QLHH : A Hyper-heuristic with QLearning for the Flowshop Permutation Problem

Mohamed Islem KARABERNOU, Adel BOULAOUAD
Sarah ABCHICHE, Mustapha Ayoub BELOUADAH
Akli YALAOUI ,Mohamed DJILANI

June 24, 2022

Abstract

Hard combinatorial optimization are problems that involves huge search space and have been studied extensively by operational research. This paper propose a hyper-heuristic based on reinforcment learning to decide the next heuristic to optimize for the Flowshop Permutation Problem

Keywords: FSP, hyper-heuristic, optimization

1 Introduction

Many methods have been developed to solve FSP problem including exact methods like Branch and Bound. Such methods exhaustively explores the search space and therefore guaranteeing the optimal solution. However trying every possible solution is tramendously time consuming and the time complexity grows exponentionally with the size of the instance.
To overcome the huge time complexity , another type of approached methods have been developed , this type introduces a trade-off between the quality of the solution and the time complexity. The first family of approached methodes are the Heuristics, a type of algorithms that are designed to solve a problem in a faster and more efficient fashion than exact methods by sacrificing optimality, accuracy, precision, or completeness for speed, they can deliver good quality solution in a reasonable amount of time, they are also problem specific which means that a heuristic is specificaly designed to solve a particular problem and cannot be generalized to others. For the flowshop permutation problem (which we will discuss in the next section) there are plenty of heuristics like NEH , CDS , Palmer and many more.
The second family of approached methods are the Metaheuristics, a metaheuristic is

an iterative process that guides a heuristic by intelligently combining different concepts to explore and exploit the search space , it represents a high-level procedure that aims to find sufficiently good solutions with a reduced time complexity. Metaheuristics are not problem specific and can be generalized to a variety of problems, and many of them exists in the literature like Genetic Algorithms , Tabu Search , Simulated Annealing and many more. However , metaheuristics required fine-tunning of parameters to get better results, this process was repeated for every instance of a problem which was a bit overwhelming and time consuming.
For that a new type of approached methods emerged which is the Hyper-heuristics, Hyper-heuristics are much more general with a high level of abstraction. They are an automated search methodology that combines simple heuristics or elements of heuristics to solve efficiently an instance or a class of instances of a problem.
Our goal in this article is to propose a new hyper-heuristic based on Q-learning to solve the flowshop permutation problem, we start first by defining the problem we are trying to solve , then we dive deep into the details of our method , after that we show the results of experiments and tests and compare our approach to existing state of the art heuristics and meta-heuristics.

2 Problem Definition

2.1 Flowshop Permutation

Flowshop permutation is a combinatorial optimization problem that consists of finding the optimal sequence of n jobs executing sequentially on m machines. The goal is to minimize the **makespan** which is the total time required to execute all jobs on all machines.
This problem follows a set of constraints :

- All jobs are available at t=0
- All machines are not interruptable
- Every machine can process at most one job at a given instant t
- No pre-emption is allowed
- All jobs have the same execution order on all machines

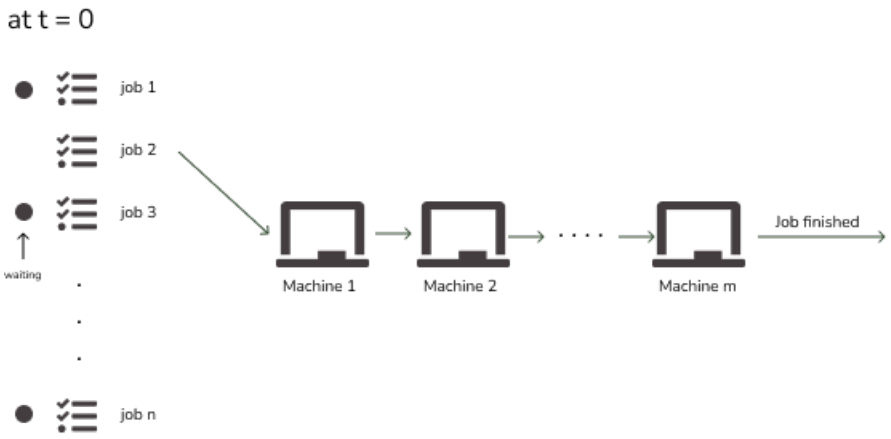


Figure 1: Illustration of the flow shop problem

2.2 Objective Function

The makespan (noted C_{max}) is the objective function to minimize in the flowshop permutation problem and it represents the total amount of time required to execute all jobs on all machines.
C_{max} can be calculated using the following formulas :

