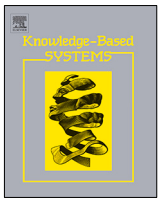
Knowledge-Based Systems 264 (2023) 110309

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys



# Toward automated algorithm configuration for distributed hybrid flow shop scheduling with multiprocessor tasks

Hadi Gholami *, Hongyang Sun

*University of Kansas, KS, USA*

ARTICLE INFO

*Article history:*
Received 2 November 2022
Received in revised form 1 January 2023
Accepted 14 January 2023
Available online 20 January 2023

Dataset link: http://swlin.cgu.edu.tw/data/DHFSP.zip

*Keywords:*
Automated algorithm design
Conditional markov chain search
Distributed hybrid flow shop scheduling
Multiprocessor tasks
Q-learning algorithm

ABSTRACT

Due to the large volume of requests and the need to speed up the provision of services, production companies are migrating from a single service center to distributed centers. To support this migration, it is necessary to make intelligence decisions that benefit from automatic design of search algorithms. Considering these, this paper addresses the distributed hybrid flow shop scheduling problem with multiprocessor tasks (DHFSP-MT) as an extension of the hybrid flow shop scheduling problem with multiprocessor tasks (HFSP-MT) to minimize the maximum completion time among distributed factories. To provide effective decision support, we apply a novel framework called conditional markov chain search (CMCS) to automate the generation of heuristics, which is presented for the first time in the distributed shop scheduling problem to the best of our knowledge. We express the HFSP-MT as a markov decision process (MDP) and solve it through a hybrid Q-learning-local search algorithm. By using the characteristics of the problem under study, we introduce two new concepts, weight and impact, which are used to develop an initial construction algorithm and two local search methods. To balance jobs between factories at runtime, we propose a load balancing method, which transfers selected jobs from certain source factories to destination factories. We compare the proposed CMCS with two state-of-the-art metaheuristic algorithms from the literature using publicly available benchmark instances. The computational results show that the proposed CMCS provides better performance than that of the existing algorithms on solving the considered DHFSP-MT.



## 1. Introduction

With a highly globalized economy, a large number of manufacturing companies nowadays are utilizing the production capacity in multi-location lines to meet the complex needs of customers and to deal with uncertain markets. Besides, distribution of computational jobs across multiple, autonomous processing units, such as heterogeneous distributed computing systems [1] and high-performance computing data centers [2], is expected to overcome the limitations of traditional centralized processing. In various scientific domains (e.g., workflow computation and particle physics), large jobs are often split into smaller ones to increase efficiency and reduce costs, and each small job is executed in a separate processing unit.

Hybrid Flow Shop Problem (HFSP) is a well-known shop scheduling problem from both theoretical and practical points of view. With the multi-location concept, distributed processing of computational jobs in HFSP can be formulated as a Distributed Hybrid Flow Shop Problem (DHFSP) with practical constraints, such as Multiprocessor Tasks (MT). DHFSP with MT (or DHFSP-MT) is a distributed scheduling problem, extending the HFSP with MT (or HFSP-MT), which was first studied by Ying and Lin [3]. In DHFSP-MT, multiple factories are located in different locations and processing of jobs within each factory is considered as an HFSP-MT. In each HFSP-MT, jobs are processed in a series of stages. Each stage may have more than one machine that can work in parallel, and jobs may also be parallel in some stages, requiring more than one machine. The extensive applications of using DHFSP-MT in different environments of computer systems such as real-time machine-vision systems [4] or production systems such as semiconductor manufacturing with more than one wafer fab [3] have led to a lot of recent research efforts to develop suitable solutions for solving the problem. Fig. 1 depicts the schematic diagram of a DHFSP-MT. As can be seen, two phases are considered for scheduling. In Phase I, a set of jobs is assigned to the dispatcher in a list, and the dispatcher distributes the jobs among the factories based on a strategy. In Phase II, each factory has a queue of jobs to be processed as HFSP-MT. In this phase, the dispatcher can migrate jobs between factories to optimize the objective function. In this paper, we aim at designing, for DHFSP-MT, an effective scheduler that exploits the potential of multiple

* Corresponding author.
*E-mail addresses:* gholamihd@gmail.com (H. Gholami), hongyang.sun@ku.edu (H. Sun).

https://doi.org/10.1016/j.knosys.2023.110309
0950-7051/

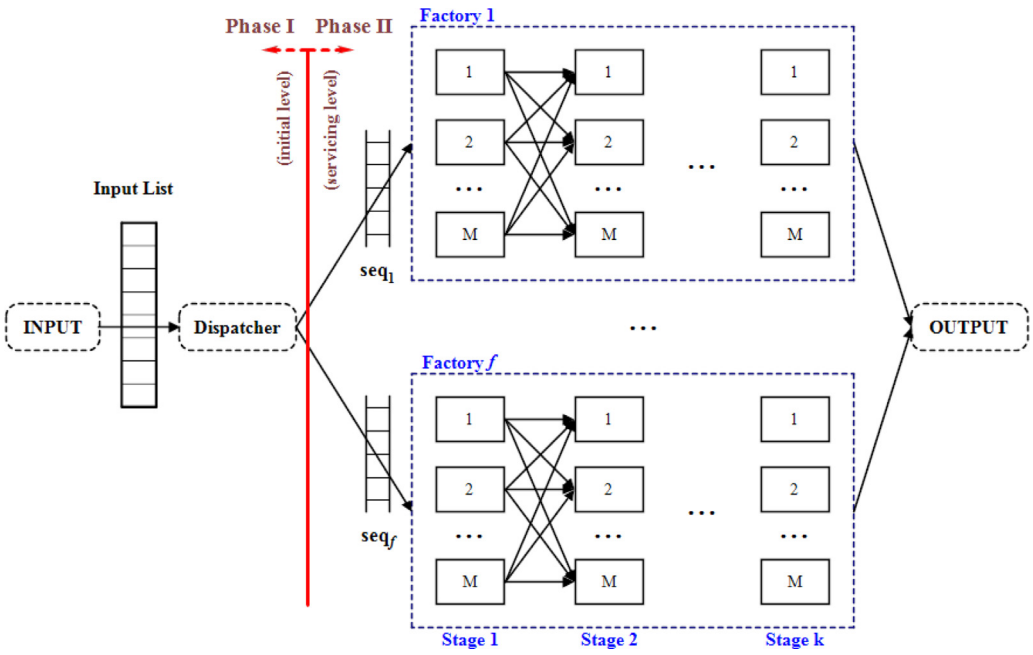



**Fig. 1.** A Distributed Hybrid Flow Shop Problem (DHFSP) with Multiprocessor Tasks (MT).

production lines to bring benefits such as reduction of the overall production period (or job completion time).

For DHFSP-MT, however, in addition to job sequencing in each traditional single factory, there is a need for Load Balancing (LBA) [5] among multiple factories in the distributed environment. LBA refers to the distribution of workload in the factories in order to optimize the efficiency of the resources of each factory and also to speed up the response time to the customer. It avoids situations where some factories are processing a significant amount of workload and others are processing a small amount of workload. LBA is critical to balance the workloads as part of the scheduling process. In the problem under study, we consider LBA for both Phase I and Phase II of Fig. 1, which is done by the dispatcher.

### 1.1. Objectives of this paper

Recently, algorithms based on interaction with the environment have been presented to improve the behavior of the algorithms in complex combinatorial optimization problems. In particular, reinforcement learning algorithms have been developed to solve some distributed shop scheduling problems [6–10]. Although these efforts were to make algorithms more intelligent and reduce human intervention in decision making, further steps are needed to inject artificial intelligence into the distributed shop scheduling problems. The development of automated algorithm design could be the next step in this research field. For example, the presence of a human expert for tuning the developed algorithms is one of the major concerns from researchers, and getting rid of such manual work and entrusting it to the computer will help to explore a larger scope of algorithms [11]. Therefore, considering the learning of algorithmic components to the realization of automated algorithm design in the distributed shop scheduling problem will offer an attractive step toward jumping from low-level algorithms to high-level ones.

Furthermore, as it is clear from the literature, there is no rule for the realization of LBA in Phase II in the studied problems, except for the random-based rules. Therefore, developing more intelligent solutions for Phase II of LBA during runtime is considered as a research gap yet to be filled. Motivated by providing methods to solve the above issues, the present paper studies DHFSP-MT to minimize the maximum job completion time or makespan.

### 1.2. Our contributions

The main contributions of this paper are listed below:

- To the best of our knowledge, this is the first attempt to inject automated search algorithm into a distributed shop scheduling problem. For this purpose, we apply the Conditional Markov Chain Search (CMCS) framework.
- We introduce a new concept, called Weight, by using the characteristics of the problem under study to cover the LBA in Phase II and tackle random-based selecting and migrating jobs between factories during runtime. Moreover, we define another new concept, called impact, so that we can prioritize jobs by determining the impact of jobs at different stages. Determining the impact of jobs is variable and can be changed by using a coefficient.
- We formulate the HFSP-MT as a Markov Decision Process (MDP), and introduce a hybrid Q-learning-local search approach for constructing intra-factory solutions.
- We develop a heuristic method to get an initial solution and introduce two new local search algorithms to produce improved solutions.
- We conduct a comprehensive set of experiments on well-known instances from the literature. Comparing the obtained results with those of two state-of-the-art algorithms shows that our method improves the performance than that of the compared algorithms.

### 1.3. Structure of the paper

The remainder of the paper is organized as follows. Section 2 reviews recent literatures about HFSP-MT and DHFSP-MT. Section 3 establishes the mathematical model for DHFSP-MT, defines the weight and impact concepts, and briefly introduces the CMCS framework. Section 4 describes the components of the framework in detail. Section 5 presents the results of the computational experiments. Finally, Section 6 concludes the paper and discusses some future works.

## 2. Literature review

Many researchers have studied HFSP-MT in the last two decades. Almost all of them offered solutions to minimize the makespan, which is considered an effective metric for customer satisfaction. The tabu search algorithm (TS) has been proposed to solve the problem [12] along with several heuristics and metaheuristics such as particle swarm optimization (PSO) [13], iterated greedy (IG) heuristic [14], parallel greedy algorithm (PGA) [15], simulated annealing (SA) algorithm [16], hybrid immune algorithm (HIA) [17], immune algorithm (IA) [18], discrete firefly algorithm (DFA) [19], ant colony optimization (ACO) [20], and memetic algorithm (MA) [21]. These algorithms, by making changes in the parameters or operators of the algorithm under study or by proposing rules (such as dispatching rules) or random-based methods, were able to introduce job sequencing and scheduling solutions in the single-factory setting.

To solve the problem of scheduling in DHFSP and Distributed Flow shop Problem (DFSP), ordering jobs according to one or more rules and dispatching them among factories is one of the activities that helps to have an efficient LBA in Phase I. In the literature, a three-step method has been presented in which a new assignment rule is used in [22]. However, in the presented hybrid enhanced discrete fruit optimization algorithm (HEDFOA), a random-based process was proposed for Phase II LBA. Sharing jobs between factories based on efficient rules was emphasized in [23] as the initial solution. The presented method in the study solved the problem under IG's framework. They used two methods to LBA in Phase II, called Insertion_between and Swap_between, both of which balanced the processing capacity of factories based on the random concept. In the same way, a multi-local search-based general variable neighborhood search (MGVNS) in [24], a cooperative coevolution algorithm by combining the estimation of distribution algorithm (EDA) and IG in [25], a population-based iterated greedy algorithm (PBIG) in [26,27], an evolution strategy algorithm in [28], a water wave optimization algorithm with problem-specific knowledge (KWWO) in [29], a cooperative memetic algorithm with feedback (CMAF) in [30], and an efficient iterated greedy (EIG) in [31]. Moreover, Li et al. [32] proposed a discrete artificial bee colony algorithm (DABC), Shao et al. [33] developed a heuristic-based approach, Khare and Agrawal [34] used a combination of harris hawks optimization (HHO) and IG, Cai et al. [35] proposed a dynamic shuffled frog-leaping algorithm (DSFLA), and Shao et al. [36] presented a multi-objective evolutionary algorithm based on multiple neighborhoods local search (MOEA-LS). It is worth mentioning that LBA can play an important role in the quality of solutions and, of course, reducing the computation time. None of the above works could introduce any methods other than randomly for the realization of efficient LBA in Phase II in terms of selecting a job from the source factory and placing the job in the destination factory. In other words, which job to choose from which factory to move is a question that has not been clearly answered.

