notwithstanding, *SHH* once again replicates the performance of the best solver (HH_2), indicating that it might be possible for a SHH model to learn how to solve specialized instances through a training process based on random instances.

This is a rather interesting phenomenon. It seems as if the HHs somehow transferred their learning to the SHH. Bear in mind that a traditional approach would seek to train the SHH with a set of instances representing the union of the training sets used for each HH. However, we use a completely different set of instances. This may prove helpful for scenarios where the original instances are no longer available or where training with them may require vast computational resources. In fact, for these tests, all solvers used the same number of training instances (30). Nevertheless, if a traditional approach were to be followed, it would have required 60 instances (30 from each HH). Although this merits deeper research, it escapes the scope of this work.

The instance distribution for this last test (Figure 5d) is a bit more dynamic, *i.e., SHH* switches solvers more often. Although *SHH* selected HH_1 in half the steps, it still managed to replicate the makespan of HH_2. A possible explanation is that at this point both HHs target the same heuristic, thus being irrelevant to which one is chosen. Another explanation is that, by this point, in the solution process, the available heuristics select the same activity to schedule. This hints that small changes in the early stages of the solution process can be more relevant than larger ones at late stages.

Let us now analyze the action zones of the three solvers. These zones represent the regions of feature values where a solver uses each one of their available approaches. The LPT heuristic predominates in HH_1, as we expected (Figure 7a). Similarly, SPT dominates in HH_2 (Figure 7b). Bear in mind that, for the sake of simplicity, these plots show the action zones from the perspective of the first two features. Nonetheless, other features could be used. However, since we are interested in glancing at the performance of the SHH model, we omit such a detailed analysis.

Figure 7c show the action zones for *SHH*. At least from this perspective, *SHH* seems to lean towards using HH_2. Nonetheless, the number of times that each HH is used depends on the feature values that the instance produces throughout its solution process. For example, feature values may stagnate in a small region, or shift throughout the whole domain.

As a final analysis consider Figure 7d, where we show the intersection of the action zones for both HHs. Although they are specialized for two opposing kinds of instances, there are three regions where both solvers agree on the kind of heuristic that should be used.

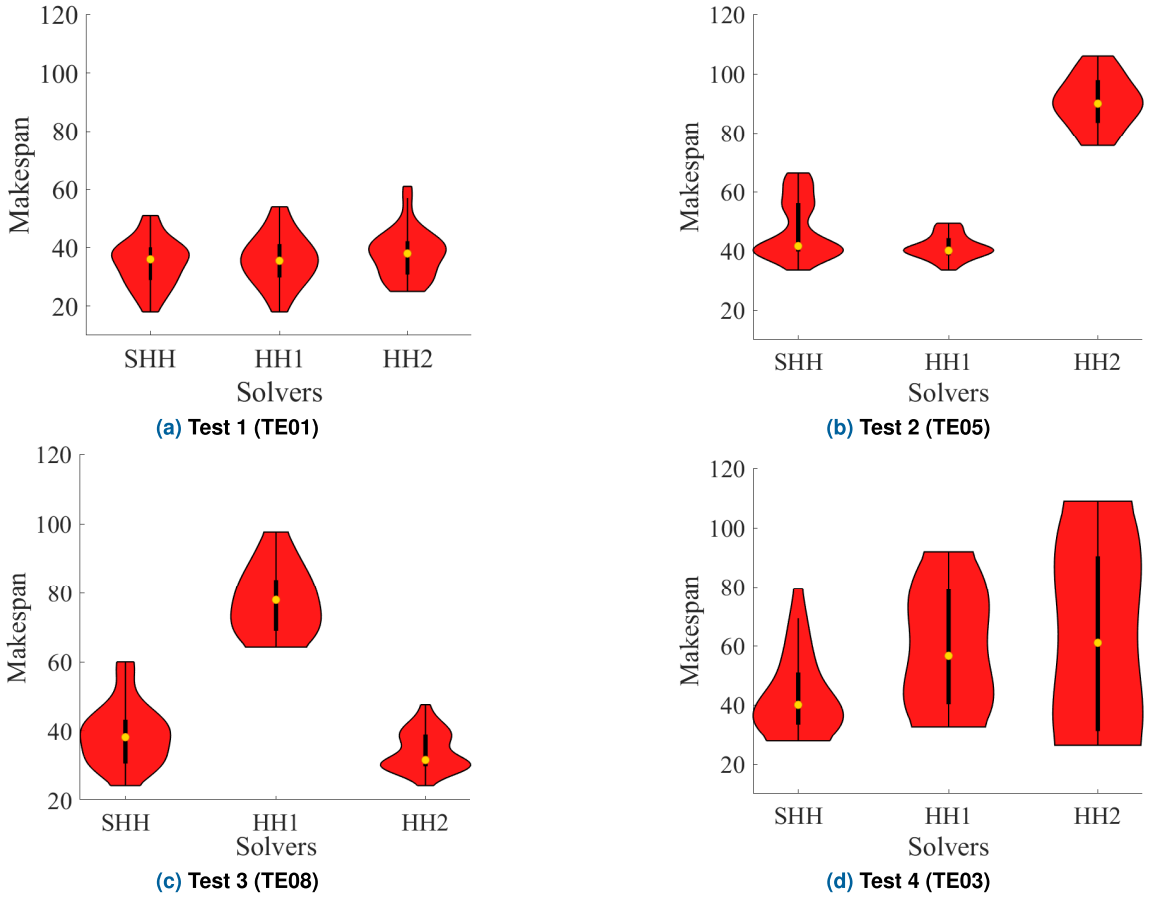


FIGURE 8. Makespan achieved by each solver (*SHH*, HH_1, and HH_2) for each test from the extended approach. The testing sets are indicated in parenthesis.

2) EXTENDED APPROACH

Figure 8 shows the results achieved by *SHH*, HH_1, and HH_2, over four instance sets (TE01, TE05, TE08, and TE03). Remember that we trained *SHH*, HH_1, and HH_2 with 30 random, 30 LPTvsSPT, and 30 SPTvsLPT instances, respectively (see Section IV-A2). We can appreciate from test 1 (Figure 8a) that *SHH* and HH_1 perform better when solving a set of random instances. Their violins have a similar shape, which suggests that *SHH* mimicks HH_1 to some extent. A similar pattern emerges in test 2, as those instances favor HH_1 (Figure 8b). Again, *SHH* reached a similar performance level. Although there are some instances where *SHH* performs worse than HH_1, they have virtually the same median value. In contrast, HH_2 performs poorly in all instances, as expected.

Test 3 provides a similar but opposite trend. In this case, *SHH* strives to mimic HH_2, whereas HH_1 performs poorly. This behavior is somewhat expected because of the instances considered for this test. Even so, the tests hint at the possibility of reaching a more general solver that can deal with both kinds of instances simultaneously through the use of specialized solvers.

This is precisely what we explore in test 4. Remember that this test considers a mix of 15 LPTvsSPT (*i.e.,* favoring LPT and hindering SPT) and 15 SPTvsLPT (*i.e.,* favoring SPT and hindering LPT) instances. Therefore, HH_1 and HH_2 should perform well in only half of the instances. Conversely, *SHH* should excel in all of them. Even if the effect is not as strong

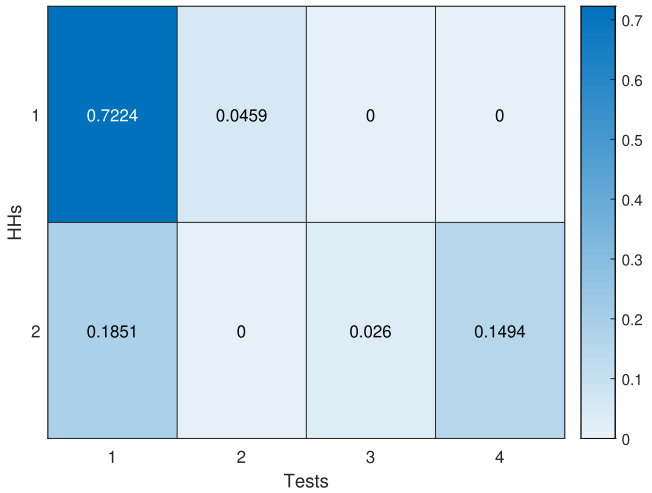


FIGURE 9. P-values of data about each test from the extended approach. The first row compares *SHH* against HH_1. The second row compares *SHH* against HH_2.

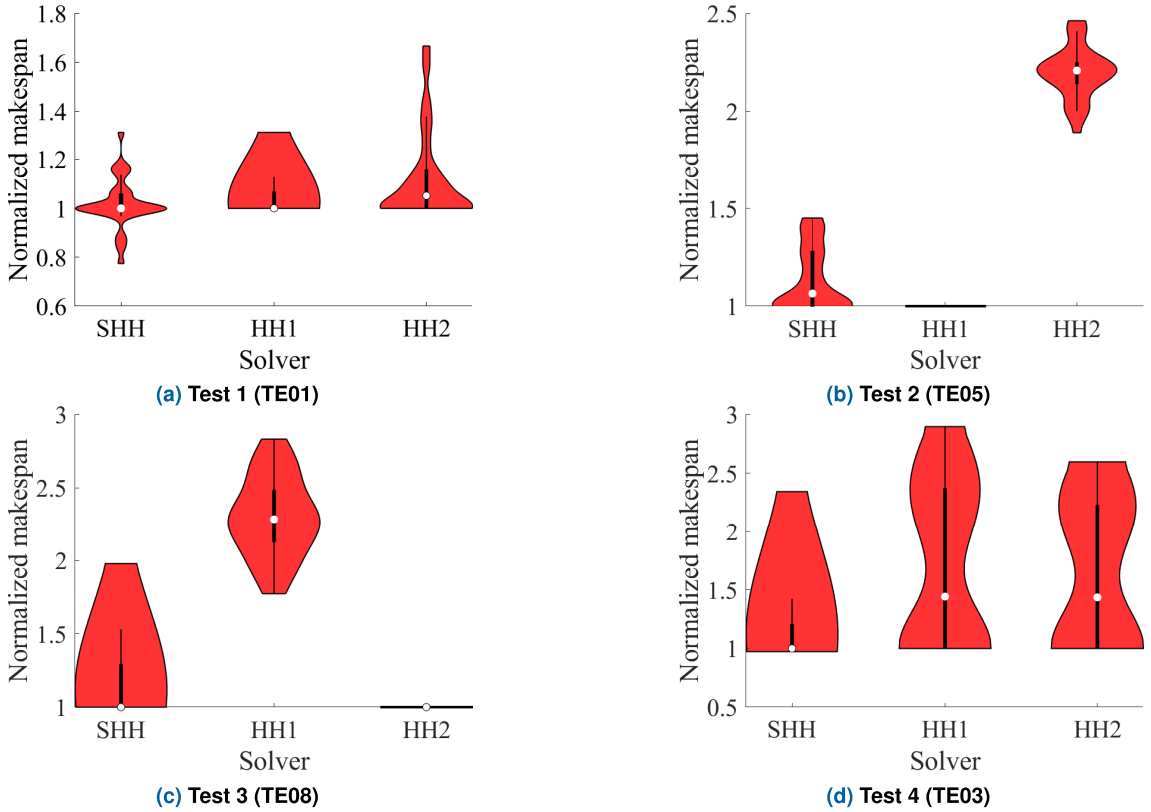


FIGURE 10. Normalized makespan values for the data shown in Figure 8. The normalization is carried out by dividing the performance of a solver into that of the best-performing *HH*.

