






# L-SAGA: A Learning Hyper-Heuristic Architecture for the Permutation Flow-Shop Problem

Younes Boukacem<sup>1</sup> <sup>a</sup>, Hatem M. Abdelmoumen<sup>1</sup> <sup>b</sup>, Hodhaifa Benouaklil<sup>1</sup> <sup>c</sup>, Samy Ghebache<sup>1</sup>,  
Boualem Hamroune<sup>1</sup>, Mohammed Tirichine<sup>1</sup> <sup>d</sup>, Nassim Ameer<sup>1</sup> <sup>e</sup> and Malika Bessedik<sup>1,2</sup>

<sup>1</sup>*Ecole Nationale Supérieure d'Informatique (ESI), BP 68M - 16270 Oued Smar, Algiers, Algeria*

<sup>2</sup>*Laboratoire des Méthodes de Conception de Systèmes (LMCS), Ecole Nationale Supérieure d'Informatique (ESI),  
BP 68M - 16270 Oued Smar, Alger, Algeria*

{ky\_boukacem, kh\_abdelmoumen, kh\_benouaklil, ks\_ghebache, kb\_hamroune, km\_tirichine, kn\_ameur, m\_bessedik}@esi.dz

**Keywords:** Flow-Shop Permutation Problem, Hyper-Heuristic, Simulated Annealing, Genetic Algorithm.

**Abstract:** The permutation flow-shop problem or PFSP consists in finding the optimal launching sequence of jobs to be sequentially executed along a chain of machines, each job having different execution times for each machine, in order to minimize the total completion time. As an NP-hard problem, PFSP has significant applications in large-scale industries. In this paper we present L-SAGA, a generative hyper-heuristic designed for finding optimal to sub-optimal solutions for the PFSP. L-SAGA combines a high level simulated annealing with a learning component and a low level PFSP adapted genetic algorithm. The performed tests on various benchmarks indicate that, while our method had competitive results on some small and medium size benchmarks thus showing interesting potential, it still requires further improvement to be fully competitive on larger and more complex benchmarks.

## 1 INTRODUCTION


The Permutation Flow Shop Problem (PFSP) consists in finding an optimal schedule for executing jobs along a chain of machines under a set of constraints (see Section 2), with the aim to minimize the total completion time also known as the makespan. This NP-hard problem is vital in most industries, where efficient scheduling boosts productivity and can induce major cost savings.


Early works addressed the PFSP using heuristics such as the NEH (Nawaz et al., 1983), the CDS (Campbell et al., 1970), and Palmer's (Palmer, 1965) heuristics. More recent works used metaheuristics like the immunity-based hybrid genetic algorithm (Bessedik et al.2016), the improved genetic immune algorithm with vaccinated offspring (Tayeb et al., 2017), the hybrid genetic algorithm and bottleneck shifting (Gao et al., 2007), and the iterated greedy algorithm (Ruiz and Stutzle, 2007). While heuris-


tics are often instance-specific and may not generalize well across instances, metaheuristics are adaptable but sensitive to hyperparameters, affecting their ability and usability to solve complex problems.


Hyper-heuristics offer higher abstraction and adaptability than traditional (meta)heuristics. They can either be selective, i.e. select appropriate heuristics from a set, or generative, i.e. generate new ones by combining existing components, see (Burke et al., 2010). We chose hyper-heuristics for their scalability and potential to improve performance across diverse problem instances.


In recent years, hyper-heuristics have gained attention for addressing NP-hard optimization problems, including the PFSP. (Garza-Santisteban et al., 2019) highlights the effectiveness of Simulated Annealing (SA) in training selection hyper-heuristics through stochastic optimization to boost performance. Additionally, (Garza-Santisteban et al., 2020) incorporates feature transformations to improve the training phase, enhancing state differentiation while also using SA in perturbative selection hyper-heuristics. A new hyper-heuristic method, HHGA, was introduced by (Bacha et al., 2019), which applies genetic algorithms to dynamically generate customized configurations for each PFSP instance. Moreover, (Alekseeva

<sup>a</sup>  <https://orcid.org/0009-0001-5896-3227>

<sup>b</sup>  <https://orcid.org/0009-0006-1459-2723>

<sup>c</sup>  <https://orcid.org/0009-0002-1239-2606>

<sup>d</sup>  <https://orcid.org/0009-0003-9205-6158>

<sup>e</sup>  <https://orcid.org/0009-0009-1120-6286>

et al., 2017) proposed a parallel multi-core hyper-heuristic based on the Greedy Randomized Adaptive Search Procedure (GRASP), automatically evaluating 315 configurations to identify the best one, with parallel computing improving efficiency.

This paper introduces L-SAGA (Learning Simulated Annealing Genetic Algorithm): a generative hyper-heuristic designed to efficiently explore the solution space of the PFSP. L-SAGA combines a PFSP-specific genetic algorithm for the low level, with a simulated annealing hyperparameter tuner equipped with a learning component for the high level. The hybrid use of these two metaheuristics has shown promising results on some small and medium size benchmarks, but still requires improvement for larger and more complex instances to be fully competitive.