An investigation of the literature reveals papers that have solved both Phases I and II of LBA by offering random-based solutions. In this regard, a hybrid brain storm optimization (HBSO) algorithm was presented by [37]. Although the proposed algorithm used some procedures such as k-means and crossover to show its efficiency, HBSO's basis has been based on random in the two mentioned phases. By presenting a shuffled frog-leaping algorithm with memeplex grouping (MGSFLA), Lei and Wang [38] considered the utilizing of random-based methods for both above phases. Besides, Li et al. [39] presented a discrete artificial bee colony algorithm (DABC), Ying and Lin [3] developed a self-tuning iterated greedy (SIG) algorithm, and Cai et al. [40] presented a new shuffled frog-leaping algorithm with memeplex quality (MQSFLA). Moreover, Phase II has not been given much attention in other distributed shop problems such as scheduling in distributed hybrid (or flexible) job shop problem (DHJSP) [41,42] like the research introduced above.

In summary, we can draw several conclusions from the literature review. First, many metaheuristics have led researchers to spend time on explaining how a metaheuristic works, and multiple operators and terms have led to less understanding for new researchers in this field. Second, several heuristics have been proposed to prevent a metaheuristic from falling into a local optimum. In other words, getting stuck in a local optimum is a problem that exists in metaheuristics, and to find the global optimal solution, researchers have developed other heuristics in addition to implementing the structure of each metaheuristic. Lastly, although some steps have been taken for LBA Phase I, effective solutions for LBA Phase II is considered a research gap. Motivated by providing methods to solve each of the above issues, the present paper uses CMCS as a single-point metaheuristic and develops few concepts and methods to cover the research gap.

## 3. Preliminaries

In this section, we first formally describe the problem under study. Then, we present two newly developed weight and impact concepts. Finally, we briefly introduce the modern framework of CMCS, which is used in our solution.

### 3.1. Problem statement

The DHFSP-MT under study consists of $u$ homogeneous factories $\mathcal{F} = \bigcup_{f=1}^{u} \{F_f\}$ in which each factory $F_f \in \mathcal{F}$ is an HFSP-MT that is located in a different location. There $k$ stages in each HFSP-MT, where the $i$th stage $s_i$, where $1 \leq i \leq k$, consists of $m_i$ identical and parallel machines (or processors). The minimum number of machines at any stage is one. In DHFSP-MT, $n$ jobs $\mathcal{J} = \bigcup_{j=1}^{n} \{J_j\}$ with MT must be distributed into $\mathcal{F}$ for processing. Job $J_j$ at stage $s_i$ takes $size_{ij}$ ($\leq m_i$) identical machines simultaneously for $p_{ij}$ unit of time. Here, $size_{ij}$ denotes the number of tasks of $J_j$ to be processed at $s_i$. Let $C_f$ represent the makespan (or completion time) of the processing of the last job of a job sequence $seq_f$ (a subset of $\mathcal{J}$) distributed to $F_f-$ over a deterministic processing time. The objective function of a schedule $S$ is the minimization of the overall makespan and can be denoted as Eq. (1):

$$\mathcal{C}(S) = \max_{f=1,2,\ldots,u} C_f \tag{1}$$

Thus, three decisions are made for each job $J_j \in \mathcal{J}$: (1) in which factory $F_f$ will the job $J_j$ be processed, (2) to which machine/machines the task/tasks of a job should be assigned at each stage $s_i$ in the factory $F_f$, and (3) when will the start time of the processing of $J_j$ at $s_i$ in $F_f$ ($st_{ijf}$) and the completion time of the processing of $J_j$ at $s_i$ in $F_f$ ($ct_{ijf}$) be? For this purpose, the task/tasks must be assigned to the machine/machines at a stage $s_i$ of a factory $F_f$ so that the processing can be done. The objective is to assign a job to a suitable factory and determine the sequence of processing for the jobs assigned to the factory in such a way that the maximum completion time among all factories is minimized.

To solve the problem, we make the following assumptions on machines and jobs: The number of machines in each stage and the number of stages in all factories in $\mathcal{F}$ are the same. Each machine processes only one task at a time and interruption during the processing is not allowed. The size and processing time of $\mathcal{J}$ are fixed during the runtime. The setup time and ready time for each $J_j \in \mathcal{J}$ are considered zero. A list of notations along with their descriptions is given in Table 1.

**Table 1**
List of notations and descriptions.

| Notation | Description |
|---|---|
| $n$ | Number of jobs |
| $k$ | Number of stages |
| $u$ | Number of factories |
| $j$ | Indexes for jobs |
| $i$ | Indexes for stages |
| $f$ | Indexes for factories |
| $\mathcal{J}$ | Set for jobs |
| $\mathcal{F}$ | Set for factories |
| $J_j$ | The job with index $j$ |
| $F_f$ | The factory with index $f$ |
| $s_i$ | The i th stage |
| $m_i$ | Number of machines at $s_i$ |
| $size_{ij}$ | Number of machines needed for processing $J_j$ at $s_i$ |
| $p_{ij}$ | Processing time of $J_j$ at $s_i$ |
| $seq_f$ | The set of jobs assigned into $F_f$ |
| $C_f$ | The maximum completion time of $seq_f$ |
| $S$ | A schedule for processing the jobs $\mathcal{J}$ in $\mathcal{F}$ |
| $\mathcal{C}(S)$, $C_{max}$ | The maximum completion time of the jobs $\mathcal{J}$ among all factories |
| $po_f$ | The processing orders of the jobs at $s_1$ of $F_f$ |
| $st_{ijf}$ | The start time of the processing of $J_j$ at $s_i$ in $F_f$ |
| $ct_{ijf}$ | The completion time of the processing of $J_j$ at $s_i$ in $F_f$ |

As stated, the problem under study was studied for the first time by Ying and Lin [3], which gives the formal mathematical model related to DHFSP-MT. In the following, a simple example is presented to further explain the problem.

**Example.** Consider a DHFSP-MT in which $n = 10$, $k = 2$, $u = 3$, $m_1 = 4$, $m_2 = 2$, and the data are given in Table 2. A feasible schedule $S$ for the problem is given in Fig. 2. According to the figure, $seq_1 = \{J_4, J_1, J_7\}$, $seq_2 = \{J_5, J_2, J_{10}, J_8\}$, and $seq_3 = \{J_9, J_3, J_6\}$. It can be observed that $C_1 = 18$, $C_2 = 23$, $C_3 = 30$, and as result $\mathcal{C}(S) = 30$.

### 3.2. Defining weight and impact

Considering the characteristics of jobs and specifications of machines, we introduce a concept called *weight*, which can be used in both Phase I and Phase II of LBA to have a proper function. Also, we will introduce a concept called *impact,* which refers to the impact of jobs in different stages. These novel concepts are used in this paper to determine the sequencing of a schedule.

**Definition 1** ($w_{ij}$)**.** Eq. (2) defines the *weight* of a job $J_j$ at stage $s_i$.

$$w_{ij} = \frac{size_{ij}}{m_i} + \frac{p_{ij}}{p_i} \tag{2}$$

where $p_i = \max\{p_{ij} | j = 1, 2, \ldots, n\}$ and denotes the largest unit of time among all jobs of $\mathcal{J}$ at $s_i$. For instance, for $J_1$ at $s_1$ in the illustrated Example in Section 3.1, we have $w_{11} = 2/4 + 2/10 = 0.7$.

**Definition 2** ($w_j$)**.** Eq. (3) defines the *total weight* of a job $J_j$ in all $k$ stages.

$$w_j = \sum_{i=1}^{k} w_{ij} \tag{3}$$

In the Example, $w_{11} = 0.7$ and $w_{12} = 1.7$. Therefore, we have $w_1 = 0.7 + 1.7 = 2.4$.

**Definition 3** ($\imath_j$)**.** Eq. (4) defines the *impact* of a job $J_j$ in all $k$ stages.

$$\imath_j = \imath_{i=1,j} + \alpha \imath_{i=2,j} + \alpha^2 \imath_{i=3,j} + \cdots + \alpha^{k-1} \imath_{i=k,j} \tag{4}$$

**Table 2**
Data for the Example.

| $J$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $p_{1j}$ | 2 | 3 | 4 | 3 | 3 | 1 | 4 | 10 | 8 | 4 |
| $size_{1j}$ | 2 | 1 | 3 | 4 | 2 | 2 | 4 | 2 | 3 | 3 |
| $p_{2j}$ | 6 | 3 | 8 | 9 | 9 | 7 | 2 | 5 | 7 | 6 |
| $size_{2j}$ | 2 | 1 | 2 | 1 | 1 | 2 | 1 | 2 | 2 | 2 |

where $\imath_{i,j}$ refers to the obtained value after applying the min–max normalization method on $w_{ij}$, and $\alpha$ is an exponentially decaying coefficient ($0 \leq \alpha \leq 1$) that can be placed in each stage $s_{i>1}$ to obtain the *impact* based on its distance from the first stage $s_1$.

In Eq. (4), we say $\imath_j$ has a far-sighted impact if $\alpha$ is closer to one and has a short-sighted impact if it is closer to zero. If far-sighted, the closer the stages are to $s_k$, the greater the impact on $\imath_j$. Conversely, if short-sighted, the closer the stages are to $s_k$, the less the impact on $\imath_j$.

It can be seen in the literature that $size_{ij}$, $p_{ij}$, or their multiplication are used to produce the rule/rules. However, such characteristics for each task cannot be used against the characteristics of other tasks or $m_i$ to measure the number of resources used by a task compared to other tasks. Therefore, we introduced Definitions 1 and 2 in order to have new criteria for realistic calculations in each stage.

### 3.3. Conditional Markov Chain Search

Nowadays, reducing human intervention in parameterization to algorithms and designing a mechanism for the presence of computers in parameterization has been considered. On the other hand, the variety of metaheuristics makes it difficult to choose one of them to solve a problem, and the development of heuristics outside the metaheuristic framework can help algorithms to be user-friendly. In this regard, Karapetyan et al. [43] developed a powerful tool called Conditional Markov Chain Search (CMCS) to automate the generation of heuristics through an offline learning manner called *configuration*. CMCS is a single-point metaheuristic based on the basic concept of algorithmic component and each component is taken as a black box. In CMCS, a pool of components is used to solve the problem and each component is independent and stateless and has its internal logic. For example, a mutator (Mut) or a hill climber (HC) can be considered a component. If $H$ is the component pool and $h_1$ and $h_2$ are the only two components in the pool (i.e,. $h_1, h_2 \in H$), then $|H|$ indicates the number of components in the components pool, which is 2. Components are not executed in sequence, but the logic of selecting components of the pool is based on numerical parameters that can be accessed in transition matrices. Hence, there are two transition matrices, $M^{succ}$ and $M^{fail}$ in CMCS, and the size of both matrices is $|H| \times |H|$. The former shows the option/options to select the component/components by the control mechanism if the current component succeeds in improving the solution, while the latter shows the component/components to select otherwise. Thus, the component that must be selected after each of the success or failure conditions in improving the solution of the current component can be extracted in these two matrices. Since the problem under study is not probabilistic, the matrices contain deterministic data (zeros and ones). The pseudocode of CMCS is shown in Algorithm 1.