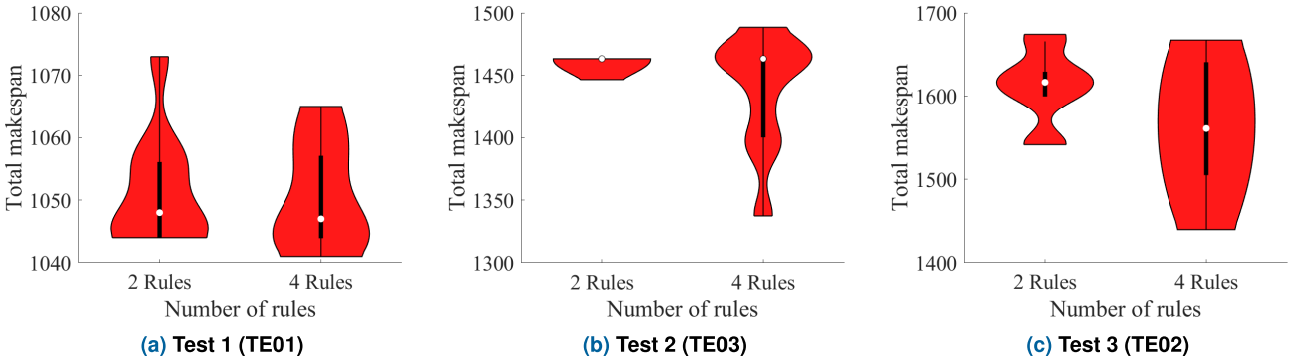
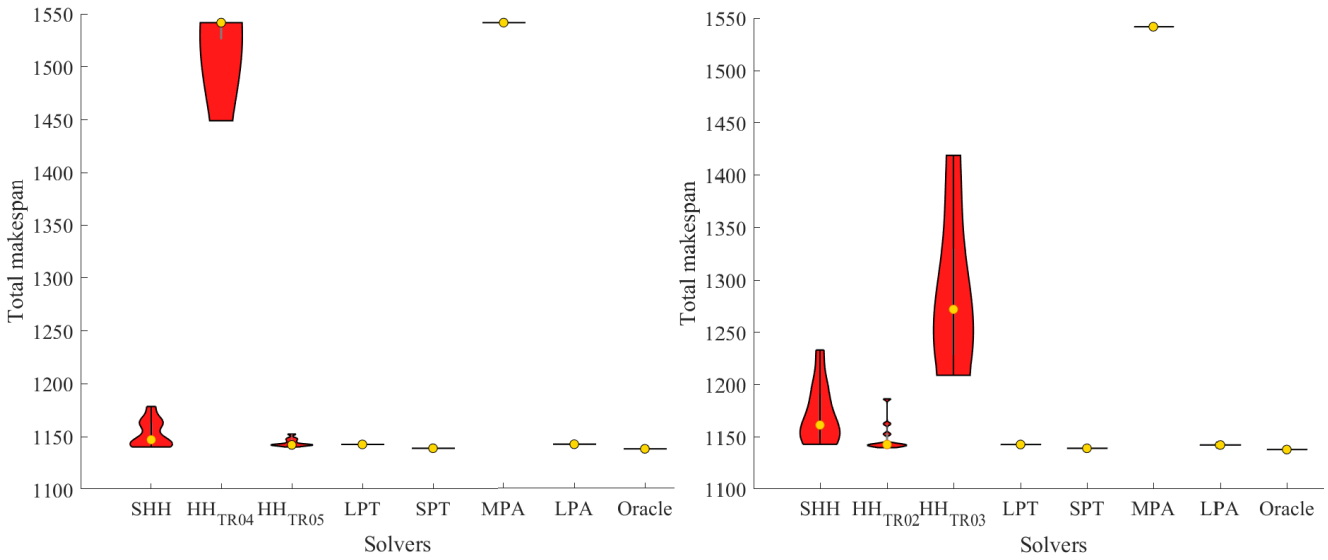


FIGURE 11. SHH performance distribution throughout the first three exploratory tests for two different numbers of rules. The testing sets are indicated in parenthesis and in all cases the SHHs were trained using TR01.

as we suspected, Figure 8d shows that *SHH* achieves a good overall performance. This is reflected in a median makespan 30% smaller than the next best solver (HH_1) and in a majority of instances with smaller makespan values.

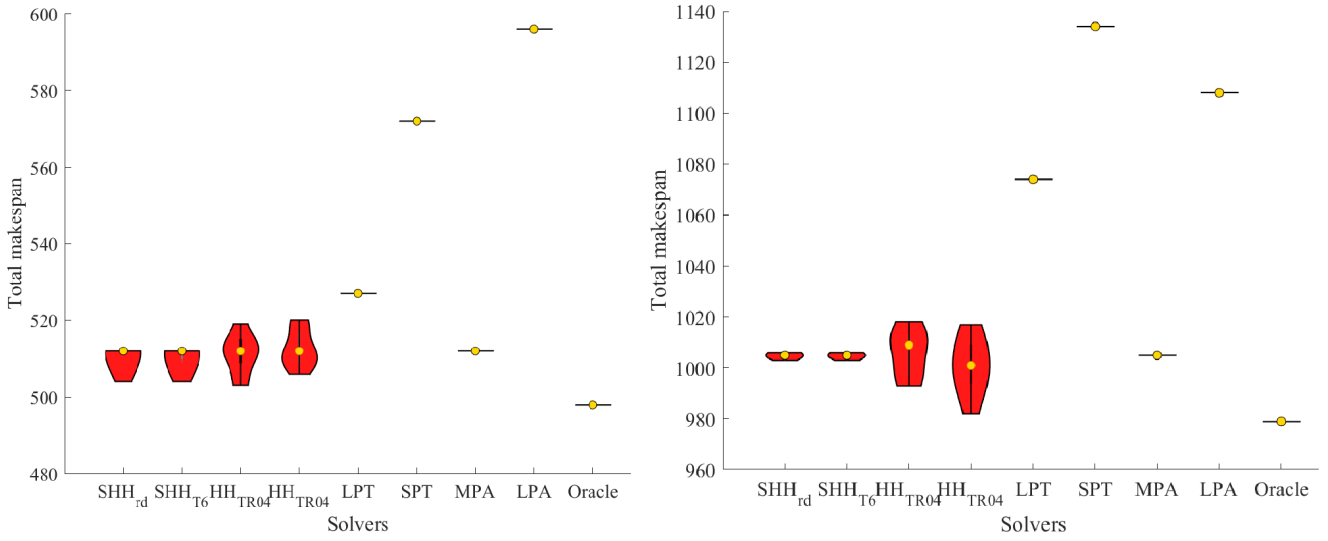
We ran a Wilcoxon test between the SHH and each HH across the four testing scenarios to assess these differences better. Figure 9 provides a heatmap that summarizes the resulting data. The p-values from test 1 reveal that there is not enough evidence to state that the SHH is significantly different from any of the HHs. Conversely, the difference becomes statistically significant in tests 2 and 3. Notwithstanding, test 4 shows a peculiar behavior. Data suggest that there is a statistically significant difference w.r.t. HH_1 but not w.r.t. HH_2. Even so, the median performance of *SHH* is better than those of the others.

Since all instances can have different lower bounds, we also provide a plot of the normalized makespan data (Figure 10). To do so, we divide each makespan into that of a 'hyper-oracle', which contains the best solution given by a HH for each instance. This means that any single HH can never reach a value below unity. However, since the SHH



(a) Test 4 (b) Test 5

FIGURE 12. Performance of exploratory tests 4 and 5 when solving TE04. The HHs displayed in each plot represent the available solvers for *SHH*. The oracle is given by the best makespan achieved by all heuristics over each JSSP instance.



(a) Training (TR01) (b) Testing (TE01)

FIGURE 13. Performance of exploratory test 6 when solving TE01. Training set TR01. SHH_{rd} summarizes the performance of SHHs with randomly generated rules. The HHs displayed in each plot represent the available solvers for *SHH*. The oracle is given by the best makespan achieved by all heuristics over each JSSP instance.

is free to swap HHs throughout the solution process, it may reach a better solution, reflecting in a lower than unity normalized makespan. Hence, it seems that it is better to combine specialized hyper-heuristics throughout the solution process when solving random instances (test 1). For tests 2 and 3, hyper-heuristics behave as expected and it is evident that *SHH* becomes distracted in some instances by the presence of the other solver. Nevertheless, it is also evident that there is something to gain from using our proposed model to solve mixed sets (test 4). This plot makes it easier to validate that *SHH* selects a proper solver for about half the instances, as the near unity median value reveals. Evidently, training a HH or SHH with a balanced set of specialized instances could yield better performance. Notwithstanding, at this point, we are not confident whether our model would still outperform the others given similar training conditions. But considering the limitations that a HH presents against a SHH, the latter should fare better.

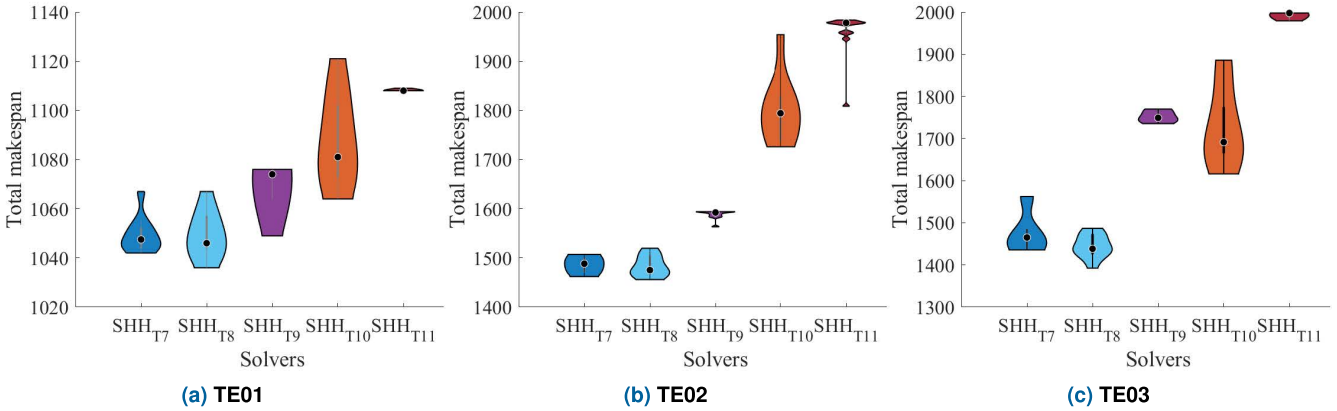


FIGURE 14. Performance achieved by SHHs from exploratory tests 7 to 11, when solving three different testing sets.