The rest of the paper is organized as follows: Section 2 formulates the PFSP. Section 3 describes the L-SAGA architecture. Section 4 presents a performance comparison between L-SAGA and state-of-the-art methods, along with an empirical analysis of L-SAGA's behavior in different testing scenarios. The paper concludes in Section 5, discussing L-SAGA's limitations and potential areas for improvement.

## 2 FORMAL PROBLEM DEFINITION

The PFSP with the objective of minimizing the makespan can be formally defined as follows. A finite set  $J$  consisting of  $n$  jobs  $J = \{J_1, J_2, \dots, J_n\}$  need to be processed on a finite set  $M$  of  $m$  machines  $M = \{1, 2, \dots, m\}$  in a sequential manner. The sequence in which jobs are processed is consistent across all machines; thus, the processing order on the first machine is maintained throughout the remaining machines.

Several standard assumptions are made regarding this problem (Pinedo, 2012):

- All  $n$  jobs are independent and available for processing at time zero.
- All  $m$  machines are continuously available without any interruptions.
- Each machine can process at most one job at a time, and each job can be processed on only one machine at a time (capacity constraints).
- The processing of any given job  $J_j$  cannot be interrupted, implying no preemption is allowed.
- Setup and removal times of jobs on machines are included in the processing times or negligible.
- If a machine required for the next operation of a job is busy, the job can wait in an unlimited queue.

The objective of the PFSP is to find a sequence, i.e., a permutation of job indices  $\{1, 2, \dots, n\}$ , that minimizes the makespan  $C_{\max}$  (Miller et al., 1967). The makespan is the total time required to complete all jobs on all machines. This problem is commonly denoted as  $n|m|P|C_{\max}$  or  $F_m|prmu|C_{\max}$  (Pinedo, 2012), where  $n$  and  $m$  represent the number of jobs and machines respectively,  $prmu$  indicates that only permutation schedules are allowed, and  $C_{\max}$  refers to the criterion of makespan minimization. Notably, the problem  $F_m|prmu|C_{\max}$  is known to be NP-complete in the strong sense when  $m > 3$  (Garey et al., 1976).

Consider a permutation  $\pi = \{\pi_1, \pi_2, \dots, \pi_n\}$  where  $n$  jobs are sequenced through  $m$  machines, and  $\pi_j$  denotes the  $j$ -th job in the sequence. Let  $p_{ij}$  represent the processing time of job  $j$  on machine  $i$ , and  $c_{ij}$  the time at which machine  $i$  finishes processing job  $j$ . The mathematical model for the completion time of the sequence  $\pi$  is formulated as follows:

$$c_{1,\pi_1} = p_{1,\pi_1} \quad (1)$$

$$c_{1,\pi_j} = c_{1,\pi_{j-1}} + p_{1,\pi_j} \quad \text{for } j = 2, \dots, n \quad (2)$$

$$c_{i,\pi_1} = c_{i-1,\pi_1} + p_{i,\pi_1} \quad \text{for } i = 2, \dots, m \quad (3)$$

$$c_{i,\pi_j} = \max(c_{i,\pi_{j-1}}, c_{i-1,\pi_j}) + p_{i,\pi_j} \quad \text{for } j = 2, \dots, n \text{ and } i = 2, \dots, m \quad (4)$$

Thus, the makespan  $C_{\max}$  of a permutation  $\pi$  can be formally defined as:

$$C_{\max}(\pi) = c_{m,\pi_n} \quad (5)$$

Therefore, the PFSP with the makespan criterion aims to find the optimal permutation  $\pi^*$  within the set of all permutations  $\Pi$  such that:

$$C_{\max}(\pi^*) \leq C_{\max}(\pi) \quad \forall \pi \in \Pi \quad (6)$$

In this study, we consider a flexible flowshop configuration that can adapt to different numbers of machines and jobs. The setup ensures compatibility with various problem sizes and maintains adherence to the aforementioned constraints.

## 3 MODEL ARCHITECTURE

As a generative hyper-heuristic, L-SAGA general architecture is composed of two levels as depicted in Figure 1: A low level which comprises a genetic algorithm (GA) specifically designed for exploring the space of the PFSP job launching sequences, and a high level which comprises a simulated annealing algorithm (SA) that will explore the space of hyperparameters of the low level's genetic algorithm with the help of a top level learning component (LC) whose main data structure is a "podium bank" which is described further.

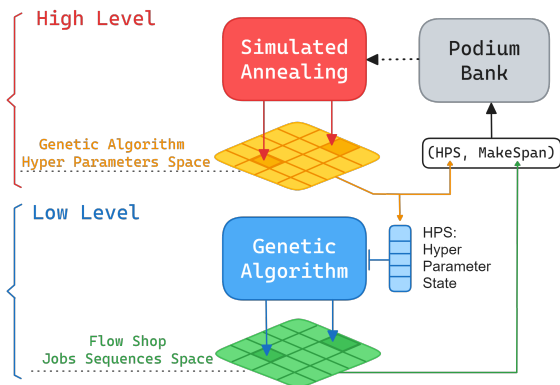


Figure 1: L-SAGA general architecture.