In CMCS, *configuration* is an automatic process performed to the next component selection in both matrices. It is the goal of this framework to realize automated configuration instead of hand-designing a metaheuristic. To start solving a problem, we need to set a configuration. For this purpose, we need to have training instances, which can be the benchmark instances. Then

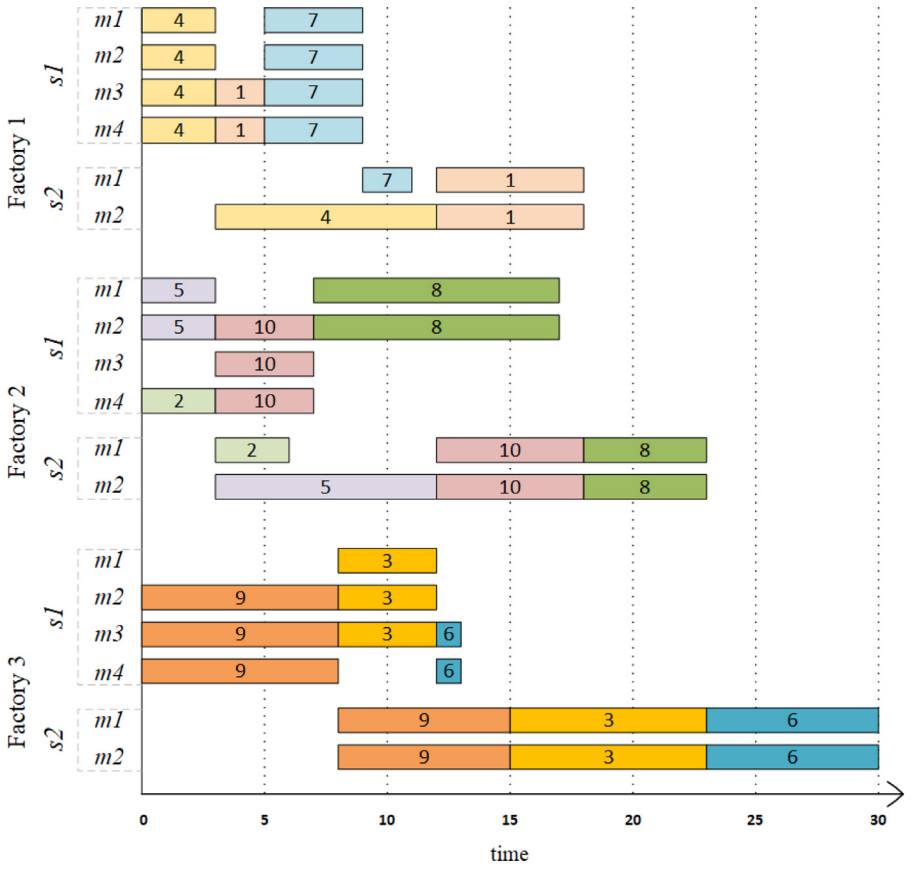



**Fig. 2.** The gantt chart of a feasible schedule for the Example.

**Algorithm 1: Conditional Markov Chain Search**

**input** : components pool $H$; matrices $M^{succ}$ and $M^{fail}$ of size $|H| \times |H|$; objective function $\mathcal{C}(S)$ to be minimized; initial solution $S_0$; termination condition $terminateCond$;

$S^* \leftarrow S_0$; $\mathcal{C}^* \leftarrow \mathcal{C}(S_0)$; $\mathcal{C}_{old} \leftarrow \mathcal{C}^*$;
$h \leftarrow 1$; $i \leftarrow 1$;
**While** $terminateCond$ not met **Do**
  $S_i \leftarrow H_h(S_{i-1})$;
  $\mathcal{C}_{new} \leftarrow \mathcal{C}(S_i)$;
  **If** $\mathcal{C}_{new} < \mathcal{C}_{old}$ **Then**
    $h \leftarrow RouletteWheel(M_{h,1}^{succ}, M_{h,2}^{succ}, \dots, M_{h,|H|}^{succ})$;
    **If** $\mathcal{C}_{new} < \mathcal{C}^*$ **Then**
      $S^* \leftarrow S_i$; $\mathcal{C}^* \leftarrow \mathcal{C}_{new}$
  **Else**
    $h \leftarrow RouletteWheel(M_{h,1}^{fail}, M_{h,2}^{fail}, \dots, M_{h,|H|}^{fail})$;
  $\mathcal{C}_{old} \leftarrow \mathcal{C}_{new}$; $i \leftarrow i + 1$;
**Return** $S^*$;

$$M^{succ} = \begin{matrix} \\ Mut \\ HC \end{matrix} \begin{matrix} \begin{matrix} Mut & HC \end{matrix} \\ \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \end{matrix} \qquad M^{fail} = \begin{matrix} \\ Mut \\ HC \end{matrix} \begin{matrix} \begin{matrix} Mut & HC \end{matrix} \\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{matrix}$$

**Fig. 3.** Transition matrices of CMCS.

each candidate configuration runs on the training instance based on the termination condition. A candidate configuration is shown in Fig. 3 which is related to combinations of elements of two matrices. Afterward, the performance of various candidate configurations is evaluated on training instances and the candidate configuration that has the greatest impact on performance is selected to use in running Algorithm 1. In the figure, if the content of the second row was [1, 0] instead of [0, 1], it can be expected that a new candidate would be created whose performance will be measured. Therefore, in the configuration, we are expecting it to learn the most successful sequence of the components.

After the configuration, it is the turn for Algorithm 1 to execute. Suppose Mut and HC are two components, so $\{Mut, HC\} \in H$, and $|H| = 2$. Consider a condition, in which the elements of $M^{succ}$ and $M^{fail}$ are given according to Fig. 3 and $\mathcal{C}(S_0) = 20$ for an initial schedule $S_0$. In this case, it can be interpreted as follows: the algorithm starts with an initial solution that can be generated randomly or by a construction heuristic. Initializations are performed in lines #1 and #2. In the algorithm, $h$ is used to control the index of each row of the two matrices and $H_h(S_{i-1})$ is a function to produce solution by the component with index $h$. Since $h = 1$, Mut takes the output of the initial solution $S_0$ as the input and modifies the current solution as the output, and we assume that the produced solution is $S_1$ and $\mathcal{C}(S_1) = 18$. If the objective function obtained from the execution of Mut is better than the best objective function found so far, the next component for execution is extracted from $M^{succ}$ (here, HC will be executed). Otherwise, it finds the next component to execute from $M^{fail}$ (here, Mut will be executed). *RouletteWheel*() is a fitness function-based selection mechanism that chooses a component in $M^{succ}$ or $M^{fail}$ uniformly at random. In the algorithm, *terminateCond* is the termination condition, which can be the total number of allowed iterations, an execution time limit, or some user-defined criteria.

Although there are other popular approaches such as MCHH [44], irace [45], and confStream [46] for automated algorithm configuration in the literature, however, receiving feedback during the search process by the candidate components is one of the outstanding features of CMCS, which leads to improvement of the solution quality. In other words, in the configuration, it parametrizes the components in two transition matrices $M^{succ}$ and $M^{fail}$ so that its performance during the search process is managed by these two matrices. This effectively avoids sequential configuration. In addition, its support for solving deterministic and non-deterministic problems and ranking the objective function values make this framework amenable for solving the problem under study. More details about CMCS can be found in [43,47].

## 4. Proposed method

This section first deals with how to represent a solution to the problem. It then discusses how to create the initial solution as CMCS's input parameter. Finally, the components of CMCS are presented in detail.

### 4.1. Solution representation

How to represent the solution is one of the most important decisions made when developing algorithms. This can prevent the algorithm from weakening, especially in large-scaled and complex problems, and help improve the quality and performance of the scheduler. In DHFSP-MT, the distribution of jobs to factories $\mathcal{F}$ and the proper sequencing of jobs in each factory $F_f \in \mathcal{F}$ are two important topics that must be considered in the representation. Permutation-based encoding is the best choice for problems whose variables are discrete and numerical, and at the same time, there should be no repetition of jobs in representation, which has been widely used in the literature [48–51]. Due to the nature of the problem, it is possible to determine the sequence of tasks at stage $s_1$, and for each subsequent stage $s_i$, where $i \geq 2$, the sequence could be determined using the concept of list scheduling [52–54] to decode the solutions. It means, as jobs of $\mathcal{J}$ are available at time zero, a permutation of jobs should be used at $s_1$. But for other stages, the jobs are scheduled based on the FIFO rule, in which the jobs are sorted in non-decreasing order based on their completion time at the previous stage in a new list and then the jobs of the new list are assigned to the machines at the current stage and so on.

Let $S = [po_1, po_2, po_3, C_1, C_2, C_3, C_{max}]$ denote a schedule for our example in Section 3.1. Here, $po_1$, $po_2$, and $po_3$ represent the processing orders of the jobs at the first stage of $F_1$, $F_2$, and $F_3$, respectively. Further, $C_1$, $C_2$, and $C_3$ represent the makespan of jobs at the three factories according to the sequences of $po_1$, $po_2$, and $po_3$, respectively, which are unknown at the beginning of the schedule. Lastly, $C_{max}$ represents the overall makespan of the jobs, as defined in Eq. (1). The schedule for Fig. 2 can thus be denoted as follows: $S = [[4, 1, 7], [5, 2, 10, 8], [9, 3, 6], 18, 23, 30, 30]$.

Regarding allocating machines to the list at $s_i$ where $i \geq 2$, there is a point worth mentioning. Let $ct_{ijf}$ and $st_{ijf}$ denote the completion time and start time of the processing of job $J_j \in \mathcal{J}$ at stage $s_i$ in factory $F_f \in \mathcal{F}$, respectively. Suppose $po_k = [J_j, J_{j'}, J_{j''}]$ is the processing order of three jobs of $\{J_j, J_{j'}, J_{j''}\} \in \mathcal{J}$ at $s_1$ in a schedule $S$ and $ct_{1jf} \leq ct_{1j'f} \leq ct_{1j''f}$. In this case, according to what can be deduced from the list scheduling algorithms, one may expect that $st_{2jf} \leq st_{2j'f} \leq st_{2j''f}$. However, in some cases it may be that $st_{2j''f} < st_{2j'f}$. For example, in the schedule of Fig. 2, we have $ct_{141} < ct_{111} < ct_{171}$ but $st_{241} < st_{271} < st_{211}$. This is because $size_{21}$ is bigger than the available free machine and $p_{27} \leq ct_{241} - ct_{171}$, so $J_7$ was processed earlier than $J_1$. This is known as *backfilling* in the scheduling literature [55,56]. In this paper, all the list jobs that are placed after the current assigned job to the machine/machines are examined to see if backfilling is possible to avoid the idleness of machines.

### 4.2. Initial solution

In Algorithm 1, the initial solution is considered as an input to CMCS, which can play a significant role in speeding up the convergence. Therefore, it is important to start with a reasonably good initial solution.

Given that the set of jobs $\mathcal{J}$ must be distributed among the members of $\mathcal{F}$, a Phase I LBA approach should be viewed to achieve a suitable schedule. To that end, we propose a Three List Scheduling Construction (3LSC) algorithm based on the concepts of constructive heuristics. The 3LSC algorithm uses three priority lists, each of which follows a certain logic to arrange the order of jobs in the list. The general idea is to calculate the objective function of each list after it is constructed, and then consider the list with the least makespan as the initial solution. The algorithm of 3LSC is shown in Algorithm 2.