B. EXPLORATORY TESTS

Before diving into the data, we want to mention that, from this point onward, we refer to a hyper-heuristic trained with a specific instance set by using a subscript with the training set. So, HH_{TR01} is a hyper-heuristic trained with the TR01 instance set.

Figure 11 shows the data for the first three experiments, where the SHHs are trained with the same set of random instances (TR01). Note that these tests were originally devised with four rules and that each HH has access to four heuristics. However, we also provide data for the performance of SHHs with two rules, seeking to analyze the effect of varying this parameter. In all three tests, the SHHs with four rules reached a smaller median makespan than the SHHs with two rules. Notwithstanding, in the second and third tests, we trained the HHs with different kinds of instances (LPTvsAll and SPTvsAll for test 2, and MPAvsAll and LPAvsAll for test 3). This is somehow mapped into the shapes of performance metrics, seemingly being less relevant when using random instances (test 1).

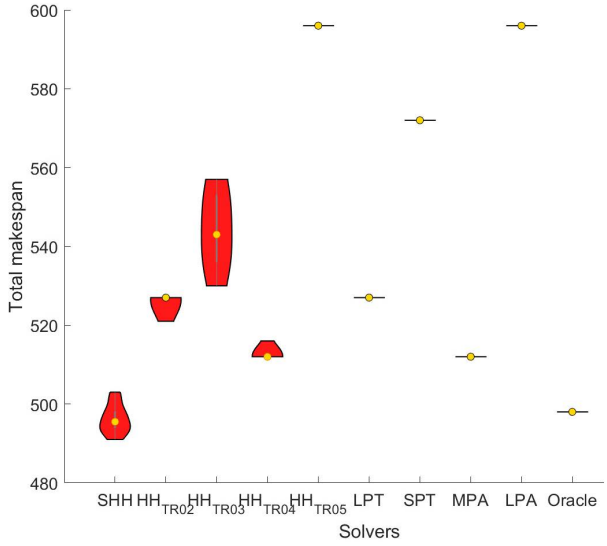
Next, we discuss tests 4 and 5. These tests involve set TE04, which incorporates instances that hinder MPA. Figure 12 shows the performance of the *SHHs*. We include the performance of the base HHs, the heuristics, and the oracle for reference purposes. In test 4 (Figure 12a), HH_{TR04} should lose against HH_{TR05}. The reason: the former specializes in instances favoring MPA, while the latter does so in instances favoring LPA, and LPA exhibits good performance on set TE04.

In the case of test 5 (Figure 12b), HH_{TR02} and HH_{TR03} specialize in instances favoring LPT and SPT, respectively. Hence, they should perform well when tested on instances hindering MPA. Notwithstanding, HH_{TR02} outperforms HH_{TR03}. We may explain this since whenever we train a HH with instances favoring SPT, the resulting solver seems to considerably select MPA (Figure 5).

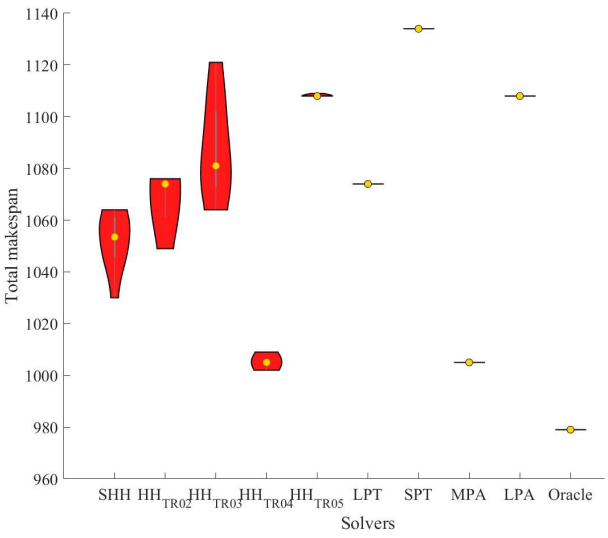
The SHHs select the best solver in both tests. This tendency is clearly stated in test 4, since *SHH* has a median performance metric virtually equal to that of HH_{TR05}. But the pattern for test 5 is rather interesting. Even if HH_{TR02} is a better solver for this kind of instances, HH_{TR03} does not perform poorly. Therefore, *SHH* does not discard it and the resulting median performance lands slightly above that of the best solver.

Since MPA dominates when solving random instances, in test 6, we trained HHs with instances favoring MPA and added them to the pool of solvers for *SHH*. Figure 13 shows the results of training and testing with random instances. The median performance achieved by *SHH* (second violin) and its solvers (third and fourth violin) is similar to the performance achieved by MPA in both training and testing. It seems that both HHs always perform well, which tells us that the training should not have gone wrong. Additionally, in both scenarios, we added the performance of 10 SHHs containing the same HHs, but with randomly generated rules, *i.e.,* where no training was carried out. This SHH is located in the first column of each figure and, as we can see, it performs almost identically to the trained SHH in both, the training and testing subsets. This indicates that if solvers perform properly, we may arbitrarily choose between them and expect a good performance.

The subsequent five tests (7-11) represent SHHs with access to arbitrary pairs of HHs. We summarize their performance in Figure 14, where the subscript indicates the test number associated with the SHH. The SHHs from tests 7 and 8 (SHH_{T7} and SHH_{T8}, respectively) outperform the others over the three testing sets (TE01, TE02, and TE03). Bear in mind that SHH_{T7} has access to a hyper-heuristic trained with random instances (HH_{TR01}) and one trained with instances favoring SPT (HH_{TR03}). Therefore, it is noteworthy that SHH_{T7} excels in all tests. Even though TE01 and TE03 include random instances and instances favoring SPT, respectively, SHH_{T7} only really has an advantage with TE01 (Figure 14a). In the case of set TE03, SPT is good only at solving half the set. Similarly, set TE02 contains instances unrelated to SPT.

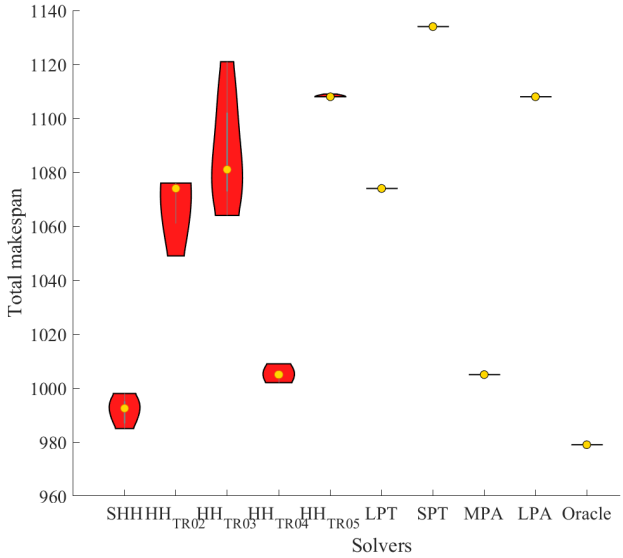


(a) Training (TR01)

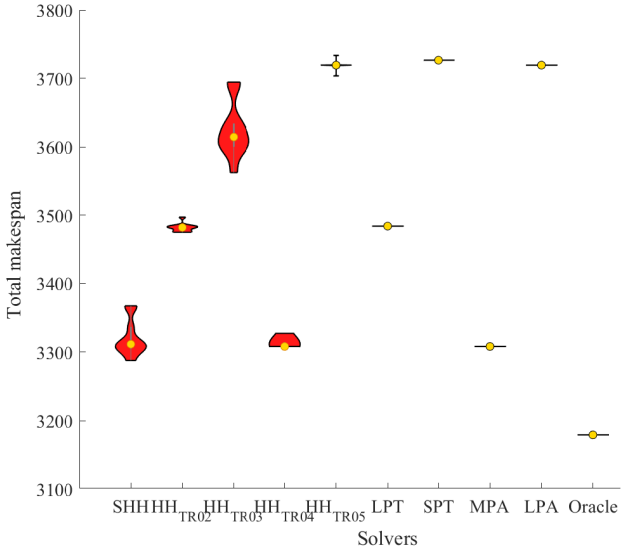


(b) Testing (TE01)

FIGURE 15. Performance of exploratory test 12. Training set: TR01. Testing set: TE01.



(a) Training (TE01)



(b) Testing (TR14)

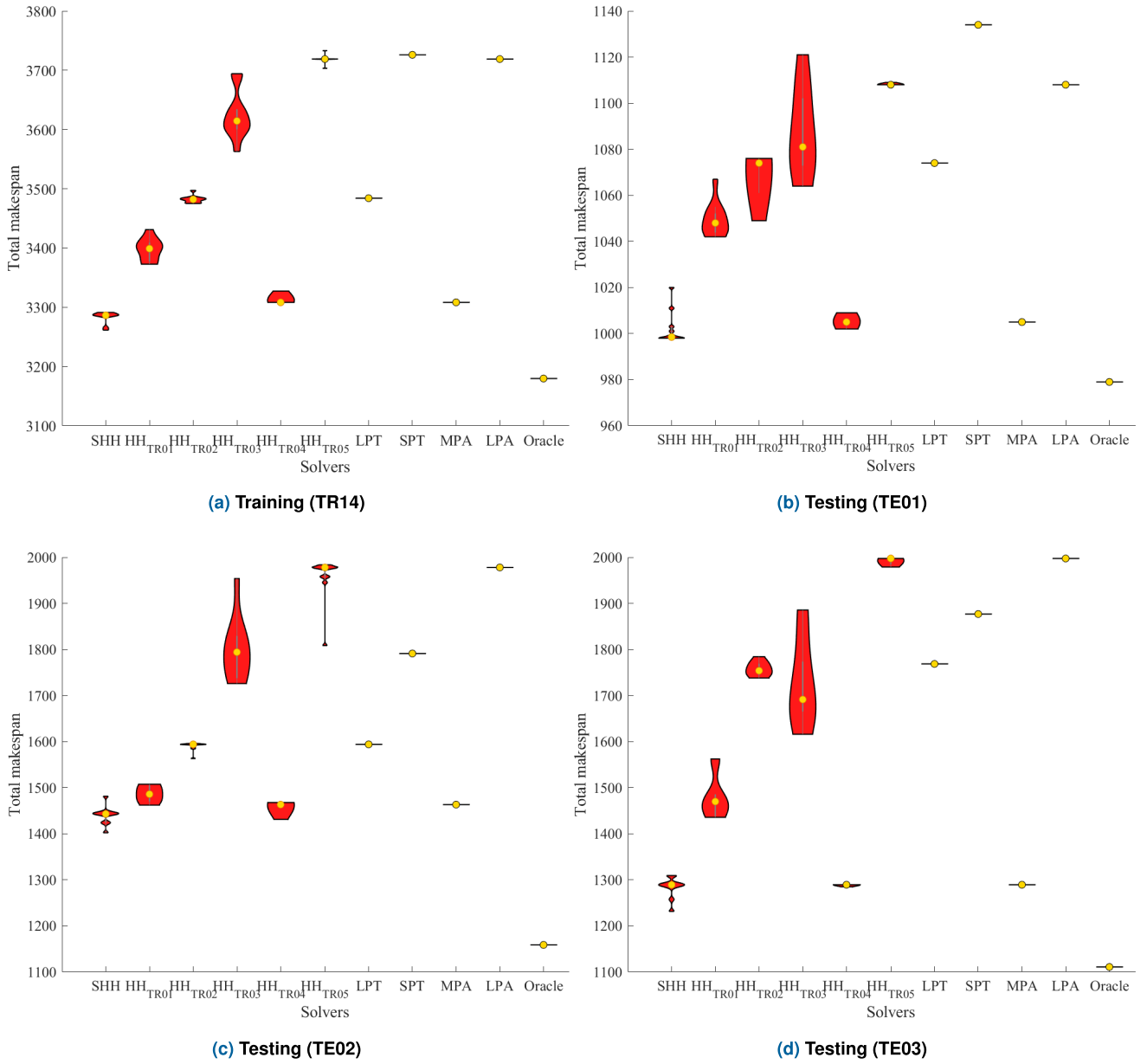
FIGURE 16. Performance of exploratory test 12 when using different training (TE01) and testing (TR14) sets.