### 3.1 Low-Level Genetic Algorithm

We designed a PFSP-adapted GA for the low level where each chromosome (or individual) of the population is a scheduling sequence of the jobs. Each gene represents a job in the sequence with the positions of the genes determining the jobs order. An individual's fitness is computed as its makespan. The architecture and operational mechanisms of the GA, which include initialization, selection, crossover, mutation, and replacement strategies, make the population evolve towards optimal solutions, as visually summarized in Figure 2.

1. **Initialization.** Multiple techniques are possible for initialization to ensure diversity and quality. Methods like the NEH, CDS, and Palmer heuristics, as well as combinations of these, are used. Full random initialization is also an option. Each technique generates job sequences (chromosomes) that are evaluated by their makespan.
2. **Selection:** The selection process determines which individuals from the current population are chosen to form a mating pool for the next generation. Various strategies are available, including roulette wheel, rank-based, elitist, tournament, and random selection. These methods ensure a diverse and high-quality mating pool by favoring better-performing solutions while maintaining genetic diversity.
3. **Crossover.** Crossover is the primary genetic operator that combines pairs of parents to produce offspring. The algorithm supports uniform crossover and  $k$ -point crossover. Each technique involves swapping job sequences between parents at specific points, generating new candidate solutions that inherit characteristics from both parents as depicted in Figure 3. The crossover rate ( $P_c$ ) controls the likelihood of crossover events, thereby influ-

encing the genetic diversity and convergence rate of the population.

4. **Mutation.** Mutation introduces genetic variation into the population by randomly altering genes in the offspring. This operator helps to maintain genetic diversity and prevents premature convergence and local optima trapping. Permutation-based mutation is employed, where jobs within a sequence are swapped, ensuring feasibility of the solutions. The mutation rate ( $P_m$ ) defines the probability of mutation for each chromosome, balancing the need for exploration of new solutions and the preservation of high-quality individuals.
5. **Replacement.** The replacement strategy governs how the new generation is formed from the current population and the newly created offspring. This step ensures that the population evolves toward better solutions while maintaining necessary genetic diversity to avoid stagnation. Various methods are employed, including replacing all parents with offspring, replacing the worst solutions in the population, or selecting the best individuals between parents and offspring. Moreover, all selection methods mentioned in the selection phase can be used here to ensure a diverse and high-quality population for the next generation.

The genetic algorithm iterates through successive generations until one of the termination criteria is met. These criteria include a predefined number of iterations or maximum stagnation, defined as the number of consecutive generations without improvement. The solution with the best makespan identified throughout the iterations is ultimately returned as the optimal schedule.

### 3.2 High-Level Simulated Annealing with a Learning Component

#### 3.2.1 Simulated Annealing

As depicted in Figure 1, L-SAGA's high-level SA will explore the hyperparameters space for the low level GA, requiring an appropriate encoding of the search space. A natural encoding was adopted, representing each hyperparameter as a dimension in a vector called HPS for HyperParameter State, with two types of dimensions: qualitative (Qual.Dims) and quantitative (Quant.Dims). The general SA algorithm, as shown in Figure 4, begins by receiving key hyperparameters:

- **T.** The initial temperature of the simulated annealing.

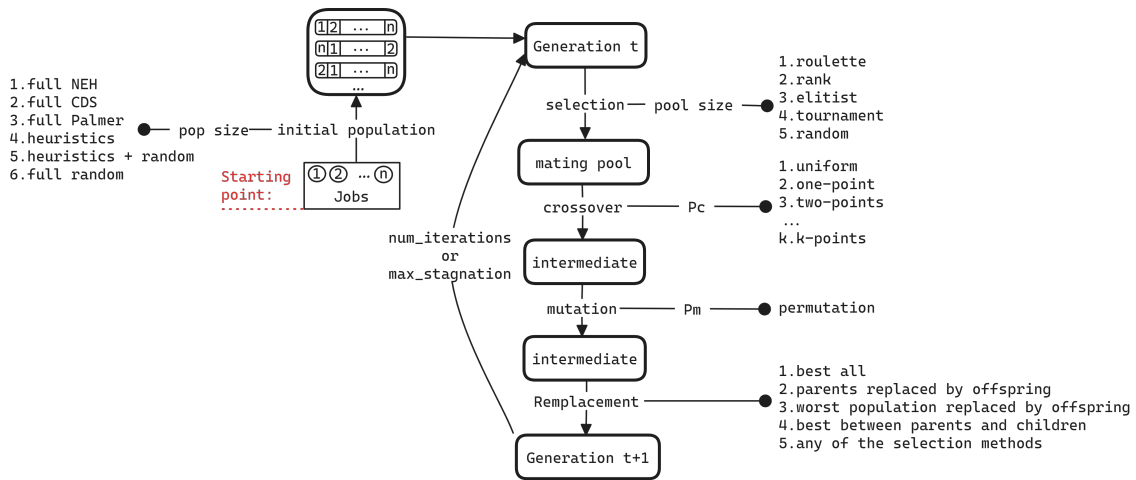


Figure 2: Low Level Genetic Algorithm.