$$\mathrm{C}_{1,j} = C_{1,j-1} + d_{j,\pi^{-1}(1)} \quad (1)$$

$$\mathrm{C}_{i,1} = \sum_{j=1}^{i} d_{1,\pi^{-1}(j)} = C_{i-1,1} + d_{1,\pi^{-1}(i)} \quad (2)$$

$$\mathrm{C}_{i,j} = max(C_{i-1,j}, C_{i,j-1}) + d_{j,\pi^{-1}(i)} \quad (3)$$

Where :

- $C_{i,j}$: is the total execution time until the i^{th} job on the j^{th} machine
- π(i) is a permutation function that takes as an argument the identifier or the number of the job and returns its position in the sequence

- $d_{j,i}$ is the execution time of the job i on the machine j

Given n jobs and m machines, $C_{n,m}$ will be the total makespan that should be minimized.
By observing the recursive nature of the formulas , it is highly recommended to use dynamic programming when calculating C_{max} in order to accelerate the computational time.

3 Proposed Method

Our approach consists of a selection perturbative hyper-heuristic with an offline learning. It first learns the best sequence of execution of the low level heuristics (LLH) using Q-learning, after that it applies that sequence to the initial solution in attempt to find the optimal solution.

In this section, we present the global architecture of the hyper-heuristic, then we show its details, finally we discuss the learning phase.

3.1 Global architecture

Our solution aims to select the best heuristic to apply to the current solution at each iteration. This is made possible due to the agent who selects the appropriate heuristic (details provided in the next section).

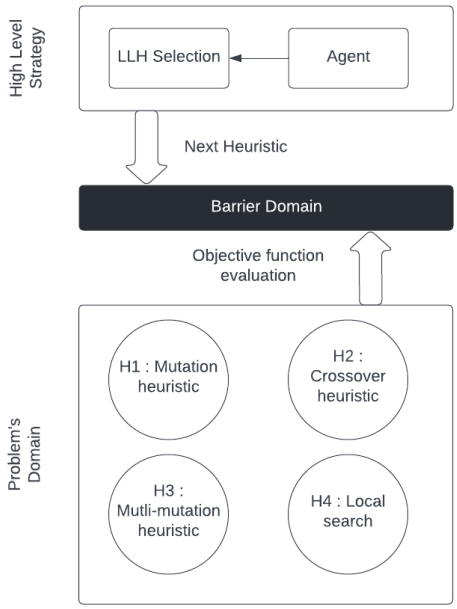


Figure 2: Global Architecture of the proposed hyper-heuristic

Our approach is flexible, we can use as many LLHs we need in the problem's domain. For now, we implemented four heuristics, each of them is a local search with slight variations :

- **h1 :** It applies a mutation to the current solution by exchanging the order of two random jobs.
- **h2 :** It combines the current solution with a randomly generated solution by applying a crossover between the two of them
- **h3 :** It similar to H1, the mutation is executed n times.
- **h4 :** We try to improve the current solution by exploring all its neighbors and pick the best one among them all.

H1 , H2 and H3 are followed with a local search in order to improve the obtained solution. The local search method used is the Rajendran and Ziegler (1997) method (denoted as RZ) as described below :

Algorithm 1 RZ Local search

Input : current Solution
Output : improved Solution
$S \leftarrow currentSolution$
while $j < numberOfJobs$ **do**
    $S' \leftarrow currentSolution$
    remove job $jfromS'$
    Test job j in all the possible positions of except for its original one.
    Insert job j in at the position resulting the lowest total flow-time.
    if $f(S') \leq f(currentSolution)$ **then**
        $currentSolution \leftarrow S'$
    end if
end while

3.2 Flow chart of the hyperheuristic

We start with an initial solution (in our case, we use Palmer's heuristic to generate it, because it is so fast and generate an acceptable solution). Next we train the agent on the given problem instance, so it will be able to determine the best sequence of heuristics selection in that particular problem instance. Then, at each iteration we generate a uniform number between 0 and 1, and if it is less than or equal to $P0$ (probability of random heuristic selection) then we select a random heuristic to apply to the current solution, else we select the best one according to current state. Finally, we compare the obtained solution with the best one found so far and we update the latter. We continue this process until a stop condition is met(such as

maximum number of iteration is reached or maximum number of iterations without improvements is reached). the flow chart [figure 2] summarizes the whole process, also the pseudo-code is detailed [Algorithm 2].