The SHH from test 8 can use HH_{TR04}, which focuses on MPA. So, it may solve half of set TE02 correctly. Nevertheless, we must still analyze how it manages to handle the other half properly. As Figure 14b suggests, the other available solver (HH_{TR01}) seems good at solving LPAvsMPA instances, which helps to explain the behavior we observed.

The following best SHH appears in test 9. This SHH contains two HHs specialized in the same kind of instances (LPTvsSPT). Still, it is pretty stable and performs rather well when solving set TE02, which contains instances of a different kind. Moreover, its performance impoverishes set TE03 since half the set contains instances of an opposing nature (SPTvsLPT).

The SHHs developed for tests 7 and 8 exhibit good performance on their own. However, training a SHH that uses their specialized solvers (test 10) proves no benefit when solving these testing sets. Hence, it seems that HH_{TR01} plays a key role by complementing the specialized HHs.

Similarly, test 11 contains a HH specialized in random instances and a HH suited to specialized instances. Notwithstanding, it yields the worst performance among the five SHHs. One may explain this by the fact that LPA is not a good heuristic in an overall sense (Figure 13a). So, combining HH_{TR01} with either HH_{TR03} or HH_{TR04} is a good approach, as HH_{TR03} tends to use MPA.



(a) Training (TR14)

(b) Testing (TE01)

(c) Testing (TE02)

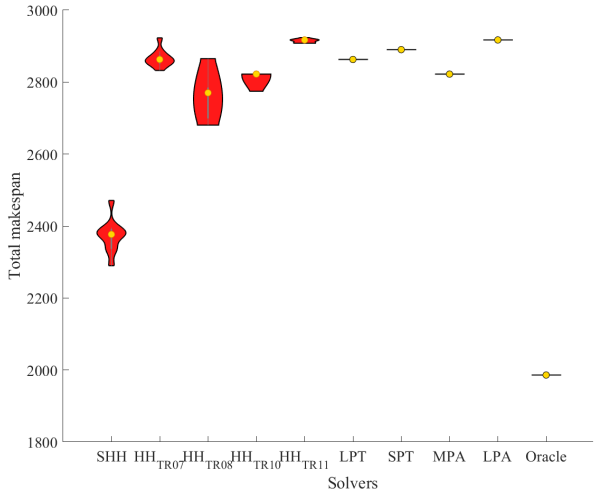
(d) Testing (TE03)

FIGURE 17. Performance achieved by SHHs throughout the exploratory test 13 for the training set (TR14) and three different testing sets (TE01 - TE03).

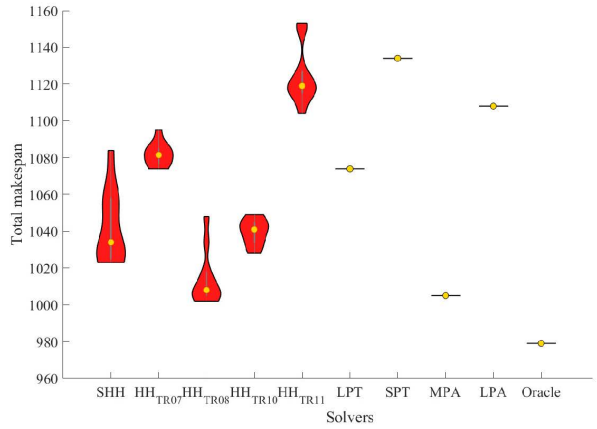
For test 12, we doubled the number of available solvers (HHs). We use the following specialized HHs: HH_{TR02}, HH_{TR03}, HH_{TR04}, and HH_{TR05}. So, we expect that performance will improve. Figure 15a shows that this indeed happens and that *SHH* outperforms the oracle. Notwithstanding, if we test the model in another set of random instances (such as TE01), performance diminishes drastically (Figure 15b).

The performance of each HH on the testing set does not differ much from the one achieved when training. Among the high-level solvers, HH_{TR04} keeps yielding the best total makespan, reaching a performance similar to MPA and worse than the oracle. This may be due the fact that HH_{TR04} specializes in MPAvsAll instances. Although these experiments are based on random instances, MPA is a good overall heuristic. Hence, a hyper-heuristic that takes advantage of that heuristic may achieve a better performance more easily.

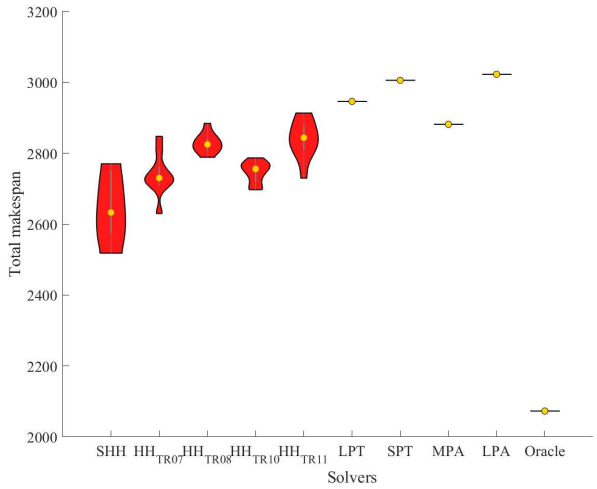
Data suggest two reasons for such a behavior. One is that the algorithm becomes over-specialized to the instances, thus hindering generalization. This might happen because we have a small amount of training instances. Conversely, it may be that *SHH* gets confused due to its available solvers. The performance of HH_{TR02} in the training set is similar to that of HH_{TR04}. But, it does not perform well on TE01. So, learning to use HH_{TR02} actually becomes counterproductive.



(a) Training (TR09)



(b) Testing (TE01)



(c) Testing (TE09)

FIGURE 18. Performance achieved by SHHs in the exploratory test 14 when solving the training set (TR09) and two testing sets (TE01 and TE09).

To better analyze this performance drop we now change the training set for TE01. This should reduce the chance of over-fitting the model, since it has twice the training instances. Figure 16a shows the resulting performance. There is a bigger difference between the two best HHs (HH_{TR04} and HH_{TR02}). This, in conjunction with the larger set of instances, should direct the rules of *SHH* towards HH_{TR04}.

Figure 16b corroborates that swapping the training set for TE01 allows for a better performance. In fact, some SHHs outperform HH_{TR04} and all the available heuristics. Moreover, and although the oracle still performs better, the difference is not abysmal.

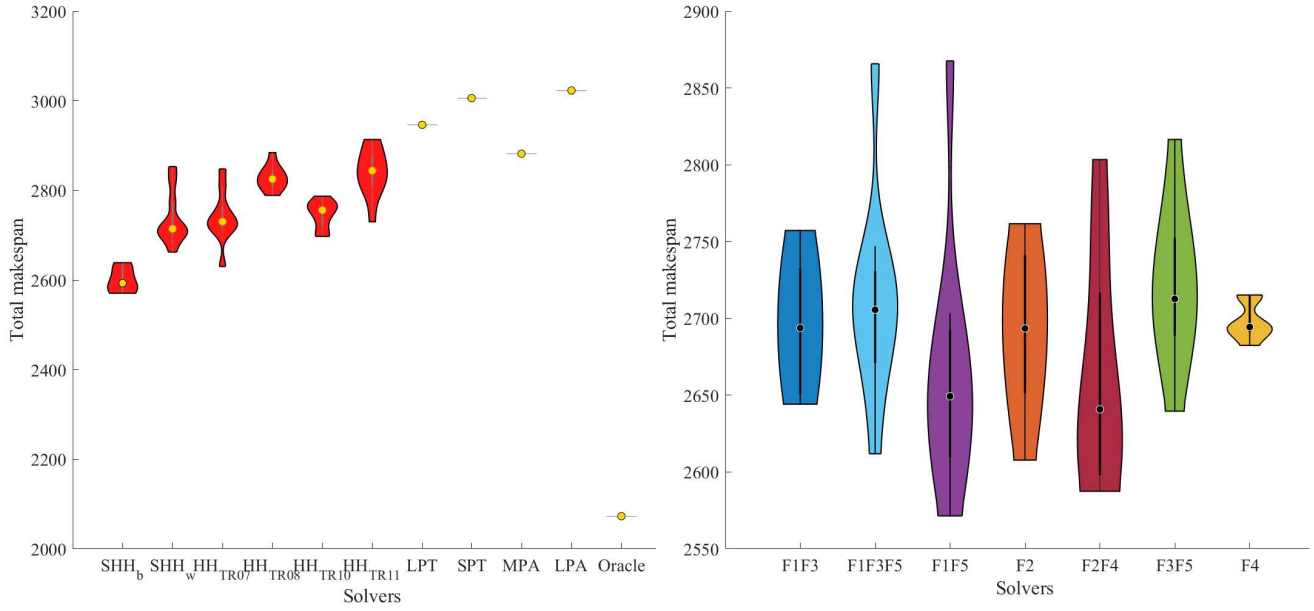
Through test 13, we push our experiments a little bit further. Here, we train using set TR14 (100 random instances), and we add an extra HH, which specializes in random instances (HH_{TR01}). Moreover, we preserve the number of rules for the SHH (four). Figure 17a shows the training data. Although the oracle performs best, *SHH* outperforms all the remaining solvers. The best solver within *SHH* appears to be HH_{TR04}, as we expected.

Let us now glance at how performance changes when solving the testing sets (TE01 - TE03). In all experiments (Figures 17b to 17d) *SHH* seems to exhibit similar behavior: it performs better than the available solvers (HHs or heuristics), but worse than the oracle. Notice that the HHs and heuristics vary their performance across tests. The only exceptions are MPA and HH_{TR04}, which excel in all instance sets. For example, HH_{TR02} performs averagely in TE01, properly in TE02, and poorly in TE03. Notwithstanding, our proposed approach remains stable across sets. Our interpretation of these data is that *SHH* seeks to replicate HH_{TR04}, which excels in all sets. So, one should strive to incorporate solvers within *SHH* with a similar performance level, to avoid biasing the model. This leads us to our next test.