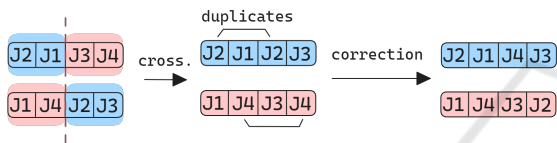


Figure 3: Crossover operator on job sequences.

- **Tmin.** The minimum temperature under which the algorithm stops.
- **$\alpha$ .** The geometric decay ( $< 1$ ) coefficient for the temperature.
- **TemperatureSession.** The number of iterations during which the temperature remains fixed.
- **JumpRate.** The probability of increasing the temperature (jumping) instead of decreasing it at the end of a 'TemperatureSession'.
- **JumpRatio.** The geometric jump coefficient ( $> 1$ ) for the temperature in case of jumping at the end of a 'TemperatureSession'.
- **RetentionRate.** The percentage of population to pass from the previous execution of the GA to the next iteration.

Starting from a randomly generated HPS, the SA component generates a neighboring HPS at each iteration, passes it to the genetic algorithm, and records the resulting makespan. The neighbor HPS and makespan are added to the podium bank for future iterations. If the makespan improves, the neighbor HPS becomes the current HPS; otherwise, it may be accepted with a probability that decreases as the temperature lowers and makespan degradation increases. The temperature adjusts based on the JumpRate probability and the TemperatureSession counter and the process ends when the temperature falls below a min-

imum or after a set number of iterations, returning the best makespan found.

### 3.2.2 Learning Component

L-SAGA proposes a learning component that can be used to help the high level's SA component in the search process of the optimal HPS. This add-on learning component is centered around the podium bank data structure which, in conjunction with a selection algorithm, aims at improving the exploitation capacity of the SA component during the neighbor HPS generation step which, in the vanilla simulated annealing algorithm, is completely random during the whole search process. The architecture of the LC is depicted in Figure 5.

The podium bank is a finite size priority queue data structure which will store during the whole search process the top P (P being a hyperparameter) encountered HPS's according to their makespan, thus constituting a memory of the HPS configurations that provided the best results. When generating a neighbor HPS, a number "variety degree" of dimensions in the current HPS will be marked as to "forcibly change" in the neighborhood space i.e. : their values won't remain in the possible neighbors. The dimensions that will be marked so are randomly chosen. Then the set of all possible neighbors (considering the variety degree) is generated: Each qualitative dimension is randomly swapped with a given number (Qual. Dim. Neighbor Window) of other possible values, and in the same way each quantitative dimension will be swapped with the other possible values in a given neighborhood window (Quant. Dim. Neighbor Window). While the variety degree helps in augmenting the exploration potential of the algorithm, the two latter hyperparameters helps in controlling the size of

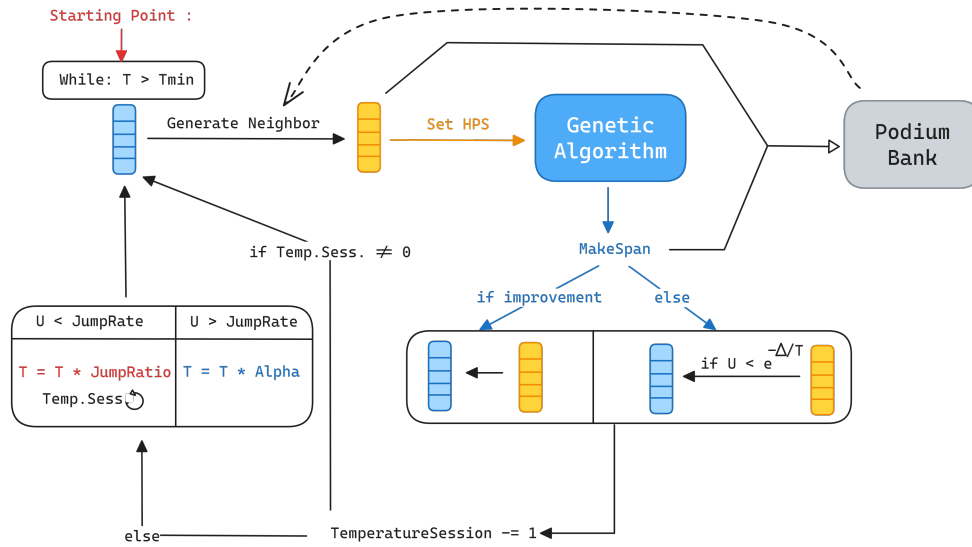


Figure 4: Top level simulated annealing component with the podium bank add-on.

the explored neighborhood which may help in exploration if large but at the cost of performance overhead.