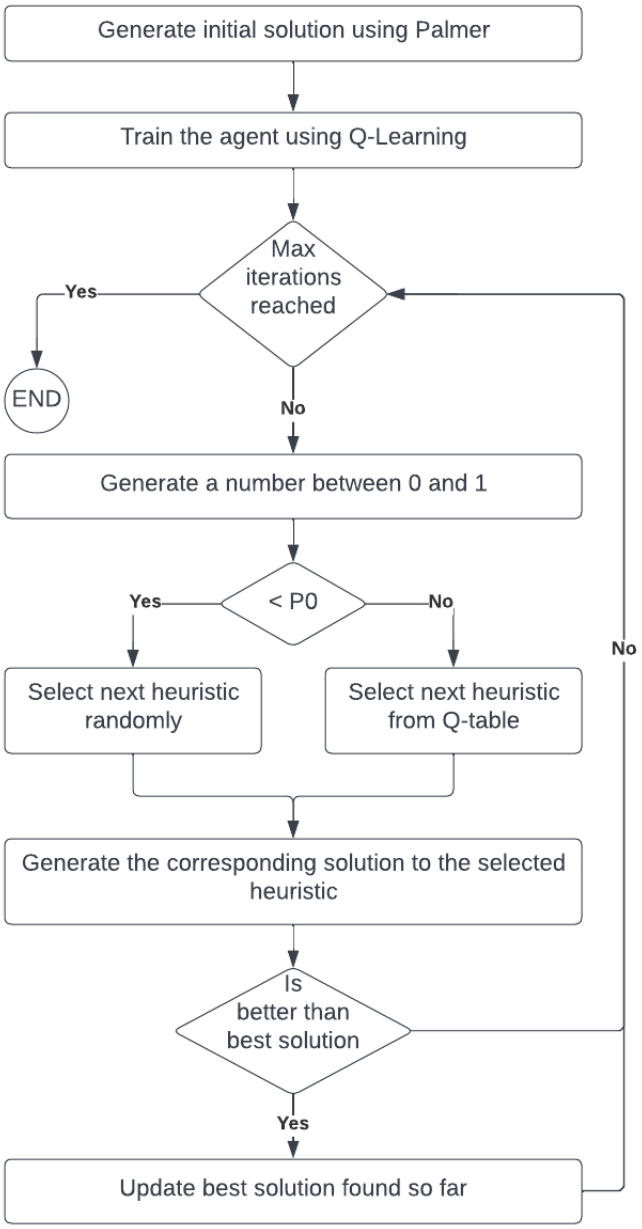


Figure 3: Flow chart of the proposed hyper-heuristic

Algorithm 2 Selection perturbative hyper-heuristic with an offline learning

Input : instance of the problem, initial solution, Max-Iterations, P0
Output : Best jobs order, make-span
Train the agent in order to determine the best sequence of heuristics selection.
Initialize initial solution using Palmer's heuristic
while $i < Max - Iterations$ **do**
    Generate random number p between 0 and 1
    if $p \leq P0$ **then**
        Chose next LLH randomly
    else
        Chose next LLH from the Q-table : The heuristic to apply is the one that has a maximum value in the row of the current state in the Q-Table.
    end if
    Apply the selected LLH to current solution
    if Current solution is better than best solution so far **then**
        Save current solution as the best solution.
    end if
end while

3.3 Learning phase

In this section, we discuss in details the learning process of the agent. So we have a problem instance that we want to learn the best heuristic to apply at each iteration. For doing that, we build an agent who will interact with that environment. At each step t, it is at a certain state s_t and chooses an action a_t to perform. The environment runs the selected action and returns a reward to the agent, the higher the reward the better is the action, so an episode can be represented as a sequence of state-action-reward.