**Algorithm 2: Three List Scheduling Construction (3LSC) algorithm**

**input** : a set of jobs $\mathcal{J}$;

**output** : a feasible solution $S\_out$;

01 $S\_out, S_1, S_2, S_3, L_1, L_2, L_3 \leftarrow \emptyset$;

02 **For** each $j$ in $\mathcal{J}$ **Do**

03 compute $w_{1j}$ and record it in $j$;

04 $L_1 \leftarrow push\_back(j)$;

05 sort $L_1$ in increasing order of $w_{1j}$; // smallest weight first

06 $S_1 \leftarrow DMM(L_1)$;

07 **For** each $j$ in $\mathcal{J}$ **Do**

08 compute $w_j$ and record it in $j$;

09 $L_2 \leftarrow push\_back(j)$;

10 sort $L_2$ in increasing order of $w_j$; // smallest weight first

11 $S_2 \leftarrow DMM(L_2)$;

12 **For** each $j$ in $\mathcal{J}$ **Do**

13 compute $(i_j)$ and record it in $j$;

14 $L_3 \leftarrow push\_back(j)$;

15 sort $L_3$ in increasing order of $i_j$; // smallest impact first

16 $S_{3'} \leftarrow DMM(L_3)$;

17 sort $L_3$ in decreasing order of $i_j$; // highest impact first

18 $S_{3''} \leftarrow DMM(L_3)$

19 $S_3 \leftarrow (S_{3'} < S_{3''})\ ?\ S_{3'} : S_{3''}$;

20 $S\_out \leftarrow minimum(S_1, S_2, S_3)$;

21 **Return** $S\_out$;

In Algorithm 2, the overall idea is to make three lists one by one. Therefore, in lines #2 to #5, a schedule is generated based on Definition 1; in lines #7 to #11, a schedule is generated based on Definition 2; and in lines #12 to #19, two lists are generated based on the concept of impact, which was described in Definition 3. Next, the list is sorted in increasing/decreasing order of the weight/impact, and then is assigned to $\mathcal{F}$ to get a schedule (and the schedule length). For instance, for $L_1$, after computing the weight $w_{1j}$ for a job in line #3 and appending the job to the list in line #4, $L_1$ is sorted in line #5. In the end, the list is considered as the input of an algorithm called Distribution Method based on Makespan (DMM) to obtain the desired schedule (line #6). A similar process is performed for $L_2$ and $S_2$. But about $L_3$ as an innovation of this paper, it is a bit different. In $L_3$ preparation, after calculating $i_j$ for all jobs (in line #13), the jobs will be sorted in ascending and descending order of $i_j$ in lines #15 and #17, respectively. After applying DMM to compute the objective function, the order that leads to the minimum objective function is considered as the output of $L_3$ (line #19). After all three schedules are obtained and makespan is specified, on line #20, the schedule with the lowest makespan is selected as the output of 3LSC.

DMM, an LBA in Phase I as shown in Algorithm 3, tries to keep the length of the schedule to a minimum. The logic of DMM is that it greedily assigns jobs to factories based on processing time. At first in line #1, the processing order of jobs ($po_f$) in $S$ is empty and as the algorithm proceeds each $po_f$ will be filled. The outer while-loop is repeated as long as there is a job in list $L$. In line #3, the first job $j$ in $L$ is removed. Then, greedily in the inner while-loop, $j$ will be assigned to all factories temporarily and the makespan of them will be computed in line #7. In line #10, the minimum makespan is identified among all factories, and $j$ is definitively assigned to the factory with the lowest makespan in line #11. Once the order of jobs in all the factories has been determined, it is considered as a schedule, and makespan is calculated in line #12.

The proposed 3LSC algorithm is a fast, dedicated heuristic that computes three schedules by considering the weight/impact of jobs. To assign the jobs to the factories, it uses DMM as an LBA method to minimize the schedule length. Finally, one of the three schedules that minimize the makespan is selected as the initial solution of CMCS.

#### 4.2.1. Time complexity

To analyze the time complexity, suppose the number of jobs, number of stages, and number of factories are $n$, $k$, and $u$, respectively. The time complexity of 3LSC algorithm is composed

**Algorithm 3: Distribution Method based on Makespan (DMM)**

**input** : list of jobs $L$; set of factories $\mathcal{F}$;
**output** : a feasible solution $S$;
01 $S \leftarrow \emptyset$;
02 **While** $L \neq \emptyset$ **Do**
03 $j \leftarrow L.pop\_front()$;
04 $i \leftarrow 0$;
05 **While** $f < \mathcal{F}.length()$ **Do**
06 $po_f \leftarrow push\_back(j)$;
07 compute ($C_f$) and record it;
08 $po_f \leftarrow po_f - j$;
09 $f \leftarrow f + 1$;
10 compute ($C_{max}$) and record the index of $C$ with minimum makespan in $m$;
11 $po_m \leftarrow push\_back(j)$;
12 compute $C_{max}(S)$;
13 **Return** $S$;

of three parts to produce the solutions of $S_1$, $S_2$, and $S_3$, each of which needs to execute DMM. It is easy to find that the time complexity of DMM is $O(nu)$. Lines #2 to #6 need to run to produce $S_1$. The first for-loop in line #2 iterates $n$ time. Thus, it takes $O(n)$ time. Since the sort method in line #5 takes $O(n \log n)$ time, the complexity to produce $S_1$ is $O(n \log n + nu)$. Lines #7 to #11 are executed to produce $S_2$, and the time complexity of those lines is similar to lines #2 to #6. Clearly, the time complexity of lines #12 to #16 is $O(nk + n \log n + nu)$ and the line #18 is $O(nu)$. The other lines take constant time. Thus, the overall time complexity of the 3LSC algorithm is as follows: $O(3LSC) = O(n \log n + nu + nk)$.

### 4.3. Components

This subsection will describe in detail the components of CMCS. In this paper, we present three components, which are Q-Learning-based Component, Load Balancer Component, and Classical Operator Component for the component pool, i.e., $|H| = 3$.

#### 4.3.1. Q-Learning-based component

The use of machine learning algorithms in scheduling problems has been considered in order to extract rules from the characteristics of the problem. Their flexibility in adjusting the parameters of the model has led them to be used instead of heuristic algorithms as fixed structure methods. Among the types of machine learning algorithms, reinforcement learning algorithms, as self-adaptive approach for sequential decision problems, can learn appropriate solutions through interaction with the environment [57]. Methods based on reinforcement learning such as dynamic programming, monte carlo, and temporal difference can provide a map in the environment without prior knowledge and a fixed model of the environment. The three mentioned methods can be classified based on two important features: model-based and model-free. While dynamic programming belongs to the category of model-based method, the other two belong to the category of model-free method. Since the environment model is not needed in model-free methods and the optimal/near-optimal solution can be obtained through a number of experiments, monte carlo and temporal difference are more popular for complex scheduling problems. Between the two, temporal difference learning has distinctive features such as sampling and bootstrapping, which have advantages such as low variance and online updating of estimates. Temporal difference learning has a number of fundamental methods such as Q-Learning, state–action–reward–state–action (SARSA), and R-learning, each of which has advantages and is used for specific environments. We refer to the literature for details of these mentioned concepts [58,59]. Because Q-learning takes the shortest path and has a higher convergence speed, we use it in the present work.

Q-Learning (QL) is an off-policy algorithm that updates its knowledge based on a trial-and-error approach [60]. QL works in an environment that is model-free and explores the environment with an action selection method like $\varepsilon$-greedy to balance between exploration and exploitation. In QL, the learning is based on an action-value function (Q-table) that specifies the expected performance for performing a specific action $a$ in a specific state $s$. The core of the algorithm is the iteratively updating of the pair $(s, a)$ in Q-table based on the Bellman Equation in which each pair has its own Q-value. Details of QL can be found in [59].

The first step to solving a problem with QL is to formulate it under the Markov Decision Process (MDP). MDPs provide a mathematical framework for modeling decision-making in situations where the results are partially random and partially under the control of a decision maker. In other words, MDP is a model for successive decisions whose results are uncertain. One property of MDP is that the next state depends only on the current state and the decision maker's action, but not those of the previous states. Since DHFSP-MT includes HFSP-MT in each factory, if we could attribute HFSP-MT to MDP, the problem under study can be solved with QL. To that end, we observe that the beginning

of the processing at stage $s_i$ is related to the order of jobs at the end of the processing at stage $s_{i-1}$ and likewise, the order of jobs at the beginning of the processing at $s_{i+1}$ is related to the end of the processing at $s_i$. In other words, in any factory, the processing order of jobs at $s_{i+1}$ is only related to $s_i$ and not related to $s_{i-1}$. To further explain, let $po_k = [J_j, J_{j'}]$, $m_1 = 2$, $size_{1j} = 1$, $size_{1j'} = 1$, $p_{1j} = 2$, and $p_{1j'} = 1$. If $st_{1jk} = st_{1j'k} = 0$, then, the jobs order at the end of $s_1$ will be $[J_{j'}, J_j]$ because $ct_{1j'k} < ct_{1jk}$. At $s_2$, if $m_2 = 2$, $size_{2j} = 1$, $size_{2j'} = 1$, and $p_{2j'} \gg p_{2j}$, then $ct_{2j'k} > ct_{2jk}$ and as result, the jobs order at the end of $s_2$ will be $[J_j, J_{j'}]$. In addition, what was stated about backfilling in Section 4.1 also strengthens this property.

We can now formulate HFSP-MT as a QL problem, with the following representations:

- **Agent**: The agent perceives and interacts with the environment to achieve its goals. In DHFSP-MT, one agent is assigned to work on each factory $F_f$ that is HFSP-MT. Therefore, a multi-agent system is used to solve the problem, in which each agent is a QL algorithm.
- **States**: The state space $\mathbb{S} = \{s^1, \ldots, s^N\}$ is defined as a finite set of states that the HFSP-MT can be in, where $N = k$ is the size of the state space. A state is defined as a tuple <stage, job> in which every job will transfer to the next stage after choosing action/actions according to $size_{ij}$ in a particular state. Therefore, $s^1 = \langle s_1, \mathcal{J} \rangle$.
- **Action**: The action space $\mathcal{A} = \{a^1, \ldots, a^K\}$ is a set of finite actions that can be performed in a particular state. Each state has a particular action space, so the number action spaces is equal to $N$. In a particular state, each of the parallel machines in that state is considered as an action. Thus, for $s^1$, we have $K = m_1$.
- **Policy**: Policy $\pi$ is considered as a guide to the agent that specifies what action the agent is allowed to take in different states. Therefore, there are different policies in each state, where each of them can be shown by $\pi^{s^x}$. The goal in QL is to access the optimal policy $\pi^* = [\pi^{s^1}, \ldots, \pi^{s^N}]$. Here, $\pi^*$ refers to a sequence of policies, indicating which $\pi^{s^x}$ should be taken in each of the states.
- **Reward**: The reward $\mathcal{R}$ is the set of immediate feedback we call reward $r$ that is kept after the evaluation of each policy $\pi$. This study uses $1/C_f$ as the reward $r$ for a policy $\pi$. For each $\pi^{s^x}$, $r_{s^x}$ is considered, which is the reciprocal of the completion time of the factory under $\pi^{s^x}$.

Initially, as the QL-based method is a component of CMCS and we named it Q-Learning-based Component (QLC), it gets $po_x$ as the input sequence. The Q-table will fill based on $\mathbb{S} \times \mathcal{A} \rightarrow \mathcal{R}$. This is the reason to obtain the quality of each combination as a real number. The pseudo-code of the proposed QL-based algorithm is detailed in Algorithm 4.

Algorithm 4 takes an initial solution in the search space, Q (a Q-table of state–action values), timestamp, and episodeTerminate as inputs. The timestamp with an integer value is related to the innovation added to this algorithm and episodeTerminate is equivalent to the maximum training session or episode. $\alpha$ and $\gamma$ are learning rate and discount factor, respectively. In each while-loop, the agent explores the environment to enhance its knowledge which is represented by Q-values. Each iteration of the for-loop in line #2 processes one state of $\mathbb{S}$ in which different policies are created. In the same way, in the for-loop in line #3, the acquisition of knowledge is based on actions. It takes every action based on $\varepsilon$-greedy which is a specific instance of soft-policy to try different actions. If $\varepsilon = 0.1$, it selects the optimal action in a state with a probability of 0.9 based on current estimates at time $t$, and on the other hand, it uses a set of non-optimal actions with a probability of 0.1. Therefore, in line #4, by observing $s^t$, an action $a^t$ is selected with the $\varepsilon$-greedy policy. Then, in line #5, after performing the action $a^t$, the reward $r_{s^t}$ and the next state $s^{t+1}$ will be observed. Finally, the value of Q will be updated for $s^t$ and the action $a^t$ in line #6.