Once the neighbor set is generated, two scenarios are possible: either a neighbor is chosen randomly according to a uniform distribution, this scenario happens with a probability that decreases as temperature decreases and is controlled by the *beta* hyperparameter (which when increased increases the uniform choice probability for more exploration). Otherwise the podium bank will be used to attribute a score to each neighbor which will determine its probability of being chosen according to equation 7:

$$P(\text{choosing neighbor}_i) = \frac{Score_i}{\sum_{n=1}^k Score_n} \quad (7)$$

The score of each neighbor represents its quality in the sense of the proximity it has with the previously best observed **HPS**s stored in the podium bank. The formula for calculating the said "proximity" is a weighted average of the "similarities" of the neighbor with the various podium bank **HPS**s.

$$Score_i = \frac{\sum_{p=1}^P makespan_p^{-1} \cdot e^{Sim(neighbor_i, PodiumHPS_p)}}{\sum_{p=1}^P makespan_p^{-1}} \quad (8)$$

$$Sim(neighbor_i, PodiumHPS_p) = \sum_{d \in Qual.Dims} \delta(neighbor_i[d] = PodiumHPS_p[d]) - \sum_{d \in Quant.Dims} \frac{|neighbor_i[d] - PodiumHPS_p[d]|}{MaxVal[d] - MinVal[d]} \quad (9)$$

where the  $\delta$  function returns 0 in case of equality and 1 otherwise.

## 4 RESULTS

We present here various results obtained from the execution of L-SAGA on the Taillard benchmarks (Taillard, 1993). The executions were all performed using manually tuned hyper-parameters for the high level such as a podium bank size of 5, a variety degree of 5, an initial and final temperature of 1 and .008 respectively, and a maximum number of iterations of 170. These values were chosen accounting for both solutions quality and available hardware resources at the time of this publication. More implementation details are available on the dedicated GitHub repository<sup>1</sup>. The testing computer had an intel i5 11th generation with 16 GB of RAM.

### 4.1 Performance Comparison of L-SAGA with State-of-the-Art Methods

The Taillard benchmark for the PFSP presents with 12 configurations of number of jobs  $\times$  number of machines, each of which has 10 different instances. For each instance, we measure the performance of L-SAGA using the **Relative Percentage Deviations** (RPD) presented in equation 10:

$$RPD_i = \frac{C_{max} - UpperBound_i}{UpperBound_i} \times 100 \quad (10)$$

Where  $C_{max}$  is the makespan of the best solution found by L-SAGA and  $UpperBound_i$  is the makespan

<sup>1</sup><https://github.com/YounesBoukacem/L-SAGA>

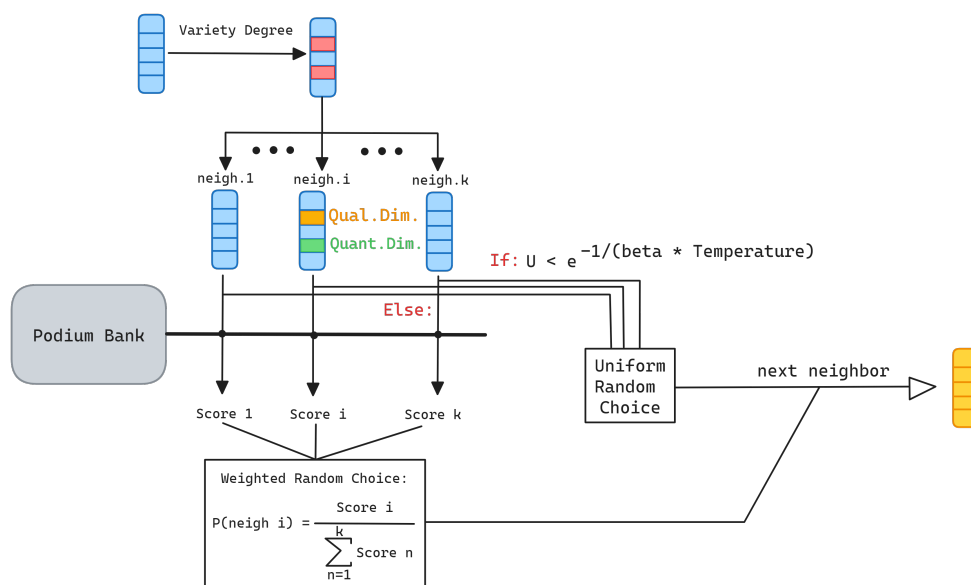


Figure 5: High level's learning component architecture.

Table 1: Comparison of different approaches with makespan objective.