Now we need to define , what are the states and actions, and how to calculate the reward. Let's suppose we are at the iteration i, and we applied heuristic h at the previous iteration $i-1$, we want to know which is the best heuristic to apply at the current iteration. With that said, we can define the set of actions, states and the reward function as follow :

- **States :** A state is the last heuristic selected. Thus, the set of states is the set of available low level heuristics.
- **Actions :** An action is to select one of available heuristics. Thus,the set of actions is the set of available low level heuristics.
- **A reward :** Let f' the objective function value of the previous iteration's solution S, and f the objective function value after executing the selected heuristic on S. The the reward r will be calculated as follows :

$$r(f', f) = \begin{cases} 1, & \text{if f' - f} \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

After defining our elements, the q-learning algorithm starts by initializing a matrix *Q(s,a)* of s rows and a lines where s is the number of possible states and a is the number of possible actions. Note that an element of the matrix represents the reward gained after choosing an action while being in a specific state. Then, the algorithm chooses an action to perform according to a ϵ-greedy rule which is a criterion of exploration-exploitation, that consists of choosing the action with the highest value of Q with probability $1 - \epsilon$, and choosing a random action with probability ϵ. Lastly, the q-value is updated with the following expression :

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a)]$$

where s_t is the current state, a_t is the action performed in the state s_t, r_t is the received reward for executing a_t in s_t, and s_{t+1} is the new state; γ is a discount factor ($0 \leq \gamma < 1$), while α ($0 < \alpha < 1$) is the learning rate.

Algorithm 3 Q-Learning

```
Initialize Q(s,a)
for each episode do
    Initialize s
    for each step of episode do
        Choose a according to the ϵ-greedy rule
        Perform action a, observe r, s'
        Update Q(s,a)
        s ← s'
    end for
end for
```

4 Experimental Results

To test our proposed method we used the taillard benchmarks for the flow shop sequencing problem, since it allows us to compare our results to other proposed methods in the litterature. The used instances are presented in the following table:

Instance	ta001	ta002	ta003	ta004	ta006	ta007	ta008
Jobs	20	20	20	20	20	20	20
Machines	5	5	5	5	5	5	5
Best makespan	1278	1359	1081	1293	1195	1234	1206
Instance	ta009	ta010	ta011	ta012	ta013	ta014	ta015
Jobs	20	20	20	20	20	20	20
Machines	5	5	10	10	10	10	10
Best makespan	1230	1108	1582	1659	1496	1377	1419

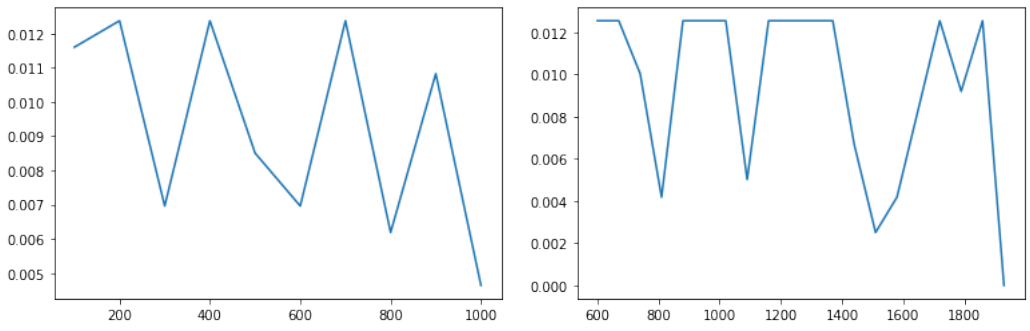
Instance	ta016	ta017	ta018	ta019	ta020	ta021	ta022
Jobs	20	20	20	20	20	20	20
Machines	10	10	10	10	10	20	20
Best makespan	1397	1484	1538	1593	1591	2297	2099
Instance	ta023	ta024	ta025	ta026	ta027	ta028	ta029
Jobs	20	20	20	20	20	20	20
Machines	20	20	20	20	20	20	20
Best makespan	2326	2223	2291	2226	2273	2200	2237
Instance	ta030	ta031	ta041	ta051	ta061	ta071	ta081
Jobs	20	50	50	50	100	100	100
Machines	20	5	10	20	5	10	20
Best makespan	2178	2724	3025	3875	5493	5770	6286