One of the flaws in QL is that the size of the Q-table gradually increases with the environment complexity and it could take a long time to learn the environment. For this reason, techniques like Deep QL were used for complex problems [7,61]. This paper tries to find near-optimal solutions by combining elite selection and neighborhood search and adding it to QL. Therefore, we can call this component a hybrid Q-learning-local search approach. For this purpose, it uses a variable called timestamp. The purpose of the timestamp is to apply the found policies to the environment in certain periods so that it can find the best policies in each period. Although the convergence to $\pi^*$ will eventually occur, it may require a long run. Based on this, after satisfying the condition in line #7 and taking into account the values of Q, the extracted policies are performed in line #8.

It was shown in [50] that to minimize the schedule length, different schedules can be produced and segmented according to a few rules. When a particular segment of schedules is compared, it can be seen that one has performed better than the others. The developed QL-local search approach was inspired by it and assumed that each $\pi^{s^x}$ is a processing order for a $po_f$ and a suitable set of elites can be obtained by separating the best policies from the view of $r_{s^x}$ and combining them. This is the logic of the for-loop in lines #9 and #12. A classical swap mutator has been used in line #15 in which two actions are selected randomly and then swapped. Finally, in line #16, findBestSolution() is responsible for finding the highest-quality policy, which is considered as the output of this hybrid QL algorithm.

In this multi-agent system, the agents work in parallel and after they are all done, the solution provided by them is considered as the output of this component.

*4.3.1.1. Time complexity.* Let $a$ and $s$ be the total number of actions and states, respectively. The for-loop in line #2 takes $O(as)$ time. The time complexity of the if-condition in line #7 is $O(q + v \log v)$, where $q$ is Q length and $v$ is the time requires for-loop in line #9 to execute. Typically, the if-condition term is less than the for-loop term in line #2. Therefore, the while-loop takes $O(eas)$ time, where $e$ is episodeTerminate repeat time. Obviously, the other functions are less than the time complexity of the while-loop. Hence, the overall time complexity of QLC is $O(eas)$.

### 4.3.2. Load Balancer Component

Load balancing in a distributed environment is a major technique that can be used to reduce makespan. Two points are important to balance the loads among factories to achieve the scheduling objective. First, designing a rule to identify factories that are not in balance (i.e., under-loaded or over-loaded). Second, creating a rule to identify and move jobs in over-loaded factories to under-loaded ones. The goal of the proposed rules is to reduce overload and increase machine utilization in factories. After the initial assignment of the jobs to the factories by DMM, in this component, the load balancing status of the environment is investigated.

Load Balancer Component (LBC) gets a solution as input. Then, for each $F_f \in \mathcal{F}$, it computes $w_j$ of jobs that are assigned to them. After summing the weights of the assigned jobs in each factory ($sw_{F_f} = \sum_{j \in po_f} w_j$) and calculating the average weight of all the factories ($\overline{w} = \frac{\sum_{f=1}^{u} sw_{F_f}}{u}$), the factories are divided into three categories of under-loaded, balanced, and over-loaded based on

**Algorithm 4: Applied QL-based algorithm for HFSP-MT**

**input** : an initial solution, Q, timestamp, episodeTerminate;

**output** : a feasible solution $S\_out$;

// $\alpha$ and $\gamma$ are predefined learning paremeters

01 **While** $(now < episodeTerminate)$ **Do**

02 **For** each $s^t$ in $\mathcal{S}$ **Do**

03 **For** each $a \in A(s^t)$ **Do**

04 choose $a^t$ based on Q and $\varepsilon$-greedy policy, and perform $a^t$;

05 observe $r_{s^t}$ and $s^{t+1}$;

06 $Q(s^t, a^t) \leftarrow Q(s^t + a^t) + \alpha[r_{s^{t+1}} + \gamma \max_a Q(s^{t+1}, a) - Q(s^t, a)]$;

07 **If** $(now \,\%\, timestamp == 0)$ **Then**

08 **Apply** Q to the environment;

09 **For** each state in $\mathcal{S}$ **Do**

10 append the three policies that have the lowest $r_s$ to pList;

11 **Apply** Q to the environment;

12 **For** each state in $\mathcal{S}$ **Do**

13 append the three policies that have the lowest $r_s$ to pList;

14 **For** each policy in pList **Do**

15 **Apply** swapMutation();

16 $S\_out \leftarrow$ findBestSolution();

17 **Return** $S\_out$;

$\overline{w}$. On this basis, the categorization of each factory $F_f$ is decided by Eq. (5) as follows.

$$F_f \text{ is } \begin{cases} underloaded, & \text{if } sw_{F_f} < \overline{w} \\ overloaded, & \text{if } sw_{F_f} > \overline{w} \\ balanced, & otherwise \end{cases} \tag{5}$$

This categorization solves the first point mentioned above, and the process is depicted in lines #2 to #5 of Algorithm 5. To illustrate the idea, consider Fig. 2 of the Example. At $F_1$, $po_1 = [J_4, J_1, J_7]$ in which in each iteration of for-loop in line #2, $w_j$ is calculated: $w_4 = 2.8$, $w_1 = 2.4$, $w_7 = 2.1$. Then we will calculate $sw_{F_1}$, that is $sw_{F_1} = 7.3$. Similarly, the total weights for the other two factories that can be calculated as follows: $sw_{F_2} = 9.7$, $sw_{F_3} = 8.9$. Thus the average weight is given by $\overline{w} = 8.6$. In line #5, the factories are placed in two categories: under-loaded (*uList*) and over-loaded (*oList*). In this situation, according to Eq. (5), $F_1$ is under-loaded, and $F_2$ and $F_3$ are over-loaded.

The second point is realized in lines #6 to #13 of Algorithm 5. The main idea is to bring under-loaded and over-loaded factories closer to balance by reducing jobs that have a high weight from *oList* and transferring them to *uList*. To prevent under-loaded factories from becoming over-loaded during this transfer, job/jobs may also need to be transferred from under-loaded factories to over-loaded ones. In other words, jobs should be exchanged between the two factories. Obtaining the difference in the weight of the jobs in both factories to understand the impact in approaching the balance is the method used to identify the jobs. For further explanation, the discussion continues in Fig. 2. After executing the sorting method in line #6, we get $oList = [F_3, F_2]$ and $uList = [F_1]$. Therefore, $uSelected = F_1$ and $oSelected = F_2$ based on lines #7 and #8. In line #11, $w_j$ of each job of the factory in *oSelected* is subtracted from $w_j$ of each job of the factory in *uSelected* and the result is stored in *tempList*. For example, $w_5 - w_4 = 2.3 - 2.8 = -0.5$; $w_5 - w_1 = 2.3 - 2.4 = -0.1$; $w_5 - w_7 = 2.3 - 2.1 = 0.2$. By executing line #12, it can be seen that 1.0 is the largest value, which is due to $J_7$ from $F_1$ and $J_8$ from $F_2$. Thus, the two jobs are exchanged in the two factories. The comment on line #12 states that the last member is not always selected and sometimes the penultimate member is also selected. The selection of the last member is with probability of $2/3$ and the other with probability of $1/3$. The comment on line #7 applies to *uList* and with the opposite logic of line #8. The comment on line #12 states that two jobs whose weight difference has the largest value may not be the only pair to be exchanged. Sometimes the jobs related to the two or three largest values will also be exchanged. This is because the Classical Operator Component (which will be presented in Section 4.3.3) has a random structure and may result in the factory weights that are very unbalanced. In this case, more than one job needs to be exchanged.

LBC benefits from three Local Search algorithms, namely, Block Search, Machine-based Search, and Weight-based Search, to further improve the solutions generated after the exchange. It is worth mentioning that two local search algorithms (Machine-based Search and Weight-based Search) are innovations of this paper. The Local Search method in line #14 is a procedure that

**Algorithm 5: Load Balancer Component**

**input** : an initial solution $S\_in$;

**output** : a feasible solution $S\_out$;

01 **Repeat**

02 **For** each $F_f \in \mathcal{F}$ **Do**

03 calculate $sw_{F_f}$;

04 calculate $\overline{sw}$;

05 cluster each $F_f \in \mathcal{F}$ based on Equation (5) into two lists of $uList$ and $oList$;

//uList is related to under-loaded and oList is related to over-loaded

06 sort $uList$ and $oList$ in ascending order;

07 $uSelected \leftarrow uList.firstMember()$; //or uList.firstMember()+1

08 $oSelected \leftarrow oList.lastMember()$; //or oList.lastMember()-1

09 **For** each $J_j$ in o$Selected$ **Do**

10 **For** each $J_j$ in u$Selected$ **Do**

11 $tempList \leftarrow (J_{oSelected} \in po_f) - (J_{uSelected} \in po_{f'})$;

12 find the largest value in $tempList$; //or values

13 exchange the referenced jobs in the $tempList$;

14 **Apply** Local Search();

15 **Until** termination condition has not been satisfied;

selects the best solution from the three algorithms. Each algorithm produces a variety of solutions based on a different logic from the other algorithms. For each solution $S$ that is considered as input of the Local Search method, each algorithm is executed separately for each of the $po_f$ in $F_f \in \mathcal{F}$ to optimize the previous schedule. In other words, each algorithm is considered as an inter-scheduler. The success of each algorithm in providing a suitable solution may vary depending on the problem instance. Hence, an algorithm in one instance may not be effective, but in another instance, it optimizes the objective function well. In addition, algorithms are structured in such a way that they produce different solutions. The solutions of each algorithm are compared with each other and the algorithm that provides the most appropriate solution is considered as output solution of the Local Search method.

The output of LBC is the best solution found during the execution of the outer loop until the termination condition is satisfied. We used the maximum iteration number as termination condition (LBC-itr). In the following, each algorithm of the Local Search method will be described.

*4.3.2.1. Block Search.* Block Search (BS) is a neighborhood search procedure that was presented in [21]. Its general concept is to randomly select two or more jobs and consider them as a block of jobs. It then moves the block in the sequence of jobs one by one between the jobs to create multiple neighbors. It records the sequences that are not able to improve the solution in a tabu list so that it does not use them in future searches. Note that we do not use tabu list in this paper. The reason is because tabu list is developed for use in meta-heuristic algorithms to solve single-factory problems. However, due to the different nature of the problem under study, the use of a tabu list was omitted. Therefore, in this local search algorithm, there is a nested loop which is repeated by a while-loop. The maximum number of repetitions of the while-loop is the number of factories. The outer-loop of the nested loop will be repeated BS-itr times while the inner-loop will be repeated based on the number of assigned jobs to a factory.

*4.3.2.2. Machine-based Search.* Inspired by the divide-and-conquer algorithm and based on $size_{ij}$, Machine-based Search (MS) provides a suitable local search solution for each $F_f$. The general logic of MS is that it tries to put together those jobs whose total number of required machines is less than or equal to $m_1$ and by creating different permutations, it can create a priority in jobs. In other words, it focuses on $size_{ij}$ and tries to shorten the length of the schedule by putting jobs together in such a way that reduces the idle time of the machines at each time unit. MS has four steps: Divide, Pair finding, Combine, and Eliminate. Its structure is given in Algorithm 6.