Benchmark	PACO	HGA RMA	IG_RS LS	PSO <sub>vms</sub>	HHGA	L-SAGA
20 × 5	0.18	0.04	0.04	<b>0.03</b>	0.04	<b>0.03</b>
20 × 10	0.24	0.02	0.06	0.02	<b>0.00</b>	1.03
20 × 20	0.18	0.05	0.03	0.05	<b>0.00</b>	<b>0.00</b>
50 × 5	0.05	<b>0.00</b>	<b>0.00</b>	<b>0.00</b>	<b>0.00</b>	<b>0.00</b>
50 × 10	0.81	0.72	<b>0.56</b>	0.57	0.62	1.47
50 × 20	1.41	0.99	<b>0.94</b>	1.36	1.03	3.19
100 × 5	0.02	0.01	0.01	<b>0.00</b>	<b>0.00</b>	<b>0.00</b>
100 × 10	0.29	0.16	0.20	0.18	<b>0.08</b>	1.17
100 × 20	1.93	<b>1.30</b>	<b>1.30</b>	1.45	<b>1.30</b>	3.14
200 × 10	0.23	0.14	<b>0.12</b>	0.18	<b>0.12</b>	1.04
200 × 20	1.82	<b>1.26</b>	<b>1.26</b>	1.35	<b>1.26</b>	2.50
Average	0.65	0.42	0.41	0.47	<b>0.4</b>	1.23

of the best-known solution for the instance  $i$  as indicated in the benchmark data.

For each instance we execute L-SAGA from 1 to 3 times and report the minimum found RPD. We compared L-SAGA with several state-of-the-art approaches such as PACO (Ruiz et al., 2006), HGA RMA (Rajendran and Ziegler, 2004), IG\_RS LS (Ruiz and Stützle, 2007), PSO<sub>vms</sub> (Tasgetiren et al., 2007), and HHGA (Bacha et al., 2019). Table 1 presents the average Relative Percentage Deviation (ARPD) for the ten instances of the first 11 configurations with the best performing results for each benchmark highlighted in bold<sup>2</sup>.

By analyzing these observations, we can see that

<sup>2</sup>The last configuration of 500×20 was not yet tested at the time of this publication and is postponed for future research.

L-SAGA had no difficulty in handling increasing number of jobs as long as the number of machines remained small (in this case 5 machines); it indeed ranked first for the 20 × 5, 50 × 5 and 100 × 5 configurations. But as the number of machines increases, L-SAGA was systematically the least performant independently of the number of jobs, **at the intriguing exception of the 20 × 20 configuration where it also ranked first**. Further analysis, presented in Section 4.2 on the final HPSs generated by L-SAGA and Section 4.3 on the influence of the podium bank size, seem to provide plausible explanations of this behavior revealing potential avenues of improvement for further research. Still, immediate solutions for improvement can be thought of, such as allowing for the parallel execution of multiple low level GAs for a given HPS during exploration; as this would allow for better statistical estimation of an HPS ability to pro-

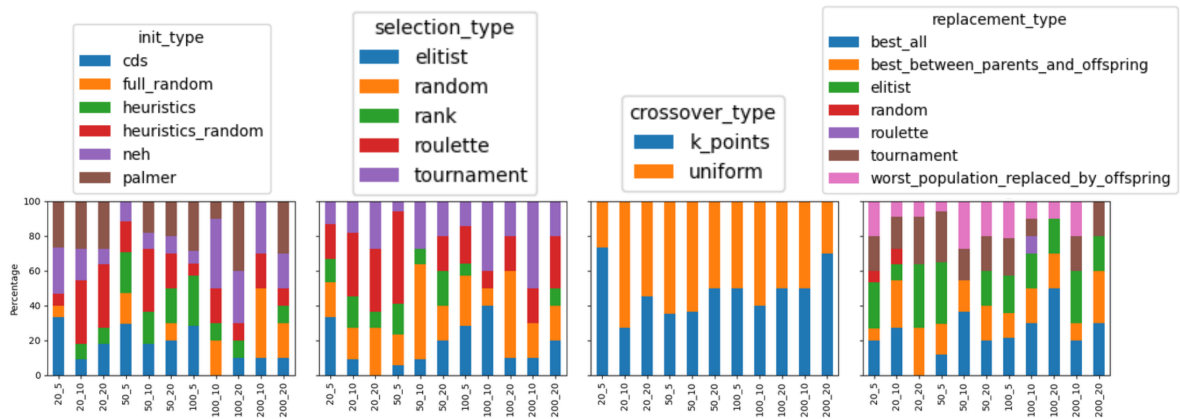


Figure 6: Percentage distribution of various initialization, selection, crossover, and replacement methods across different benchmark configurations.

duce good results (given that the low level GA rely on random processes) while maintaining minimum execution times. Also the current similarity measure used in the learning component could be improved by the online computation of correlation coefficients between the different hyperparameters of the HPS and the obtained makespans to give more weight to the more correlated dimensions when computing the similarity measure presented in equation 9.

#### 4.2 Analysis of L-SAGA’s Generated Low Level Hyperparameters

In order to gain some insight on the behavior of our method, we examined for each instance a selected set of hyperparameters in the final HPS generated by L-SAGA when producing the results in Section 4.1 and aggregated the results for analysis.

##### 4.2.1 Qualitative Hyperparameters

For each configuration, we count, over the instances, the number of times a particular value of a qualitative hyperparameter appeared in the final HPS. The percentages are graphically represented in Figure 6 for initialization, selection, crossover, and replacement methods.