In test 14, we consider a balanced training set. Our idea is to have a set where all solvers (both HHs and heuristics) yield a similar performance level. So, we consider a set of instances built with bounded instances of different kinds (TR09). The chief reason is to bound the performance of MPA, as it outperforms the other solvers. Moreover, we train the HHs with bounded instances of opposite kinds. For example, we train HH_{TR07} using 30 instances with $\Delta = 20$: 15 favoring LPT (LPTvsAll) and 15 favoring SPT (SPTvsAll). Figure 18a shows that all HHs and heuristics yield a total makespan for the training set around 2800. Even so, MPA remains the best heuristic. However, its overall performance is closer to the other heuristics than in previous tests.

The best HH when training is HH_{TR08} as it portrays the lowest median. Notwithstanding, it exhibits the highest variation across repetitions. We believe this is due to the presence of LPA. Conversely, *SHH* excels upon the other solvers and it performs close to the oracle.

Figure 18b shows the performance distribution when solving the set of random instances TE01. We already know that MPA is good at solving this set. Consequently, the best HHs in this test are HH_{TR08} and HH_{TR10}, which contain instances that favor MPA in their training sets. Although the SHH performed well in previous tests (see Figure 18b), this time it does not excel, yielding a median that is only better than the



(a) Best and worst HHs (b) Feature subsets

FIGURE 19. Left: Performance achieved by SHHs with access to the best (SHH_b) and worst (SHH_w) base HHs during the exploratory test 14 and when solving the test set (TE09). Right: Effect on the training performance of SHHs (set TR09) when using different feature subsets.

second best HH. However, when solving TE09 it outperforms the other solvers (Figure 18c). Nonetheless, it is hard to beat the oracle in this set of specialized instances, since they were tailored to different heuristics.

Another variable to explore is the way in which the performance of HHs affect the performance of the SHH. For doing so, we select the best and worst runs for each kind of HH from test 14, *i.e.,* those with the lowest and highest makespans for set TE09, respectively. Figure 19a shows the performance distribution of SHHs trained with the best (first violin) and worst (second violin) HHs. The former yields a better performance, as one may expect. The remaining violins in the figures represent the other solvers and the oracle. Note that SHHs trained with the worst HHs still managed to obtain a better median than the best solver, which is noteworthy.

Throughout this test, we also explored the effect of varying the features considered for analyzing the problem. Figure 19b shows seven combinations of features, and the total makespan yielded for different SHHs. Even if we only present some feature combinations, it is evident that performance can change drastically. For example, using F1 (`Mirsh222`) and F5 (`Mirsh95`) leads to a good median result but with a high variance across repetitions. Oppositely, performance remains stable and at an average level when using F4 (`Mirsh282`). This opens a door for exploring the remaining combinations and finding the subset of features that might be a better option when training SHHs.

Figure 20 compares representative solvers from Figures 18 and 19. Moreover, we include the performance distribution of a set of 10000 SHHs that use ten randomly generated rules each and the same four HHs than for the SHHs from

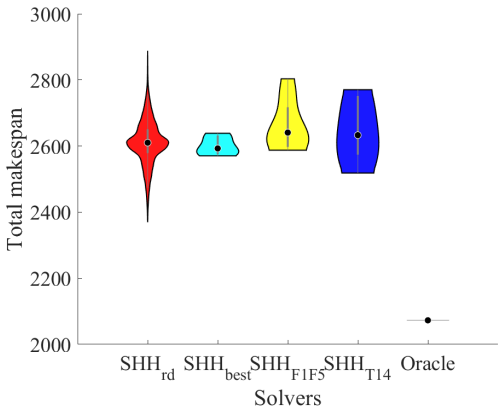


FIGURE 20. Performance distribution of SHHs obtained with the best testing conditions from exploratory test 14 when solving TE09 and including the performance of 10000 SHHs with randomly generated rules (SHH_{rd}), for comparison purposes. SHH_{F1F5} represents SHHs trained with features F1 (`Mirsh222`) and F5 (`Mirsh95`).

Figure 18c. We also include the performance of the oracle for comparison purposes. It is noteworthy that all sets of SHHs yield a similar median value. This is interesting because of the random nature of the SHHs from the first violin. Although this suggests that we may not need the training to reach this performance level, it is important to remember that this approach tests the performance of a high number of SHHs.

We have shown that combining HHs improves performance. Despite this, we are unaware of the best performance level that the model may achieve and whether it is better than the oracle's. For shedding some light on this matter, we create

a sort of hyper-oracle, identified as HH-oracle, which stores the result of the HH that yields the lowest makespan over every single instance. However, this oracle can be approached from two perspectives. First, we have an absolute HH-oracle, which analyzes all runs simultaneously and chooses the best ones for each solver. We can also define a relative HH-oracle in a similar way, but which only compares the HHs from a single run. So, the latter provides a variance related to the variation of the solvers across the runs.

When we applied such oracles to test 14, we noticed that they both excel in the training and testing sets. Notwithstanding, and although differences are small, the heuristic oracle remains the best performing solver, even outperforming both HH-oracles. However, we omit the plot for the sake of brevity.

Our last experiment in this stage covers some of the classic benchmarks. More specifically, we use our model for tackling Taillard instances. As mentioned in the methodology, we use set Tai1515 for training, containing ten instances with 15 jobs and 15 machines each. With it, we train the SHH and the HHs, arriving at the performance levels shown in Figure 21a.

It is interesting to see that these solvers outperform the oracle. Another element that stands out is that the HH-oracles also perform better than the heuristic oracle. This tells us that for these instances, combining HHs is better than only combining heuristics, which favors the existence of our SHH model. So, we use these solvers for tackling sets Tai2015 and Tai2020 (Figure 21b). The performance is similar to when training, with the difference that the performance of the SHH and the HHs is slightly worse in set Tai2015. Moreover, when solving set Tai2020, all solvers yield a median similar to the oracle and MPA. Besides, MPA remains the dominant heuristic.

The results of this last exploratory experiment are very favorable for the SHH, but there are a couple of issues. The first one is that the instances are unbalanced for the kind of heuristics that we are using, as MPA predominates. As we showed before, this may lead to troubles during the training. The second setback is that Taillard instances are large instances, which reduces the number of experiments that we can run in the same amount of time. For these reasons, and given that we also have another positive experiment, we exploit test 14 in the next section, searching for a combination of parameters that improve the performance of our proposed model.

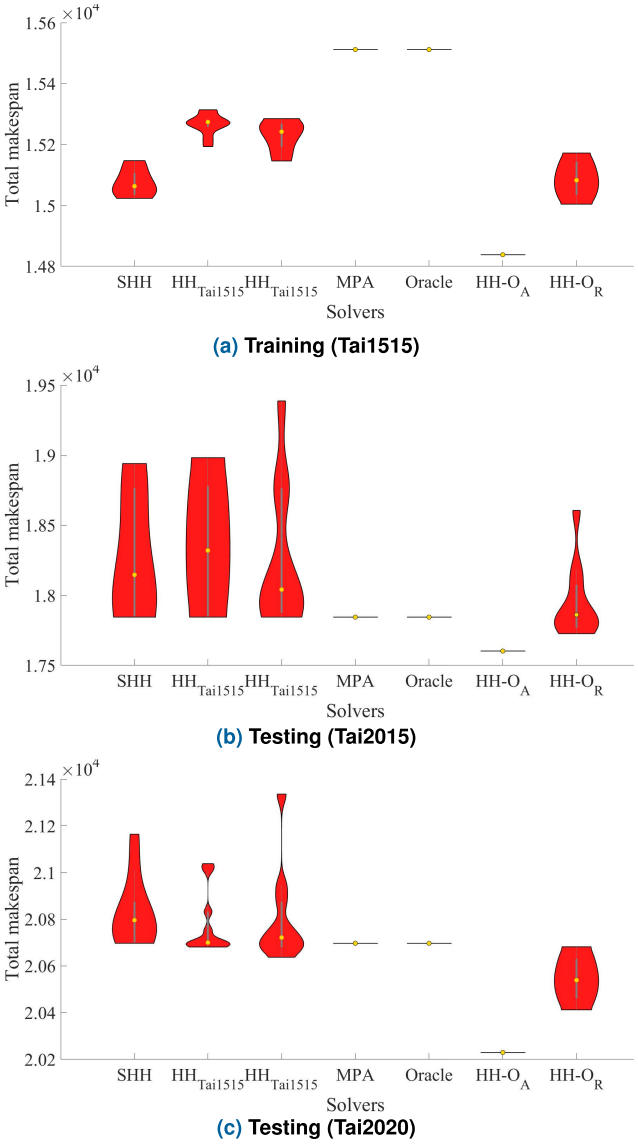


FIGURE 21. Performance achieved by SHHs and by the available solvers when tackling Taillard instances (test 15) in the training set (Tai1515) and two testing sets (Tai2015 and Tai2020). The figures also display the performance of three kinds of oracles, for comparison purposes.

C. DETAILED TESTS

In this section, we explore two paths. The first one assesses whether a subset of features might yield better performance than the whole set. As we mentioned in Section III-C, throughout this work, we consider a set of five features from the literature. So, we tested all combinations of such features (Figure 22). Data reveal that violin 16 is the best combination of features, which corresponds to F1 and F5 (*i.e.,* `Mirsh222` and `Mirsh95`, respectively). This is actually of benefit since a model with fewer features also requires lower computational resources. Moreover, this enhances the possibilities of finding a good model throughout the training stage since UPSO needs to tune fewer variables, *i.e.,* it needs to search within a smaller search domain. Therefore, we use these two features from this point onward.

The next path focuses on varying the number of training instances and the number of rules, leading to 90 parameter combinations. Figure 23 summarizes the median performance of the resulting SHHs. Bear in mind that this figure is built considering the inverse median value throughout the 30 runs of each parameter combination. This is done for the sake of figure readability. So, taller bars represent better performing SHHs when solving TE09.