MS gets a solution and tries to improve the schedule of each member of $\mathcal{F}$ as follows: for an $F_f \in \mathcal{F}$, in the Divide step, $po_f$ is broken down into sublists as long as each sublist contains a job (line #4). Then, in the Pair finding step, it tries to create sublists whose total number of required machines is less than or equal to $m_1$, considering the number of machines required for each $J_j$ ($size_{1j}$). The sublists are stored in a table, called the PF-table, to be used in the Combine step. The PF-table has two columns and the number of rows is equal to the length of $po_f$. Each row of the first column contains one job of the list while each row of the second column contains all sublists related to the job of the first column. Table 2 depicts the PF-table for $F_2$ (or $po_2$) in the example of Fig. 2, in which $po_2 = [J_5, J_2, J_{10}, J_8]$. Since $m_1 = 4$, it is expected that the total number of required machines in each sublist will be

**Algorithm 6: Machine-based Search**

**input** : a solution $S\_in$;
**output** : a feasible solution $S\_out$;
01 $C_{best}, C_{temp}, po_{best}, po_{temp}, S\_out \leftarrow \emptyset$;
02 **For** each $po_f$ in $S\_in$ **Do**
03 $C_{best} \leftarrow C_f$; $po_{best} \leftarrow po_f$;
04 **Apply** Divide step on $po_f$;
05 **Apply** Pair finding step on $po_f$ members;
06 **Repeat**
07 **Apply** Combine step to create $po_{temp}$;
08 **Apply** Eliminate step on $po_{temp}$;
09 compute ($C_{temp}$);
10 **If** $C_{temp} < C_{best}$ **Then**
11 $C_{best} \leftarrow C_{temp}$; $po_{best} \leftarrow po_{temp}$;
12 **Until** termination condition has not been satisfied
13 $S\_out \leftarrow push\_back(po_{best}, C_{best})$;
14 compute $C_{max}(S\_out)$;
15 **Return** $S\_out$;

at most 4. For $J_5$ in the first row, there are two sublists of $[J_5, J_2]$ and $[J_5, J_8]$. In general, each sublist $[J_j, J_{j_1}, J_{j_2}, \ldots, J_{j_s}]$ that contains $s+1$ jobs should satisfy Eq. (6) below:

$$size_{ij} + \sum_{j'=j_1}^{j_s} size_{ij'} \leq m_i \tag{6}$$

where $j$ is the index of the job in the first column of the table centered on which the sublists are created, and $[j_1, j_2, \ldots, j_s]$ denote the indices for a subset of other jobs available in the list. Note that $j'$ can be null or contain more than one job.

As $size_{15} = 2$, the Pair finding step tries to find one or more suitable job/jobs for $J_5$ that could satisfy Eq. (5). Accordingly, $J_5$ was examined with other jobs, of which only two jobs $J_2$ and $J_8$ fulfilled this condition ($size_{15} + size_{12} \leq m_1$ and $size_{15} + size_{18} \leq m_1$). Now suppose $size_{18} = 1$. In this case, the sublist $[J_5, J_2, J_8]$ was expected to be one of the sublists formed for $J_5$ as well, because $size_{15} + size_{12} + size_{18} \leq m_1$. It is worth noting that the minimum length of the sublist can be one, i.e. it contains only one job. For example, suppose $size_{15} = 4$, in which case the second column for $J_5$ would be $[J_5]$, because $size_{15} \leq m_1$.

After creating the PF-table in the Pair finding step, it is time to run the repeat-until-loop of the MS algorithm (lines #6 to #12). In the Combine step, a row of the PF-table is selected at random. Then, a sublist is randomly selected in the row. The selected sublist is added to a new list, which is initialized to be an empty list. This process continues from among the unselected rows until all the rows have been selected. At the end of the Combine step, we have a new list of jobs. But, there may be duplicate jobs in this list. Therefore, the list must be traversed from the beginning to remove duplicate jobs. The process of eliminating duplicate jobs is done in the Eliminate step. In the end, the new list is considered as a schedule and will be assigned to $F_f \in \mathcal{F}$.

**Table 3**
The PF-table related to $F_2$ of Fig. 2.

| Column 1 | Column 2 |
|---|---|
| $J_5$ | $[J_5]$ , $[J_5, J_2]$ , $[J_5, J_8]$ |
| $J_2$ | $[J_2]$ , $[J_2, J_5]$ , $[J_2, J_{10}]$ , $[J_2, J_8]$ |
| $J_{10}$ | $[J_{10}]$ , $[J_{10}, J_2]$ |
| $J_8$ | $[J_8]$ , $[J_8, J_5]$ , $[J_8, J_2]$ |

The repeat-until-loop is iterated MS-itr times as a termination condition.

For more explanation of the two last steps, consider Table 3 again. Suppose the order of randomly generated row indices is 3, 4, 2, and 1. It means that, at first, the 3rd row related to $J_{10}$ is selected, then, the 4th row related to $J_8$ is selected, and so on. If the selected sublists related to each row are $[J_{10}, J_2]$, $[J_8, J_5]$, $[J_2, J_8]$, and $[J_5, J_2]$, respectively, the new list created after the Combine step is $po_{temp} = [[J_{10}, J_2], [J_8, J_5], [J_2, J_8], [J_5, J_2]]$. According to what was described, the new list after applying Eliminate step will be as follows: $[[J_{10}, J_2], [J_8, J_5], [,], [,]]$ and we have $po_{temp} = [J_{10}, J_2, J_8, J_5]$. After computing the schedule length, the obtained makespan is compared to the best makespan ever found (lines #9 to #11) and the list and the makespan will be recorded on the output solution (line #13). This process is done for all members of $\mathcal{F}$ to form the output solution. Finally, $C_{max}$ is calculated according to Eq. (1) (line #14).

*4.3.2.3. Weight-based Search.* By focusing on weight, a Weight-based Search (WS) makes it possible to identify jobs that complement each other at all stages while minimizing machine idle time at each stage. The evaluation the weight $w_{ij}$ of the jobs showed that although the weight of $J_j \in \mathcal{J}$ is low at $s_i$, it may be high at $s_{i+1}$. If the order of jobs in $po_f$ is ascending or descending on the basis of weights for all stages, there will be more idle time for the machines, and it will lead to an increase in the length of the makespan. In other words, jobs may not match together for all stages. This can be observed in Fig. 4, which shows the Gantt chart of $F_1$ in the example of Fig. 2 in the condition where the jobs are sorted according to $w_{1j}$ and in descending order. It can be seen that this schedule has led to an increase in makespan: in Fig. 2, the length of the schedule was 18, while now it is 22.

Thus, the idea behind this algorithm is to find a pair of jobs, where the first job has low-weight at stage $s_i$ and high-weight at stage $s_{i+1}$, and the second job is the opposite, i.e., has high-weight at $s_i$ and low-weight at $s_{i+1}$. Then, the two jobs will be assigned in $po_f$ as two consecutive jobs. The general structure of the proposed WS algorithm is depicted in Algorithm 7.

First, the weight $w_{ij}$ for each job at each stage is calculated and the weights are kept separately in a list called $L_i$ for each stage $s_i$ (line #6). In order to classify the jobs into two categories of high-weight and low-weight jobs, the average weights of jobs available in $L_i$ are calculated (it is named $avg_i$) and the jobs are divided into two categories based on $avg_i$. Therefore, after calculating the $avg_i$ in line #7, the jobs that are greater or equal than $avg_i$ are in category $L_i^g$ and less than $avg_i$ are in category $L_i^l$ (line #8 and #9).

Afterward, the algorithm generates a new sequence of jobs ($po_{temp}$) to improve the input sequence $po_f$. In particular, several new sequences will be generated by the repeat-until-loop (lines #10 to #30) until the termination condition of WS-itr is met. The identification of a pair of jobs that complement each other is done in the while-loop (lines #12 to #25). In each iteration of the while-loop, three conditions are considered, and if one of them is not fulfilled, the loop terminates. The first condition is the maximum number of iterations, which is twice the number of jobs in $\mathcal{J}$. This condition can be considered as an input parameter of the algorithm. It should be noted that members of $L_i^g$ and $L_i^l$ will be reduced if a pair of jobs is found. Hence, one or both of

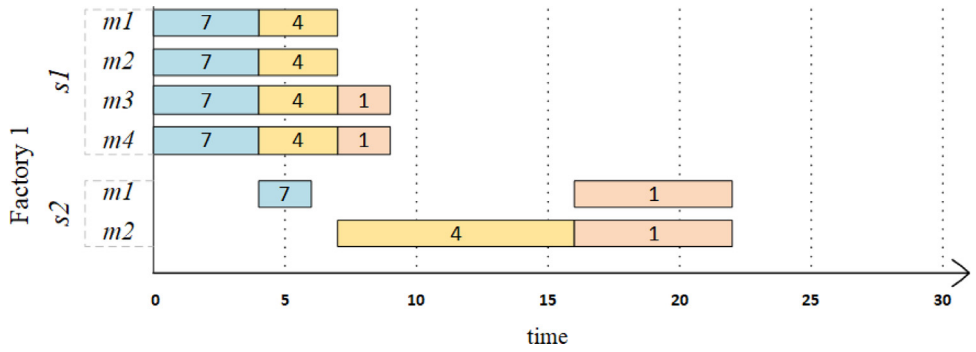



**Fig. 4.** New schedule Gantt chart for $F_1$ in the example of Fig. 2 based on $w_{1j}$.

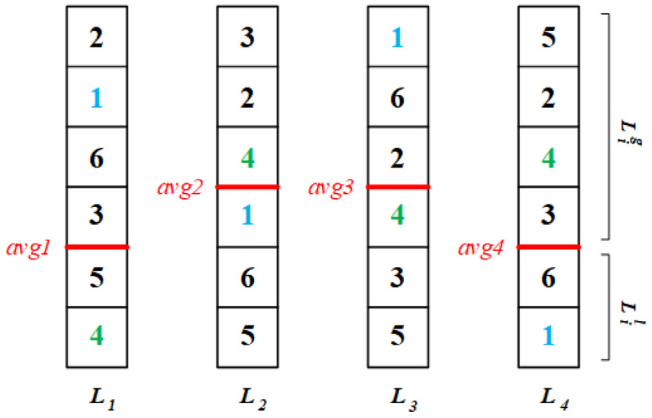



**Fig. 5.** Illustration of WS algorithm.

them may become empty at any stage before the first condition is met, from which it will not be possible to select a task (the algorithm encounters a logical error). Therefore, the other two conditions consider the lengths of $L_i^g$ and $L_i^l$. Then, two jobs $j_g$ and $j_l$ are randomly selected from $L_1^g$ and $L_1^l$, respectively (lines #13 and #14). The two selected jobs in the inner repeat-until-loop (lines #15 to #21) are examined until the last stage so that $j_g$ is in the even stages in $L^l$, and in the odd stages in $L^g$. Similarly, $j_l$ is in the even stages in $L^g$, and in the odd stages in $L^l$. If the stated conditions are not violated by the last stage, the two selected jobs will be considered as a pair and will be appended to $po_{temp}$ in line #23 and they are subsequently removed from $L_1^g$ and $L_1^l$ so that they will not be included in the next selections (line #24). After the conditions of the while-loop condition are met, the number of jobs in $po_{temp}$ is compared to that in $po_f$, and jobs that are not in $po_{temp}$ are transferred from $po_f$ to it (line #26). This is because some jobs may not be appended to $po_{temp}$ as members of any pair, i.e., they are not complimentary at all stages.