1. **Initialization.** The 'NEH' method is widely used, proving effective for generating good initial solutions. 'Full-random' and 'heuristics-random' are also popular for providing diverse starting points, while the 'Palmer' method sees higher use in specific benchmarks (e.g., 100×20 and 200×20).
2. **Selection.** The 'random' selection method is most common, followed by 'roulette,' highlighting a balance between selection pressure and population

diversity. The 'elitist' method, though less common, plays a crucial role in retaining top solutions.

3. **Crossover.** 'Uniform' crossover dominates most benchmarks, with 'k-points' crossover being preferred in cases like 20×5 and 200×20.
4. **Replacement.** There’s a balanced use of replacement methods, with 'best-all' slightly more favored in benchmarks like 50×10 and 100×20, while 'roulette' replacement is used less, showing it to be less effective in this role.

From these observations it appears that L-SAGA does attempt to adapt to the different instances across different configurations. While this is indeed the desired behavior starting from the principle that no HPS configuration could fit all possible instances, the results presented in Section 4.1 show that L-SAGA still has difficulty in efficiently identifying the optimal combinations of qualitative hyperparameters. A possible explanation could be a local optimum stagnation caused by a too small beta value which would induce an early exploitation, or a too small podium bank size (see Section 4.3) which could cause a lack of variability during exploitation itself. Experimenting with different values of these two hyperparameters in a more extensive setup than manual fine tuning, along with analyzing the content of the whole podium bank at the end of the execution (instead of the final HPS only), could provide better insight and potentially a solution to this problem.

##### 4.2.2 Quantitative Hyperparameters

**1. Average Crossover Rate (ACR) and Average Mutation Rate (AMR):** for each configuration, we average over the instances the values of the CR and the MR that appears in the final HPSs. The analysis of Figure 7 reveals important adaptive mechanisms

found by the algorithm: The AMR consistently surpasses the ACR, emphasizing the importance of mutation for enhancing exploration and discovering new solution areas, and both rates fluctuate in tandem, indicating a coordinated approach that balances exploration (through mutation) and exploitation (through crossover).

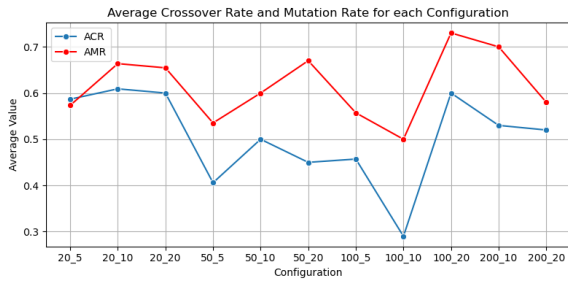


Figure 7: Behavior of the Average Crossover Rate (ACR) and Average Mutation Rate (AMR) across different configurations.

**2. Average Relative K-Point Crossover Value:** we also examined, among the instances on which L-SAGA execution led to a  $k$ -point crossover type in the final HPS, the evolution of the value of  $k$  across configurations. More specifically we compute for each instance the percentage of  $k$  relatively to the job sequence length i.e. the number of jobs (since  $1 < k < \text{number of jobs in the sequence} - 1$ ), and we then compute a per-configuration average. Observing this average in Figure 8 reveals a clear trend; as the number of jobs and machines increases, the average percentage of  $k$ -points decreases. This reduction suggests that L-SAGA tends towards less a fragmented approach to genetic material exchange for larger problem configurations. Analyzing this graph reveals an interesting fact: the only configuration with more than 5 machines (in this case  $20 \times 20$ ) in which L-SAGA actually ranked first displays a peak on the relative  $k$ -point crossover value where it reaches 25%. This could indicate that large values of  $k$ -point crossover are actually preferable. We suspect that one of the reasons that L-SAGA didn't evolve towards larger values of  $k$ -point crossover for other configurations is that the exploration window length that we set to 4 was not sufficient to allow for an effective exploration of this low level hyperparameter which worsened as the number of jobs increased. This suggests that experimenting with larger values of the exploration window, especially for this low level hyperparameter, should be a next step for further research.

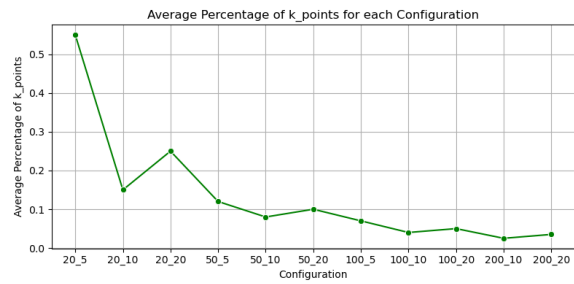


Figure 8: Average percentage of  $k$ -points in crossover operations across different problem configurations.

### 4.3 Effect of the Podium Bank Size

We studied the impact of the podium bank size on both the quality of the solutions and the execution time.