Even though the SHH models trained with eight rules and 72 instances seem to work best, the data of all other

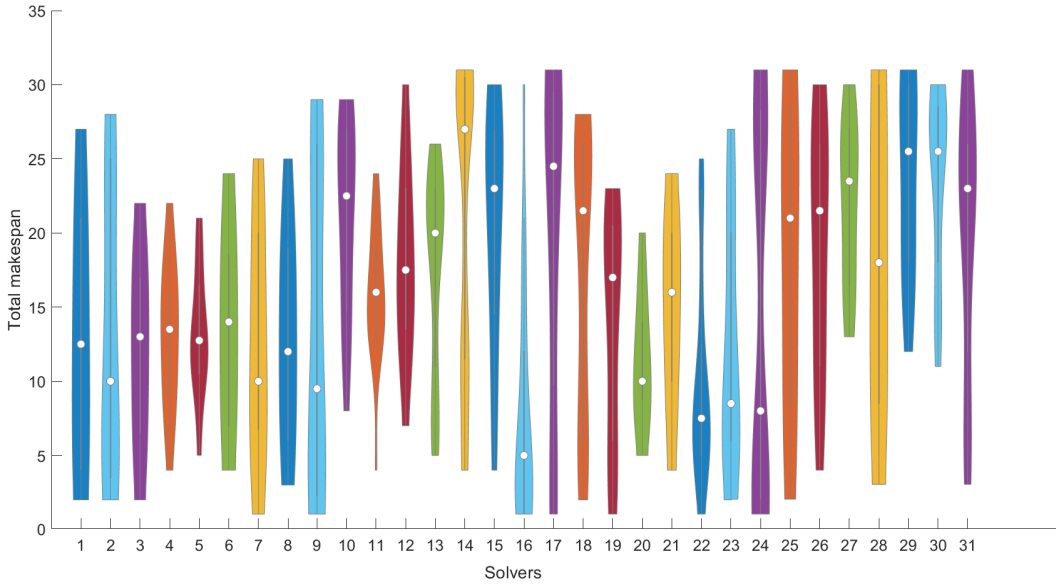


FIGURE 22. Performance distribution of SHHs trained using different feature subsets. Each SHH has 4 rules and was trained with TR09. Violins contain the ranking distribution of the median performance (30 runs) achieved by SHHs across all available testing sets. Feature combinations are given in Table 5.

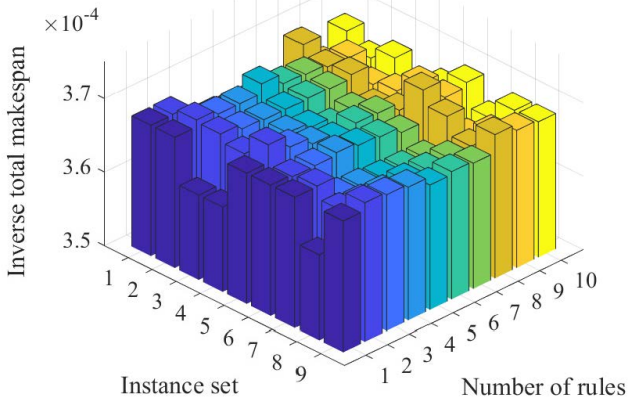


FIGURE 23. Median inverse total makespan yielded by SHHs trained with different number of training instances and rules (30 runs each), and when solving the testing set (TE09).

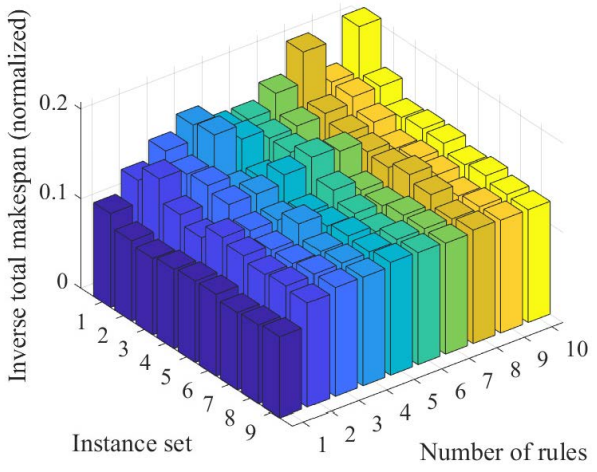
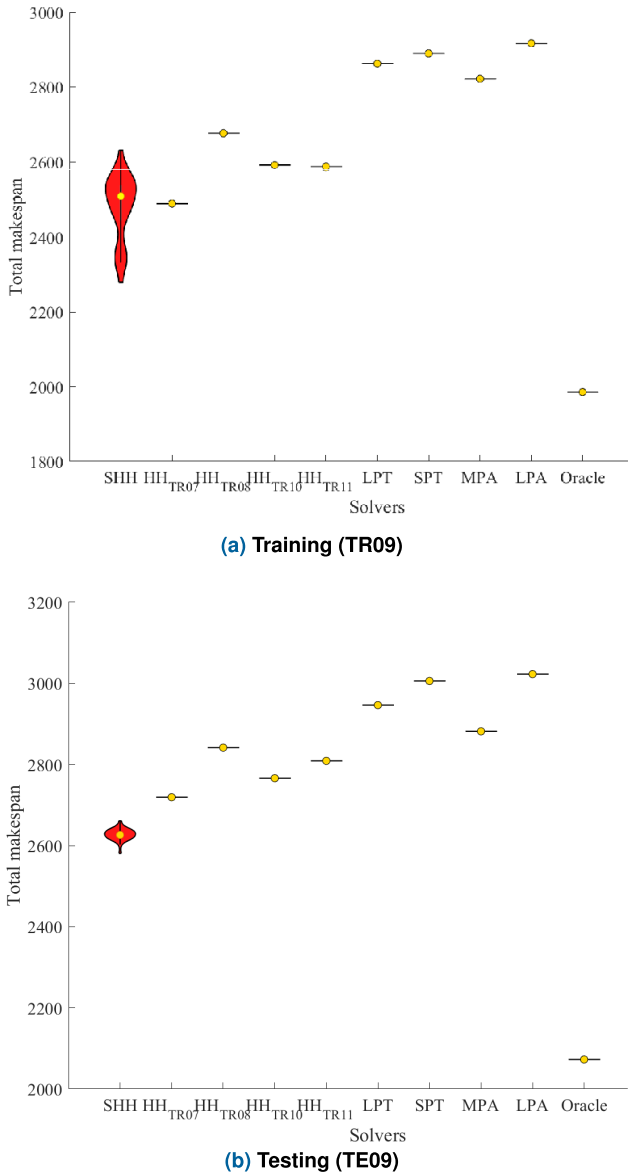


FIGURE 24. Median normalized inverse total makespan yielded by SHHs trained with different number of training instances and rules (30 runs each), and when solving their corresponding training sets (subsets from TR09).

models remain somewhat stable. So, let us now analyze what happens when solving the training set. Nonetheless, such a comparison is not straightforward. The reason is that in varying the number of training instances, the total makespan changes. Hence, SHHs trained with smaller sets would seem to perform better, which may not be accurate.

To remove this bias, we migrate to a normalized value of the previous metric. This value is calculated by subtracting the performance achieved by the oracle from the median performance of the SHH, and then dividing such a difference by the number of training instances. The resulting pattern is presented in Figure 24. This time, the figure shows two tendencies. First, performance improves when there are fewer training instances. This happens because it becomes easier to over-fit the SHH model to the training set.

The second tendency is that performance increases as the number of rules grow. This is favorable because a SHH with more rules can generate more complex action zones, which allows it to adapt to more varied scenarios. Nonetheless, caution must be taken as an excessively high number of rules may lead to overly complex models. This kind of model may overfit the training data more easily, and they are also more difficult to train since they have a higher number of design variables. In any case, we recommend training the model



(a) Training (TR09)

(b) Testing (TE09)

FIGURE 25. Performance distribution of SHHs, HHs, and heuristics (30 runs) when solving the training (TR09) and testing (TE09) subsets. Each SHH has access to the HHs and heuristics shown in the figure.

with a higher number of instances and rules, as it seems to generalize better.

D. CONFIRMATORY TESTS

As a final experiment, we train SHHs with access to both, HHs and heuristics (Figure 25). Comparing their performance against that of previous SHHs (*cf.* Figure 20) evidences that performance remains similar for both training and testing. However, it is also clear that the inclusion of the heuristics dampened the training performance. Moreover, the median performance of the SHH is worse than the median of HH_{TR07}, which is a previously unseen behavior. This suggests that combining both kinds of solvers (HHs and heuristics) might hold the model back.

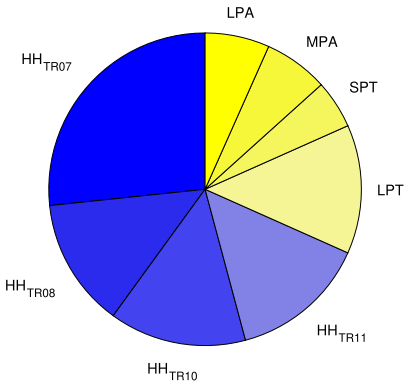


FIGURE 26. Solver composition of the SHHs created for the confirmatory stage (Figure 25). Regions shaded in yellow relate to heuristics and regions shaded in blue relate to hyper-heuristics.

One reason for the reduced performance stems from the fact that more solvers imply a larger search domain. Thus, the search becomes more difficult. Besides, these heuristics perform poorly when solving TE09 and TR09 (the testing and training set of this experiment). So, another reason is that the heuristics bind the performance of the SHHs, thus reducing overall performance. Therefore, unless we have a different set of heuristics, it seems better for the SHH to only use HHs.

Figure 26 shows the distribution of solvers regarding the SHHs trained in this stage. The proportions are obtained by counting the number of times each solver appears in a rule of the 30 SHHs. The pie clearly shows that about two-thirds of the rules target HHs (regions shaded in blue), with HH_{TR07} being the most popular one. Regarding heuristics (regions shaded in yellow), LPT is the most common one by far. As we can see in Figure 25a, the heuristics by themselves do not perform well when solving the training set (TR09). Therefore, the model seeks to disregard them, albeit not completely.

VI. CONCLUSION AND FUTURE WORK

In this work, we proposed a new approach for solving Job Shop Scheduling Problems (JSSPs): a dual-layered hyper-heuristic. Our goal was to propose a model akin to current Hyper-Heuristics (HHs) but with an extra layer of freedom. We feel that such a layer could be paramount for widening the generalization capability of hyper-heuristics. So, we called this model a Squared Hyper-Heuristic (SHH). To assess its feasibility, we pursued four testing stages. We first solved simple scenarios and compared the results against an existing model. Then, we explored 14 scenarios to determine the effect of the kind of instance used for training and the number of rules and solvers. Afterward, we analyze the influence of the features and the combination of the number of rules and training instances for one selected scenario. Finally, we delved into the idea of training a SHH that uses both HHs and heuristics.

Our feasibility tests hint at the idea of transferring knowledge from the HHs into a SHH. This may prove to be an

interesting path worth pursuing, as it could allow applying the ideas from transfer learning into hyper-heuristics. Moreover, it improves the reusability of a hyper-heuristic, as it can be trained for a specific purpose and then combined with another one to enhance their generalization capability.

In the exploratory section, we learned that increasing the number of rules leads to better performance. Moreover, if a particular solver excels over the training instances, the SHH will strive to mimic it. So, one should aim at having a balanced set of solvers for the training set. For this work, we achieved it by using bounded instances. As a result, the SHH distributed its rules across the available solvers, which allowed it to outperform them.

The detailed tests revealed that features play a crucial role and that having more features is not necessarily better. When using features `Mirsh222` and `Mirsh95`, our proposed model performed better than when using the entire set of features. So, we recommend using the reduced set in future works. We also analyzed the effect of changing the number of training instances and the number of rules for the SHH. Data suggest that using more rules usually translates into better training performance. However, caution must be taken as an excessively complex model could prove worthless and challenging to train.