After presenting the used instances, we will present some experimental results on the number of episodes used in the training and the number of iterations after the training, to study the effect of the learning phase on the finale results and on the execution time. To judge the qualiy of the solution returned, we calculated each time the gap between it's makespan and the optimale solution's makespan from the taillard's instances :

$$GAP = \frac{ObtainedMakespan - BestMakespan}{BestMakespan} \tag{4}$$

4.1 Number Of Episodes Analysis

In this section, we will present the results of the experiments made to test the effect of the number of episodes of the Q-learning algorithm on the final results. To do so, we fixed a number of iterations of 300 iterations and varied the number of episodes. The following graphes represents the results of the experiment on three instances (ta004, ta006 and ta022):



Number of episodes

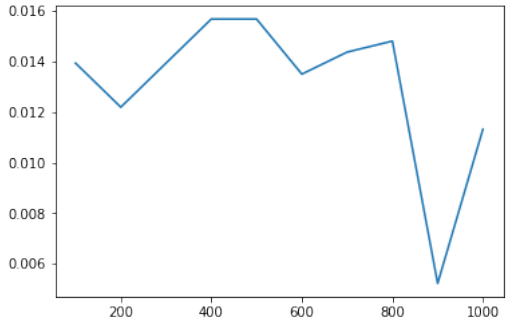


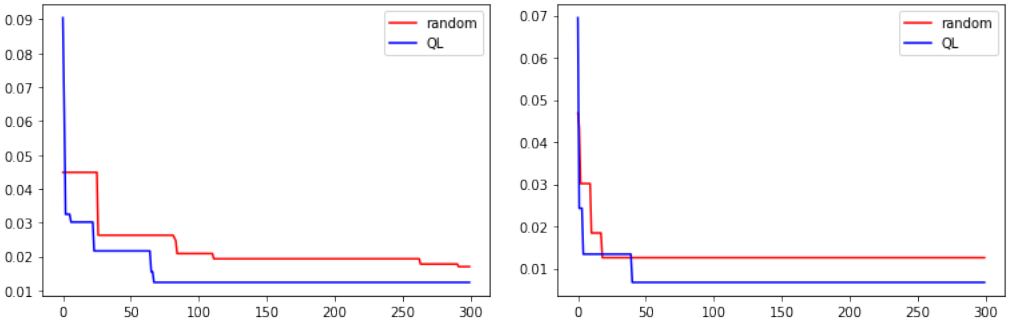
Figure 4 - Evolution of the obtained GAP for different numbers of episodes

4.1.1 Discussion

The effect of the augmentation of the number of episodes differ from an instance to another. As we can see for the instance ta022, better results were found with bigger number of episodes. But for the ta004 and ta006 instances, we noticed that for a big range of the number of episodes, the results were a little bit random, which indicates the non-convergence of the Q-Learning algorithm yet. For example, for the ta006 instance, we had to go until 2000 episodes to understand the behaviour of the alorithm. But the common result is that, for the best results, we should pass a bigger number of episodes.

4.2 Number Of iterations Analysis

In this section, we will test the effect of the learning phase on the evolution of the results at each moment of the execution of the algorithm. To do so, we compared the method with the reinforcement learning phase with another one where we choose randomly the next heuristic to apply. The following graphs represents the evolution of the best solution found at each iteration of the two methods for 300 iterations applied to the same three instances (ta004, ta006 and ta022):



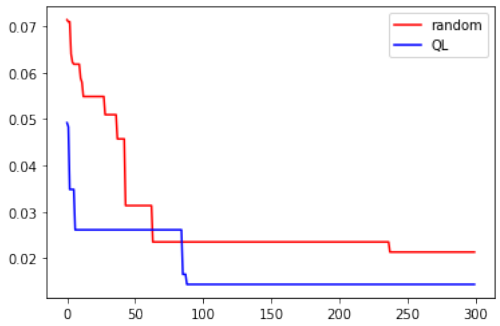


Figure 5 - Comparaison of the evolution of the best obtained GAP at each iteration between random and QL hyper-heuristic

4.2.1 Discussion

As was expected, the effect of the learning phase is obvious since we could get to better solutions in a smaller number of iterations compared to the random selection method. In the following section, we will see more the effectivness of our proposed method compared to the random selection one and to other used approached.