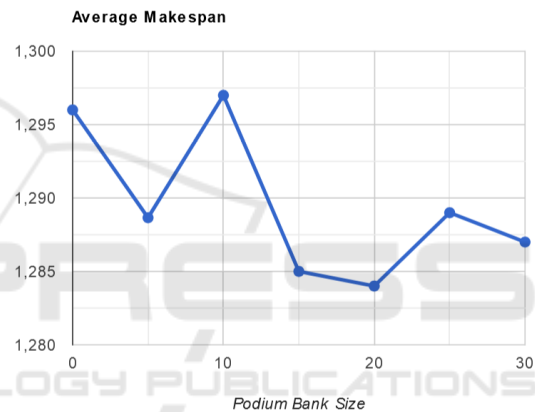


Figure 9: Study of the Average Makespan against Podium Bank Size.

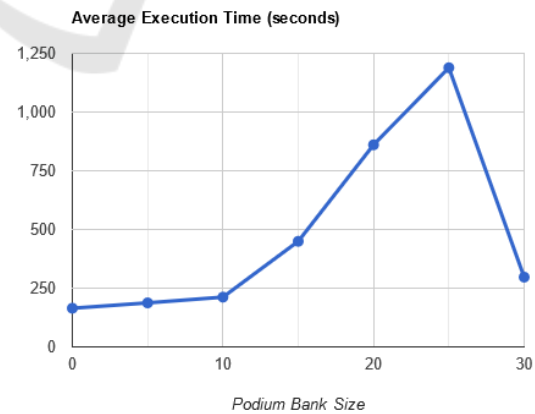


Figure 10: Study of the Average Execution Time against Podium Bank Size.

The experimentation was done by performing three L-SAGA executions for each podium bank size on the first instance of the 20 jobs by 5 ma-



chines configuration, and then averaging the obtained makespans and execution times. From Figure 9 it appears that an increase in the podium bank size tends to provide better results as the average makespan decreases, but at the price of an increase in the execution time as shown in Figure 10 which can be explained by the necessity to perform more calculations for computing the podium scores during search. The execution time could be reduced by implementing parallelization during the scores computation as these don't depend on each other. Also, the sudden drop in execution time noticed at the podium bank size of 30 is explained by the early stopping of the execution due to a stagnation in the makespan, which when correlated with the relatively low average makespan value at this podium bank size could support the idea that the algorithm finds better optimum more quickly.

## 5 CONCLUSION

In this work we have presented L-SAGA, a generative hyper-heuristic conceived for finding near optimal solutions to the PFSP. L-SAGA, by combining a simulated annealing backed by a specially designed learning component for the high level, and a PFSP adapted version of the genetic algorithm for the low level, managed to give competitive results on some specific small to medium size benchmarks. Still, as of this paper's results, the execution of L-SAGA shows a lack of performance on larger and more complex instances when compared to the state of the art. The results obtained by analyzing L-SAGA behavior suggest that the most important next step in improving its performance would be to perform a more extensive high level hyperparameters fine-tuning that would leverage the insights found in this first study, such as those concerning the effect of the podium bank size, the relative k-point crossover value, or the beta hyperparameter of the learning component. Coupled with an automated setup, and along some additional implementations such as multiple GA executions per HPS and a refined similarity measure, we expect this fine-tuning to allow L-SAGA to significantly improve on more complex instances. This shall be our next step.

## REFERENCES

- Alekseeva, E., Mezma, M., Tuyttens, D., and Melab, N. (2017). Parallel multi-core hyper-heuristic grasp to solve permutation flow-shop problem. *Concurrency and Computation: Practice and Experience*, 29(9):e3835.
- Bacha, S. Z. A., Belahdji, M. W., Benatchba, K., and Tayeb, F. B.-S. (2019). A new hyper-heuristic to generate effective instance ga for the permutation flow shop problem. *Procedia Computer Science*, 159:1365–1374.
- Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Woodward, J. R. (2010). A classification of hyper-heuristic approaches. *Handbook of metaheuristics*, pages 449–468.
- Garey, M. R., Johnson, D. S., and Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129.
- Garza-Santisteban, F., Amaya, I., Cruz-Duarte, J., Ortiz-Bayliss, J. C., Özcan, E., and Terashima-Marín, H. (2020). Exploring problem state transformations to enhance hyper-heuristics for the job-shop scheduling problem. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.
- Garza-Santisteban, F., Sánchez-Pámanes, R., Puente-Rodríguez, L. A., Amaya, I., Ortiz-Bayliss, J. C., Conant-Pablos, S., and Terashima-Marín, H. (2019). A simulated annealing hyper-heuristic for job shop scheduling problems. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 57–64. IEEE.
- Miller, L. W., Conway, R. W., and Maxwell, W. L. (1967). *Theory of Scheduling*.
- Pinedo, M. L. (2012). *Scheduling: Theory, Algorithms, and Systems*. Springer Science & Business Media.
- Rajendran, C. and Ziegler, H. (2004). Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. *European Journal of Operational Research*, 155(2):426–438.
- Ruiz, R., Maroto, C., and Alcaraz, J. (2006). Two new robust genetic algorithms for the flowshop scheduling problem. *Omega*, 34(5):461–476.
- Ruiz, R. and Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049.
- Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285.
- Tasgetiren, M. F., Liang, Y.-C., Sevkli, M., and Gencyilmaz, G. (2007). A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. *European Journal of Operational Research*, 177(3):1930–1947.