Although our results are noteworthy, there are some practical implications to note when seeking to apply an SHH for real-life situations. One of them is the modeling of the problem as a JSSP. Another one is the set of instances available for training. Finally, we have the issue of the set of available solvers. In the first case, there can be external issues that deviate the problem from the one we considered here. Consider, for example, the issue of controlling traffic lights to minimize traffic jams. Even if it is straightforward to think of the travel time as the makespan of a JSSP, the stochastic behavior of car arrivals makes its implementation difficult. In this sense, each car could be considered a job, where operations are given by each traffic light the car encounters along its path, and each traffic light represents a machine. This, however, poses two issues. First, there is an interdependence in some machines (traffic lights at each crossing), which our model does not consider. Second, the JSSP changes the order in which jobs are scheduled, implying changing the ordering in which cars arrive. In the case of instances, the issue befalls to collecting real data where there can be numerous traffic lights and where external issues may affect the data (*e.g.,* extra lanes or issues down the road). This leads to the last issue, which appears because of the size of real-life instances. Even if our model assumes that traditional hyper-heuristics have been already trained, in a practical scenario, one must carry out such training. If the set of instances is huge, it will require considerable computing time.

Despite the good results we achieved, our work exhibits some drawbacks. Although the main reason for such drawbacks is the already substantial number of tests and related data, it is evident that several elements remain unexplored. These can be grouped in: experiment variety, cross-domain applicability, and model generalization. For example, it shall prove interesting to see what happens if we use instances with fixed feature values during the training of a SHH. In doing so, each HH could identify the set of solvers that works best at different regions of the feature domain. By combining these HHs, the SHH may recognize each solver better and improve its performance.

Dynamic JSSPs is a current trend and is a path worth pursuing. Consequently, we recommend analyzing the effect of higher-order hyper-heuristics when tackling dynamic JSSPs. Also, bear in mind that, in principle, it is possible to tackle multi-objective problems with our proposed approach. That is, if the HH solvers of the SHH are trained with a multi-objective approach, the SHH is prepared to tackle multi-objective goals as well. Naturally this requires the definition of a multi-objective fitness function and a multi-objective solver, such as a MOEA. Additionally, extensive tests should be run to assess the capability of the method, as this approach is currently theoretical.

Another topic to develop in the future is reducing the SHH model into a simplified HH, *e.g.,* with some translation algorithm. This could be done by looking at the action zones of a SHH. For example, if a SHH always chooses one or two HHs, then we may replace them with their component heuristics. This process will lead to a HH model since it will only have heuristics as solvers. Notwithstanding, this process must be tackled with caution as it may lead to unusually complex models.

We also need to figure out the possible applications this model might have and in which domains it shall prove the most useful. One application to keep in mind is the recombination of different techniques from the literature without having to train the solvers from scratch. This can be done by using them as solvers for the SHH while using the same training instances across all solvers. Conversely, one may use a new set of instances for training the SHH to learn how to combine solvers properly for this new goal. This would allow working in layers so that the final model exhibits an enhanced performance and generalization without incurring in excessive training costs.

Another idea that comes to mind is to delve deeper into the layered architecture. So far, we have added one extra layer to the hyper-heuristic model. Nevertheless, our proposed model can recursively add more layers until it finds the desired generalization level. Alas, this concept may be easier to describe than to develop, as it might end up being computationally infeasible.

An additional possibility is using the higher-order model to simplify the resulting regions of influence, thus streamlining the final model. This, of course, requires that the simplified solver outperforms the others. For example, consider a set of two complementary hyper-heuristics, which have the same shape for the regions of influence but with contradictory solvers. Let us consider, for simplicity, that one of these solvers excels across all instances. So, in creating the SHH,

the resulting model may learn to always use said heuristic. Nonetheless, it is necessary to run extensive tests to assess the reach of this idea.

Finally, it is important to disclose that, in this work, we consider a performance metric given by the total makespan. However, this is not the only feasible metric (nor we pretend it to be the best one). So, it should also prove interesting to analyze the effect of using different metrics.

REFERENCES

[1] J. Denzinger, M. Fuchs, and M. Fuchs, "High performance ATP systems by combining several AI methods," in *Proc. 15th Int. Joint Conf. Artif. Intell. (IJCAI)*, 1997, pp. 102–107.

[2] 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," *J. Oper. Res. Soc.*, vol. 64, no. 12, pp. 1695–1724, 2013.

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

[4] J. H. Drake, A. Kheiri, E. Özcan, and E. K. Burke, "Recent advances in selection hyper-heuristics," *Eur. J. Oper. Res.*, vol. 285, no. 2, pp. 405–428, Sep. 2020.

[5] D. Wei, F. Wang, and H. Ma, "Autonomous path planning of AUV in large-scale complex marine environment based on swarm hyper-heuristic algorithm," *Appl. Sci.*, vol. 9, no. 13, p. 2654, Jun. 2019.

[6] S. Yu, A. Song, and A. Aleti, "A study on online hyper-heuristic learning for swarm robots," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jun. 2019, pp. 2721–2728.

[7] H. Majeed and S. Naz, "Deja vu: A hyper heuristic framework with record and recall (2R) modules," *Cluster Comput.*, vol. 22, no. S3, pp. 7165–7179, May 2019.

[8] A. Adnan, A. Muhammed, A. A. A. Ghani, A. Abdullah, and F. Hakim, "Hyper-heuristic framework for sequential semi-supervised classification based on core clustering," *Symmetry*, vol. 12, no. 8, p. 1292, Aug. 2020.

[9] R. Cui, W. Han, X. Su, Y. Zhang, and F. Guo, "A multi-objective hyper heuristic framework for integrated optimization of carrier-based aircraft flight deck operations scheduling and resource configuration," *Aerosp. Sci. Technol.*, vol. 107, Dec. 2020, Art. no. 106346.

[10] J. Zhong, Z. Huang, L. Feng, W. Du, and Y. Li, "A hyper-heuristic framework for lifetime maximization in wireless sensor networks with a mobile sink," *IEEE/CAA J. Automatica Sinica*, vol. 7, no. 1, pp. 223–236, Jan. 2020.

[11] A. Toledo, M.-C. Riff, and B. Neveu, "A hyper-heuristic for the orienteering problem with hotel selection," *IEEE Access*, vol. 8, pp. 1303–1313, 2020.

[12] R. Bai, J. Blazewicz, E. K. Burke, G. Kendall, and B. McCollum, "A simulated annealing hyper-heuristic methodology for flexible decision support," *4OR*, vol. 10, no. 1, pp. 43–66, Mar. 2012.

[13] P. B. C. Miranda, R. B. C. Prudêncio, and G. L. Pappa, "H3AD: A hybrid hyper-heuristic for algorithm design," *Inf. Sci.*, vol. 414, pp. 340–354, Nov. 2017.

[14] J. M. Cruz-Duarte, I. Amaya, J. C. Ortiz-Bayliss, S. E. Conant-Pablos, and H. Terashima-Marin, "A primary study on hyper-heuristics to customise metaheuristics for continuous optimisation," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8.

[15] J. M. Cruz-Duarte, I. Amaya, J. C. Ortiz-Bayliss, S. E. Conant-Pablos, H. Terashima-Marín, and Y. Shi, "Hyper-heuristics to customise metaheuristics for continuous optimisation," *Swarm Evol. Comput.*, vol. 66, Oct. 2021, Art. no. 100935.

[16] S. S. Choong, L.-P. Wong, and C. P. Lim, "Automatic design of hyper-heuristic based on reinforcement learning," *Inf. Sci.*, vols. 436–437, pp. 89–107, Apr. 2018.

[17] N. R. Sabar, M. Ayob, G. Kendall, and R. Qu, "Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 309–325, Jun. 2015.

[18] M. Sanchez, J. M. Cruz-Duarte, J. C. Ortiz-Bayliss, H. Ceballos, H. Terashima-Marin, and I. Amaya, "A systematic review of hyper-heuristics on combinatorial optimization problems," *IEEE Access*, vol. 8, pp. 128068–128095, 2020.

[19] J. J. van Hoorn, "The current state of bounds on benchmark instances of the job-shop scheduling problem," *J. Scheduling*, vol. 21, no. 1, pp. 127–128, Feb. 2018.

[20] G. Da Col and E. Teppan, "Google vs IBM: A constraint solving challenge on the job-shop scheduling problem," *Electron. Proc. Theor. Comput. Sci.*, vol. 306, pp. 259–265, Sep. 2019.

[21] L. Hernández-Ramírez, J. Frausto-Solis, G. Castilla-Valdez, J. J. González-Barbosa, J. D. Terán-Villanueva, and M. L. Morales-Rodríguez, "A hybrid simulated annealing for job shop scheduling problem," *Int. J. Combinat. Optim. Problems Informat.*, vol. 10, no. 1, pp. 6–15, 2019.

[22] I. González-Rodríguez, J. Puente, J. J. Palacios, and C. R. Vela, "Multi-objective evolutionary algorithm for solving energy-aware fuzzy job shop problems," *Soft Comput.*, vol. 24, no. 21, pp. 16291–16302, Nov. 2020.

[23] J. Liang, Y.-H. Zhu, Y.-Z. Luo, J.-C. Zhang, and H. Zhu, "A precedence-rule-based heuristic for satellite onboard activity planning," *Acta Astronautica*, vol. 178, pp. 757–772, Jan. 2021.

[24] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Learning iterative dispatching rules for job shop scheduling with genetic programming," *Int. J. Adv. Manuf. Technol.*, vol. 67, nos. 1–4, pp. 85–100, Jul. 2013.

[25] C. Y. Lee, S. Piramuthu, and Y. K. Tsai, "Job shop scheduling with a genetic algorithm and machine learning," *Int. J. Prod. Res.*, vol. 35, no. 4, pp. 1171–1191, 1997.

[26] S. Olafsson and X. Li, "Learning effective new single machine dispatching rules from optimal scheduling data," *Int. J. Prod. Econ.*, vol. 128, no. 1, pp. 118–126, Nov. 2010.

[27] C. W. Pickardt, T. Hildebrandt, J. Branke, J. Heger, and B. Scholz-Reiter, "Evolutionary generation of dispatching rule sets for complex dynamic scheduling problems," *Int. J. Prod. Econ.*, vol. 145, no. 1, pp. 67–77, 2013.

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

[29] E. Lara-Cardenas, A. Silva-Galvez, J. C. Ortiz-Bayliss, I. Amaya, J. M. Cruz-Duarte, and H. Terashima-Marin, "Exploring reward-based hyper-heuristics for the job-shop scheduling problem," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Dec. 2020, pp. 1–8.

[30] C.-C. Wu, D. Bai, J.-H. Chen, W.-C. Lin, L. Xing, J.-C. Lin, and S.-R. Cheng, "Several variants of simulated annealing hyper-heuristic for a single-machine scheduling with two-scenario-based dependent processing times," *Swarm Evol. Comput.*, vol. 60, Feb. 2021, Art. no. 100765.