5 Comparative Analysis

In this part, we will test both methods (RHH and QLHH) for different instances of the problem and compare it to different solutions we implemented : Heuristics (Palmer, DCS), metaheuristics (Tabu search, NEH).
In order to get clear and generalized results, we will be running our tests on two types of instances : small instances with 20 jobs (including 30 instances of 5,10, 20 machines) and big instances with 100 jobs (including 10 instances 5,10,20 machines). The result is the mean of 3 executions for each type of instance.
The following figures present the evolution of the makespan and execution time for each type of solution.

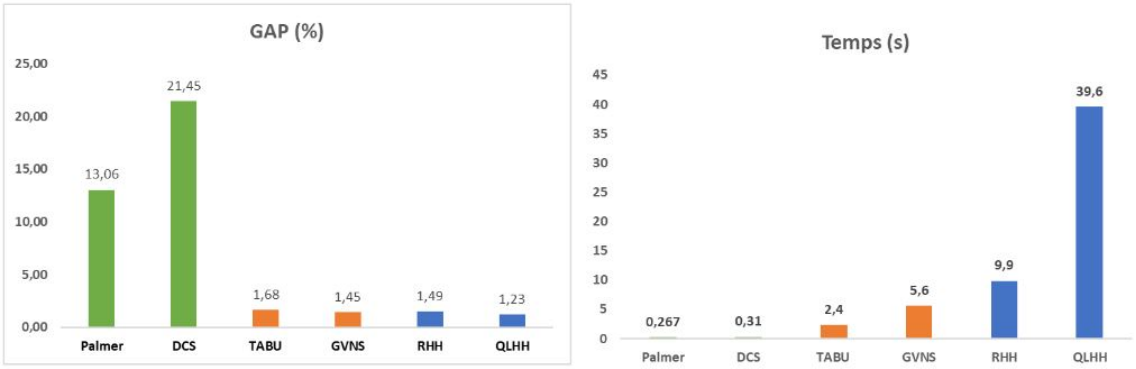


Figure 6 - Evolution of the makespan and execution time for small instances

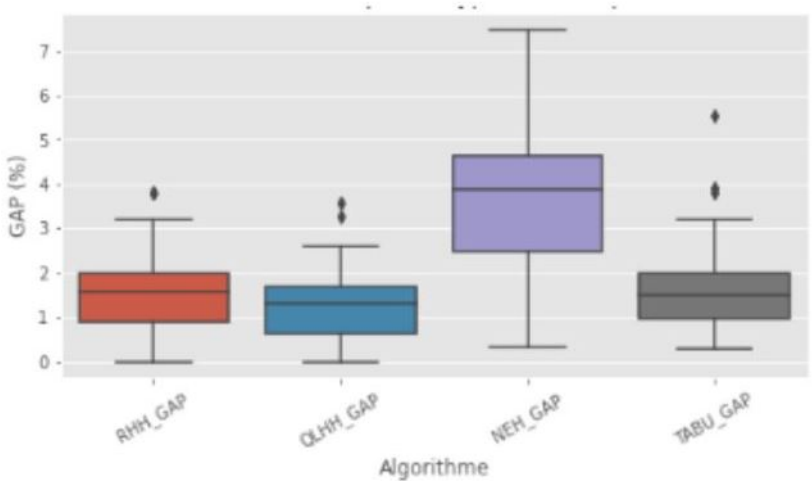


Figure 7 - GAP comparaison between solution for small instances

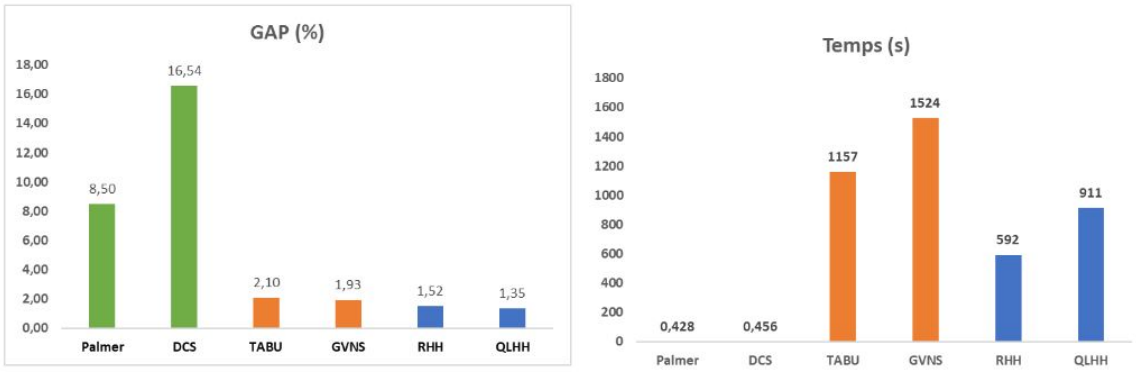


Figure 8 - Evolution of the makespan and execution time for big instances

We observe that the results obtained by the hyperheuristics are better than other methods in terms of GAP score in small and big instances. Some instances can achieve the optimal solution (0% Gap). Heuristics achieve weak results but in a very short amount of time, we can use them as initial solutions for advanced optimization. Metaheuristics offer a good compromise between time and GAP for small instances. For bigger instances we were able to optimize the time of the hyperheuristics by stopping the algorithm after a number of steps of non amelioration. This maintained good results while saving some time.

6 Conclusion

In this work, we presented our approach using the hyper-heuristic to solve the permutation flow shop problem.the hyper-heuristic have been implemented in order to have a more general method with a higher level of abstraction, and that does not depend on the parameters or the initial solution feeded. Random hyper heuristic RHH and Qlearning hyper heuristic QLHH have been used. RHH selects randomly a low level heuristic, QLHH integrates machine learning through Q learning as a high-level strategy that will manipulate low-level heuristics.Our method produces good quality solutions in terms of Gap, The Gap is further reduced compared to metahehristics by using simple and easy heuristic components,these results are obtained in a similar time as that of metaheuristics.