Fig. 5 gives an example of WS. Suppose $po_f = [j_4, j_5, j_1, j_2, j_3, j_6]$ and $k = 4$. In this case, there will be four lists, $L_1$, $L_2$, $L_3$, and $L_4$, to record $w_{ij}$. As stated, $w_{1j}$ will be kept in $L_1$. In the figure, $avg_i$ is shown with a red line and $L_i^g$ and $L_i^l$ are separated by the red line. The numbers inside each list indicate the index of each job. For the first iteration of the while-loop in the example, $j_g = j_1$(blue color) and $j_l = j_4$(green color) are selected. Therefore, it can be seen that both jobs have the stated condition for complementarity, that is, they appear alternatively in $L_i^g$ and $L_i^l$ across even and odd stages, and the new list is represented by $po_{temp} = [j_1, j_4, ?, ?, ?, ?]$. If another pair of jobs cannot be added to $po_{temp}$, other jobs are randomly placed instead of each of the symbols of ?. For example, $po_{temp} = [j_1, j_4, j_6, j_5, j_2, j_3]$ may be a schedule.

*4.3.2.4. Time complexity.* To analyze the time complexity of Algorithm 5, it is clear that the repeat-until loop is divided into two main parts: lines #2 to #13 and line #14 in which the three stated local searches are executed sequencly. In the first part, the worst time includes loops of lines #2 and #3, which is $O(un)$, while in other lines, the worst time is $O(n)$. Regarding the first local search of Block Search, it is easy to see that the time complexity is $O(ut_bn)$, where $t_b$ is BS-itr. Regarding the Machine-based Search in Algorithm 6, we see that it applies four steps (Divide step, Pair finding step, Combine step, and Eliminate step). In order to analyze the time complexity of Algorithm 6, it is observable that the time complexity of Pair finding step and Combine step is $O(n^2)$, while the time complexities of Divide step and Eliminate step are $O(n)$ and $O(n \log n)$, respectively. The repeat-until loop takes $O(t_r n^2)$ time because the time complexity of Combine step is more than others in this loop, where $t_r$ represents the termination condition of the repeat-until loop. Totally, the time complexity of Algorithm 6 is $O(ft_r n^2)$, where $f$ is the number of factories in the for-loop. Regarding Algorithm 7, the main loop iterates $f$ times. The main loop contains two parts of for-loop in line #4 and repeat-until loop in line #10. The time complexity of the first part is $O(kn)$ and the time complexity of the second part is $O(t_u kn)$, where $t_u$ is the time of repeat-until-loop in line #10. Therefore, time complexity of this algorithm is $O(ft_u kn)$. Hence, the overall time complexity of Algorithm 5 is $O(ut_b n + ft_r n^2 + ft_u kn)$.

### 4.3.3. Classical Operator Component

Since the problem we consider in this paper is an NP-hard problem [3], the problem space cannot be fully and efficiently explored. QLC and LBC as presented in the preceding sections are two components that explore the problem space in a rule-based manner. The Classical Operator Component (COC), unlike the other two components, tries to construct the solution with the help of random exploration. In this way, one can hope to reach unknown spaces that are otherwise unexplored by rule-based approaches. Two-point Crossover (TC) and Hill Climber (HC) are two algorithms used in this component. While TC is used to construct an inter-factory solution, HC is used for an intra-factory one. Both algorithms are briefly explained below.

TC works with two strategies. In the first strategy, two factories $F_f$ and $F_{f'}$ with the smallest and largest makespan are selected. Then, two positions in $po_f$ and $po_{f'}$ are randomly selected and the jobs between those two positions are exchanged in the two factories. The second strategy is similar to the first strategy, with the difference that $F_f$ and $F_{f'}$ are also randomly selected.

HC, in contrast, is applied to each factory $F_f \in \mathcal{F}$. It selects a job in $po_f$ in a random position $x$ and moves the job to other positions $x+1$, $x+2$, or $x-1$, and $x-2$. Of course, the number of selected jobs depends on the number of jobs in $po_f$. If the number of jobs is more than 20, two jobs will be selected. Iteration number of the HC is considered as the termination criteria, that is, the

**Algorithm 7: Weight-based Search**

**input** : a solution $S\_in$;
**output** : a feasible solution $S\_out$;
$C_{best}, C_{temp}, po_{best}, po_{temp}, S\_out \leftarrow \emptyset$;
**For** each $po_f$ in $S\_in$ **Do**
  $C_{best} \leftarrow C_f$; $po_{best} \leftarrow po_f$;
  **For** each stage $i$ **Do**
    **For** each job $j$ **Do**
      $L_i \leftarrow push\_back(compute(w_{ij}))$;
    $avg_i \leftarrow$ the average weight of the members of $L_i$;
    $L_i^g \leftarrow$jobs in $L_i$ whose weight is at least $avg_i$;
    $L_i^l \leftarrow$jobs in $L_i$ whose weight is less than $avg_i$;
  **Repeat**
    $x \leftarrow 1$; $z \leftarrow 1$;
    **While** ($x < \mathcal{J}.length() \times 2$) AND ($L_i^g != \emptyset$) AND ($L_i^l != \emptyset$) **Do**
      $j_g \leftarrow rand(1, L_z^g.length())$;
      $j_l \leftarrow rand(1, L_z^l.length())$;
      **Repeat**
        $z \leftarrow z + 1$;
        **If** $z$ % 2 == 0 **Then**
          **If** ($j_g$ in $L_z^g$) OR ($j_l$ in $L_z^l$) **Then** break();
        **Else**
          **If** ($j_g$ in $L_z^l$) OR ($j_l$ in $L_z^g$) **Then** break();
      **Until** $z \leq k$
      **If** $z == k$ **Then**
        $po_{temp} \leftarrow push\_back(j_g, j_l)$;
        $L_z^g.removeJob(j_g)$; $L_z^l.removeJob(j_l)$;
      $x \leftarrow x + 1$; $z \leftarrow 1$;
    $po_{temp} \leftarrow push\_back(po_f - po_{temp})$;
    compute $C_{temp}$;
    **If** $C_{temp} < C_{best}$ **Then**
      $C_{best} \leftarrow C_{temp}$; $po_{best} \leftarrow po_{temp}$;
  **Until** termination condition has not been satisfied
  $S\_out \leftarrow push\_back(po_{best}, C_{best})$;
compute $C_{max}(S\_out)$;
**Return** $S\_out$;

$po_f$ length. In addition, COC will be repeated COC-itr times. The general structure of COC is given in Algorithm 8. The COC would take $O(tun)$ time, where $t$ is COC-itr and $n$ is the number of jobs in $po_f$.

## 5. Experimental results

In this section, we first evaluate the performance of the proposed components of the CMCS framework and then compare it

$$M^{succ} = \begin{matrix} \\ LBC \\ QLC \end{matrix} \begin{matrix} \begin{matrix} LBC & QLC \end{matrix} \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \end{matrix} \qquad M^{fail} = \begin{matrix} \\ LBC \\ QLC \end{matrix} \begin{matrix} \begin{matrix} LBC & QLC \end{matrix} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{matrix}$$

**Fig. 6.** Transition matrices of CMCS on small instances.

**Algorithm 8: Classical Operator Component**

**input** : an initial solution $S\_in$;
**output** : a feasible solution $S\_out$;
**Repeat**
  **Apply** TC;
  **For** each $F_f \in \mathcal{F}$ **Do**
    **Apply** HC;
**Until** termination condition has not been satisfied;

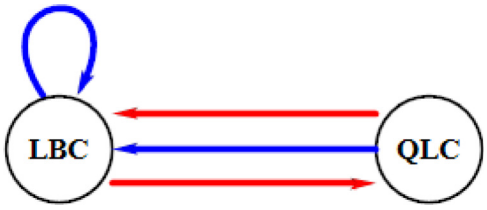



**Fig. 7.** CMCS configuration trained on small instances.

with start-of-the-art algorithms. Finally, we perform a sensitivity analysis of the algorithm.

### 5.1. Benchmark instances

We use the benchmark instances developed by Ying and Lin [3], which can be accessed in http://swlin.cgu.edu.tw/data/DHFSP.zip, for evaluating the performance of the algorithms. The benchmark consists of both small and large sets with a total number of 570 instances that are categorized into 57 groups (i.e., 10 instances per group). Each group is characterized by $n$ jobs, $k$ stages, and $u$ factories. The small set has $n \in \{5, 10\}$, $k \in \{2, 5\}$, $u \in \{2, 3, 4\}$, and the large set has $n \in \{20, 50, 100\}$, $k \in \{2, 5, 8\}$, $u \in \{2, 3, 4, 5, 6\}$. Therefore, there are 12 small groups (hence 120 small instances) and 45 large groups (hence 450 large instances). In this paper, for ease of presentation, each group is represented as $n/k/u$. For example, $5/2/4$ represents a small group with $n = 5$ jobs, $k = 2$ stages, and $u = 4$ factories, and there are 10 instances in the group. The average makespan of the instances of each group is used for comparison.

### 5.2. CMCS configuration

In order to automate the configuration of $M^{succ}$ and $M^{fail}$, which is the goal of CMCS, a training instance set is required. For training, five instances from each small group and five instances from each large group were selected at random. Hence, the training data set included 60 small instances and 225 large instances. The termination condition of each component and the two local search methods are as follows: episodeTerminate (Algorithm 4), LBC-itr (Algorithm 5), and COC-iter (Algorithm 8) $= k \times u \times 10$; MS-itr (Algorithm 6) and WS-itr (Algorithm 7) for small instances $= 100$; MS-itr and WS-itr for large instances $= 200$. Since the problem under study is deterministic, we restrict CMCS to the deterministic case, i.e., we have one non-zero element in each row of the transition matrices. To generate a candidate configuration, one component of the pool is selected randomly as the start component, then the order of other components is chosen randomly until all possible combinations are formed. This process was done for all components such that they are the first selected component to start with. Each feasible configuration is evaluated on the training instances. The time budget given to each candidate configuration is as follows: $0.2 \times n \times k$ (sec).

The transition matrices related to the best performing CMCS configurations for small and large instances are shown in Figs. 6 and 8. Also, to graphically represent the matrices in both instance sizes, we expressed them through automata, which are illustrated in Fig. 7 and Fig. 9. In these automata, each node represents a component of the components pool, and each arc corresponds to a transition. Also, blue arcs indicate the success of the last execution of a component while the red correspond to the unsuccessful transitions. Since the proposed CMCS turns into a deterministic control mechanism, the automata have at most one outgoing success arc and one outgoing failure arc for each component.

It is easy to explain small instances according to Figs. 6 and 7. CMCS chose to use two components from the pool. The configuration focuses on LBC as long as it improves the solution. This may be due to the quality of neighborhood search algorithms and the strategy used for LBA. In case of failure in improvement, it refers to QLC so that it can use the QL capabilities. QLC also refers to LBC in both cases of success or failure. What is evident is that CMCS decided not to use COC during the training process. This is because the success of COC was very small and LBC and QLC were able to achieve satisfactory results by interacting with each other. To conclude, we can see that the algorithm repeatedly applies LBC until it fails, then applies QLC, and then returns to LBC disregarding the success or failure of QLC.

According to Figs. 8 and 9, the configuration of large instances is a bit more complicated than the small ones, but it is interpretable. All three components in the pool were used here. LBC and QLC have good cooperation to improve the solution or reach the local minimum. When each of them improves the solution, they refer to the other for a deeper search. But the presence of both is notable in times of failure, which is referring to COC. This is because COC is applied to diversify the search based on a random manner. If COC succeeds, it returns to LBC and otherwise to QLC.

What can be understood from the configuration on small and large instances is that in the first step, LBC is selected. Also, LBC is the first choice of other components after improvement. This shows the effectiveness of the logic presented for LBA and also the neighborhood searches introduced in it.