[31] H. Fan, H. Xiong, and M. Goh, "Genetic programming-based hyper-heuristic approach for solving dynamic job shop scheduling problem with extended technical precedence constraints," *Comput. Oper. Res.*, vol. 134, Oct. 2021, Art. no. 105401.

[32] W. Bouazza, Y. Sallez, and D. Trentesaux, "Dynamic scheduling of manufacturing systems: A product-driven approach using hyper-heuristics," *Int. J. Comput. Integr. Manuf.*, vol. 34, pp. 1–25, Jul. 2021.

[33] F. Garza-Santisteban, I. Amaya, J. Cruz-Duarte, J. C. Ortiz-Bayliss, E. Ozcan, and H. and Terashima-Marin, "Exploring problem state transformations to enhance hyper-heuristics for the job-shop scheduling problem," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8.

[34] A. Vela, J. M. Cruz-Duarte, J. C. Ortiz-Bayliss, and I. Amaya, "Tailoring job shop scheduling problem instances through unified particle swarm optimization," *IEEE Access*, vol. 9, pp. 66891–66914, 2021.

[35] V. López, I. Triguero, C. J. Carmona, S. García, and F. Herrera, "Addressing imbalanced classification with instance generation techniques: IPADE-ID," *Neurocomputing*, vol. 126, pp. 15–28, Feb. 2014.

[36] 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," *J. Intell. Manuf.*, vol. 30, no. 4, pp. 1809–1830, Apr. 2019.

[37] T. Borreguero-Sanchidrián, R. Pulido, Á. García-Sánchez, and M. Ortega-Mier, "Flexible job shop scheduling with operators in aeronautical manufacturing: A case study," *IEEE Access*, vol. 6, pp. 224–233, 2017.

[38] C.-C. Wu, D. Bai, J.-H. Chen, W.-C. Lin, L. Xing, J.-C. Lin, and S.-R. Cheng, "Several variants of simulated annealing hyper-heuristic for a single-machine scheduling with two-scenario-based dependent processing times," *Swarm Evol. Comput.*, vol. 60, Feb. 2021, Art. no. 100765.

[39] M. M. Ahmadian, A. Salehipour, and T. C. E. Cheng, "A meta-heuristic to solve the just-in-time job-shop scheduling problem," *Eur. J. Oper. Res.*, vol. 288, no. 1, pp. 14–29, Jan. 2021.

[40] G. A. Rolim and M. S. Nagano, "Structural properties and algorithms for earliness and tardiness scheduling against common due dates and windows: A review," *Comput. Ind. Eng.*, vol. 149, Nov. 2020, Art. no. 106803.

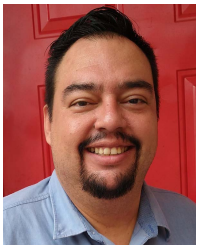
[41] M. E. Bruni, S. Khodaparasti, and E. Demeulemeester, "The distributionally robust machine scheduling problem with job selection and sequence-dependent setup times," *Comput. Oper. Res.*, vol. 123, Nov. 2020, Art. no. 105017.
[42] M. Mahmoodjanloo, R. Tavakkoli-Moghaddam, A. Baboli, and A. Bozorgi-Amiri, "Flexible job shop scheduling problem with reconfigurable machine tools: An improved differential evolution algorithm," *Appl. Soft Comput.*, vol. 94, Sep. 2020, Art. no. 106416.
[43] D. Yüksel, M. F. Taşgetiren, L. Kandiller, and L. Gao, "An energy-efficient bi-objective no-wait permutation flowshop scheduling problem to minimize total tardiness and total energy consumption," *Comput. Ind. Eng.*, vol. 145, Jul. 2020, Art. no. 106431.
[44] X. Li and S. Olafsson, "Discovering dispatching rules using data mining," *J. Scheduling*, vol. 8, no. 6, pp. 515–527, Dec. 2005.
[45] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, Jul. 2008.
[46] C. Ansótegui, J. Gabàs, Y. Malitsky, and M. Sellmann, "MaxSAT by improved instance-specific algorithm configuration," *Artif. Intell.*, vol. 235, pp. 26–39, Jun. 2016.
[47] J. M. Cruz-Duarte, I. Amaya, J. C. Ortiz-Bayliss, S. E. Conant-Pablos, and H. Terashima-Marin, "A primary study on hyper-heuristics to customise metaheuristics for continuous optimisation," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8.
[48] F. Garza-Santisteban, I. Amaya, J. Cruz-Duarte, J. C. Ortiz-Bayliss, E. Ozcan, and H. Terashima-Marin, "Exploring problem state transformations to enhance hyper-heuristics for the job-shop scheduling problem," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8.
[49] C. Yu, P. Andreotti, and Q. Semeraro, "Multi-objective scheduling in hybrid flow shop: Evolutionary algorithms using multi-decoding framework," *Comput. Ind. Eng.*, vol. 147, Sep. 2020, Art. no. 106570.
[50] 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 *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jun. 2019, pp. 57–64.
[51] P. Taylor, "Addressing the gap in scheduling research: A review of optimization and heuristic methods in production scheduling Addressing the gapin scheduling research: A review of optimization and heuristic methods in production scheduling," *Manuf. Eng.*, vol. 31, pp. 37–41, Jan. 1993.
[52] C. Mencía, M. R. Sierra, and R. Varela, "Depth-first heuristic search for the job shop scheduling problem," *Ann. Oper. Res.*, vol. 206, no. 1, pp. 265–296, Jul. 2013.
[53] K. Parsopoulos and M. Vrahatis, "UPSO: A unified particle swarm optimization scheme," in *Proc. Int. Conf. Comput. Methods Sci. Eng. (ICCMSE)*, Jan. 2019, pp. 868–873.
[54] M. Klusch, M. Pechoucek, and A. Polleres, *Artificial Intelligence: Methodology, Systems, and Applications* (Notes in Computer Science), vol. 4183. Berlin, Germany: Springer, 2006.
[55] M. Clerc and J. Kennedy, "The particle swarm–explosion, stability, and convergence in a multidimensional complex space," *IEEE Trans. Evol. Comput.*, vol. 6, no. 1, pp. 58–73, Aug. 2002.
[56] E. S. Peer, F. van den Bergh, and A. P. Engelbrecht, "Using neighbourhoods with the guaranteed convergence PSO," in *Proc. IEEE Swarm Intell. Symp. (SIS)*, Aug. 2003, pp. 235–242.
[57] P. Cowling, G. Kendall, and E. Soubeiga, "A hyperheuristic approach to scheduling a sales summit," in *Practice and Theory of Automated Timetabling III* (Lecture Notes in Computer Science), vol. 2079, E. Burke and W. Erben, Eds. Berlin, Germany: Springer, 2001, doi: 10.1007/3-540-44629-X_11.
[58] K. Sim and E. Hart, "An improved immune inspired hyper-heuristic for combinatorial optimisation problems," in *Proc. Annu. Conf. Genetic Evol. Comput.*, New York, NY, USA, Jul. 2014, pp. 121–128.
[59] S. Nguyen, M. Zhang, and M. Johnston, "A genetic programming based hyper-heuristic approach for combinatorial optimisation," in *Proc. 13th Annu. Conf. Genetic Evol. Comput. (GECCO)*, New York, NY, USA, 2011, pp. 1299–1306.
[60] N. R. Sabar and G. Kendall, "Population based Monte Carlo tree search hyper-heuristic for combinatorial optimization problems," *Inf. Sci.*, vol. 314, pp. 225–239, Sep. 2015.
[61] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "A classification of hyper-heuristic approaches: Revisited," *Int. Ser. Oper. Res. Manage. Sci.*, vol. 272, pp. 453–477, 2019.
[62] S. Mirshekarian and D. N. Šormaz, "Correlation of job-shop scheduling problem features with scheduling efficiency," *Expert Syst. Appl.*, vol. 62, pp. 131–147, Nov. 2016.

ALONSO VELA was born in Piedras Negras, Coahuila, Mexico, in 1996. He received the B.Sc. degree in biomedical engineering and the M.Sc. degree in computer science from the Tecnologico de Monterrey, in 2019 and 2021, respectively.

In 2018, he worked at O-I Packaging Solutions as a Systems Engineer, where he was in-charge of the Quality Laboratory. His research interests include machine learning, artificial intelligence, engineering design, and combinatorial optimization problems solved through the field of hyper-heuristics.

JORGE M. CRUZ-DUARTE (Member, IEEE) was born in Ocaña, N.S., Colombia, in 1990. He received the B.Sc. and M.Sc. degrees in electronic engineering from the Universidad Industrial de Santander, Bucaramanga, Santander, Colombia, in 2012 and 2015, respectively, and the Ph.D. degree in electrical engineering from the Universidad de Guanajuato, Mexico, in 2018.

He was a Postdoctoral Fellow with the Research Group with Strategic Focus in Intelligent Systems, Tecnológico de Monterrey, Mexico, from 2019 to 2021, where he is currently a Research Professor with the School of Engineering and Sciences. His research interests include data science, optimization, mathematical methods, thermodynamics, digital signal processing, electronic thermal management, and fractional calculus.



JOSÉ CARLOS ORTIZ-BAYLISS (Member, IEEE) was born in Culiacan, Sinaloa, Mexico, in 1981. He received the B.Sc. degree in computer engineering from the Universidad Tecnologica de la Mixteca, in 2005, the M.Sc. degree in computer science from the Tecnologico de Monterrey, in 2008, the Ph.D. degree from the Tecnologico de Monterrey, in 2011, the M.Ed. degree from the Universidad del Valle de Mexico, in 2017, the B.Sc. degree in project management from the Universidad Virtual del Estado de Guanajuato, in 2019, and the M.Ed.A. degree from the Instituto de Estudios Universitarios, in 2019.

He is currently an Assistant Research Professor with the School of Engineering and Sciences, Tecnologico de Monterrey. His research interests include computational intelligence, machine learning, heuristics, metaheuristics, and hyper-heuristics for solving combinatorial optimization problems. He is a member of the Mexican National System of Researchers, the Mexican Academy of Computing, and the Association for Computing Machinery.

IVAN AMAYA (Senior Member, IEEE) was born in Bucaramanga, Santander, Colombia, in 1986. He received the B.Sc. degree in mechatronics engineering from the Universidad Autónoma de Bucaramanga, in 2008, and the Ph.D. degree in engineering from the Universidad Industrial de Santander, in 2015.

From 2016 to 2018, he was a Postdoctoral Fellow with the Research Group with Strategic Focus in Intelligent Systems, Tecnologico de Monterrey, where he has been a Research Professor with the School of Engineering and Sciences, since 2018. His research interests include numerical optimization of both, continuous and discrete problems, through the application of heuristics, metaheuristics, hyper-heuristics, and finding new ways of using feature transformations for improving hyper-heuristic performance. He is a member of the Mexican National System of Researchers, the Mexican Academy of Computing, and the Association for Computing Machinery.