References

[1] M. Ben-Daya and M. Al-Fawzan. A tabu search approach for the flow shop scheduling problem. *European Journal of Operational Research*, 109(1):88–95, 1998.

[2] Andreas Fink and Stefan Voß. Solving the continuous flow-shop scheduling problem by metaheuristics. *European Journal of Operational Research*, 151(2):400–414, 2003. Meta-heuristics in combinatorial optimization.

[3] Fernando Garza-Santisteban, Roberto Sánchez-Pámanes, Luis Antonio Puente-Rodríguez, Ivan Amaya, José Carlos Ortiz-Bayliss, Santiago Conant-Pablos, and Hugo Terashima-Marín. A simulated annealing hyper-heuristic for job shop scheduling problems. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 57–64, 2019.

[4] Eva Vallada, Rubén Ruiz, and Jose M. Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3):666–677, 2015.

[5] Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285, 1993.

[6] Edmund K Burke, Graham Kendall, Mustafa Mısır, and Ender Özcan. Monte carlo hyper-heuristics for examination timetabling. *Annals of Operations Research*, 196(1):73–90, 2012.

[7] Abdellah Salhi and José Antonio Vázquez Rodríguez. Tailoring hyper-heuristics to specific instances of a scheduling problem using affinity and competence functions. *Memetic Computing*, 6(2):77–84, 2014.

[8] İlker Gölcük and Fehmi Burcin Ozsoydan. Q-learning and hyper-heuristic based algorithm recommendation for changing environments. *Engineering Applications of Artificial Intelligence*, 102:104284, 2021.

[9] Shin Siang Choong, Li-Pei Wong, and Chee Peng Lim. Automatic design of hyper-heuristic based on reinforcement learning. *Information Sciences*, 436:89–107, 2018.

[10] Xingye Dong, Houkuan Huang, and Ping Chen. An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion. *Computers & Operations Research*, 36(5):1664–1669, 2009.