The behavior of CMCS with the three developed components is considerable from one point of view, regardless of the above explanations. CMCS plans to use the components without any prior knowledge, which is the nature of automated algorithm configuration. When it is faced with the problem/problems, it considers a component or some components unnecessary after

$$M^{succ} = \begin{matrix} \\ LBC \\ QLC \\ COC \end{matrix} \begin{matrix} \begin{matrix} LBC & QLC & COC \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \end{matrix} \qquad M^{fail} = \begin{matrix} \\ LBC \\ QLC \\ COC \end{matrix} \begin{matrix} \begin{matrix} LBC & QLC & COC \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

**Fig. 8.** Transition matrices of CMCS on large instances.

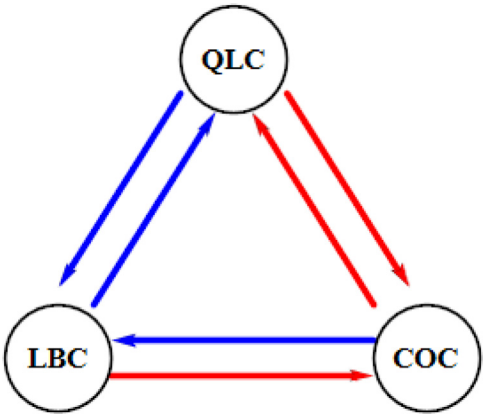



**Fig. 9.** CMCS configuration trained on large instances.

training and does not use the component/components during the solving of the problem/problems. In addition, the quality of each component is also important. It can be expected that the quality of the solution by the generated configuration is better than human-designed metaheuristics, because the use or non-use of the components as well as the order of their use is completely impartial and based on the existing conditions.

### 5.3. Comparison to state-of-the-art

The proposed CMCS is compared against two metaheuristic-based approaches, namely, SIG and DSFLA, which are presented by Ying and Lin [3] and Cai et al. [35], respectively.

- SIG uses an adaptive cocktail decoding mechanism to adjust probabilities of decoding rules dynamically. In the initial step of SIG, it uses a rule to assign priority to each job and assigns jobs to factories based on a makespan-based greedy approach. Then it selects a job from a factory with the largest makespan and moves the job to a factory with the lowest makespan.
- DSFLA uses a dynamic search strategy based on two-point crossover to enhance the search process and increase the resource utilization. It also introduces a dynamic multiple neighborhood search that takes advantage of roulette selection.

The proposed algorithms in this paper were coded in Java, and the experiments were run on a Windows machine with an Intel(R) Core(TM) i7 and 8 GB RAM. To make a fair comparison, we used same the termination condition for CMCS as was used in [3,35], which is $0.15 \times n \times k$ (sec), where $n$ is the number of jobs, and $k$ is the number of stages. CMCS was run five times independently for each instance and the best result was recorded.

The results of the three algorithms in terms of the objective function (makespan) by problem size are reported in Tables 4 and 5. As shown in Table 4, the two-component CMCS configuration improved one of 12 best-known makespan in small groups (as highlighted in bold). In 11 cases, it recorded results similar to SIG and in 10 cases similar to DSFLA.

**Table 4**
Comparison results of CMCS, DSFLA, and SIG on small instances.

| $n/k/u$ | SIG | DSFLA | CMCS | $n/k/u$ | SIG | DSFLA | CMCS |
|---|---|---|---|---|---|---|---|
| 5/2/2 | 16.7 | 16.7 | 16.7 | 10/2/2 | 29.2 | 29.2 | 29.2 |
| 5/2/3 | 16.5 | 16.5 | 16.5 | 10/2/3 | 20.0 | 20.0 | 20.0 |
| 5/2/4 | 15.7 | 15.7 | 15.7 | 10/2/4 | 19.0 | 19.0 | 19.0 |
| 5/5/2 | 34.8 | 34.8 | **34.7** | 10/5/2 | 44.9 | 45.1 | 44.9 |
| 5/5/3 | 35.5 | 35.5 | 35.5 | 10/5/3 | 40.8 | 40.8 | 40.8 |
| 5/5/4 | 34.9 | 34.9 | 34.9 | 10/5/4 | 35.8 | 35.8 | 35.8 |

**Table 5**
Comparison results of CMCS, DSFLA, and SIG on large instances.

| $n/k/u$ | SIG | DSFLA | CMCS | $n/k/u$ | SIG | DSFLA | CMCS |
|---|---|---|---|---|---|---|---|
| 20/2/2 | 47.1 | 47.1 | 47.1 | 50/5/5 | 67.2 | 67.2 | **66.3** |
| 20/2/3 | 36.2 | 36.2 | 36.2 | 50/5/6 | 61.5 | 61.2 | **60.6** |
| 20/2/4 | 29.0 | 29.1 | 29.0 | 50/8/2 | 162.4 | 162.5 | **162.1** |
| 20/2/5 | 21.8 | 21.9 | **21.6** | 50/8/3 | 120.3 | 120.2 | **116.6** |
| 20/2/6 | 20.2 | 20.2 | **19.5** | 50/8/4 | 102.1 | 101.8 | **99.6** |
| 20/5/2 | 70.6 | 70.7 | 70.6 | 50/8/5 | 89.5 | 89.7 | **89.3** |
| 20/5/3 | 53.8 | 54.2 | **52.9** | 50/8/6 | 82.1 | 81.7 | **80.6** |
| 20/5/4 | 46.0 | 46.1 | **45.2** | 100/2/2 | 225.6 | 224.9 | 224.9 |
| 20/5/5 | 42.4 | 42.4 | **41.7** | 100/2/3 | 146.3 | 145.4 | **145.1** |
| 20/5/6 | 40.7 | 40.6 | **38.9** | 100/2/4 | 118.6 | 118.1 | **117.1** |
| 20/8/2 | 89.0 | 89.4 | **88.8** | 100/2/5 | 93.2 | 92.6 | **92.0** |
| 20/8/3 | 74.1 | 74.3 | **73.4** | 100/2/6 | 76.0 | 75.4 | **75.1** |
| 20/8/4 | 66.7 | 66.7 | **65.6** | 100/5/2 | 267.5 | 267.5 | **265.7** |
| 20/8/5 | 62.7 | 62.6 | **59.9** | 100/5/3 | 187.5 | 187.3 | **184.2** |
| 20/8/6 | 60.4 | 60.5 | **57.9** | 100/5/4 | 141.8 | 141.4 | **139.1** |
| 50/2/2 | 114.9 | 114.6 | 114.6 | 100/5/5 | 112.3 | 112.3 | **106.5** |
| 50/2/3 | 81.1 | 81.2 | **80.9** | 100/5/6 | 93.9 | 92.8 | **86.2** |
| 50/2/4 | 63.6 | 63.5 | **63.4** | 100/8/2 | 290.2 | 289.4 | **288.5** |
| 50/2/5 | 47.6 | 47.8 | **47.5** | 100/8/3 | 199.4 | 199.3 | **195.6** |
| 50/2/6 | 39.0 | 39.2 | **38.7** | 100/8/4 | 166.0 | 166.2 | **164.1** |
| 50/5/2 | 147.8 | 147.7 | **147.6** | 100/8/5 | 140.7 | 140.8 | **137.9** |
| 50/5/3 | 103.5 | 103.5 | **102.3** | 100/8/6 | 121.4 | 121.2 | **117.9** |
| 50/5/4 | 75.6 | 75.3 | **74.1** | | | | |

In large groups, the number of improved makespan was very significant as shown in Table 5. CMCS with the three-component configuration outperforms the other two algorithms in 39 out of 45 cases (as highlighted in bold). In the remaining six cases, the results of the three algorithms are similar in three cases, in one case, CMCS is better than DSFLA and in two cases, it is better than SIG. Also, in these six cases, the number of jobs or the number of factories is relatively small. In other words, as the number of jobs or factories increases, the performance benefit of the developed components becomes more prominent.

Overall, CMCS was able to improve the best results obtained by the other two algorithms on 40 of 57 groups of the instances. Also, it recorded the best results in the remaining 17 groups and the recorded results were the same with one or both of the other algorithms.

The efficiency of each of the three algorithms was evaluated for the benchmark using the metric of average percentage deviation (PD). The PD values were calculated based on Eq. (7) as follows:

$$PD = \frac{\mathcal{C}(S)^h - LB}{LB} \times 100\% \tag{7}$$

where $\mathcal{C}(S)^h$ denotes the makespan value obtained by each algorithms of SIG, DSFLA, and CMCS, and $LB$ represents a lower bound on the optimal makespan presented in [35].

**Table 6**
PD and percentage of improvement of CMCS over the best of DSFLA and SIG on small groups.

| $n/k/u$ | PD | | Imp. | $n/k/u$ | PD | | Imp. |
|---|---|---|---|---|---|---|---|
| | Best of SIG&DSFLA | CMCS | | | Best of SIG&DSFLA | CMCS | |
| 5/2/2 | 0.285 (S&D) | 0.285 | – | 10/2/2 | 0.123 (S&D) | 0.123 | – |
| 5/2/3 | 0.684 (S&D) | 0.684 | – | 10/2/3 | 0.229 (S&D) | 0.229 | – |
| 5/2/4 | 0.481 (S&D) | 0.481 | – | 10/2/4 | 0.275 (S&D) | 0.275 | – |
| 5/5/2 | 0.243 (S&D) | 0.239 | 1.65% | 10/5/2 | 0.188 (S) | 0.188 | – |
| 5/5/3 | 0.320 (S&D) | 0.320 | – | 10/5/3 | 0.218 (S&D) | 0.218 | – |
| 5/5/4 | 0.274 (S&D) | 0.274 | – | 10/5/4 | 0.288 (S&D) | 0.288 | – |

Note: S=SIG, D=DSFLA, Imp.=Improvement.

We also have calculated the percentage of improvements of CMCS over the best of the two other algorithms. Both PD and percentage of improvement of the proposed CMCS over the best of SIG and DSFLA on small and large groups are summarized in Tables 6 and 7, respectively. As can be seen in Table 6, CMCS improved the result only in one case (by 1.65%), and in the rest of the cases, there was no improvement. The PD values also indicate that the makespan of all three algorithms is between about 10% and 70% higher than the lower bound (or the optimal solution) depending on the instances.

As shown in Table 7, CMCS provided improvement in almost all cases. Apart from six cases in which no improvement was observed, the minimum improvement was 1.61% and the maximum improvement was 66.12%. In comparing the average of improvements in terms of $n$, the highest improvement was when $n = 100$ and the lowest was in $n = 20$. In terms of $u$, the average of the improvement in $u = 6$ was the highest. Also, the improvement can be observed in all groups when $u = 5$ and $u = 6$. Thus, it can be seen that the amount of improvement increases with the increase of $n$ and $u$. The PD values also show that the makespan of the algorithms is now closer to the LB (or the optimal solution), suggesting their efficacy for solving larger problem instances.

By observing the results of both small and large instances, it can be seen that when the number of jobs is small, the results of all three algorithms are almost the same. However, as the number of jobs increases, CMCS shows its superiority over the other two algorithms. This can also be observed for the number of factories. It can be seen that with the increase in the number of factories, CMCS's efficiency does not decrease, since a suitable strategy has been presented for LBA. It has been observed that in the small instance, there was no need for COC, and LBC with its LBA strategy and neighborhood search was able to perform well like the other metaheuristics. As a result, it can be concluded that the proposed algorithm is more effective than the other two algorithms both in terms of generating the smallest makespan and producing the best solution with very close makespan to LB.

### 5.4. Sensitivity analysis

Sensitivity analysis is a common method in optimization problems to understand the effect of changes in the main parameters of the problem. The potential fluctuations of the parameters may have a significant impact on the optimal solution. Hence, this method can be of great help in the analytical study of changes in input parameters in real-life situations. The focus of the computational study here is to evaluate the impact of the processing time of a job in stages ($p_{ij}$) on the performance of the proposed algorithm. The changes are made with respect to the original values in the interval $[-20\%, 20\%]$ with the step size of 10%. Since the benchmark instances are divided into two categories of small and large, one instance from the group of 10/2/3 is chosen as an example from the small instances and one instance from the group of 50/5/4 is chosen as an example from the large instances. After generation of each parameter in each instance, we solved the optimization problems using the developed CMCS. The obtained results are given in Figs. 10 and 11.

As shown in both Figs. 10 and 11, the makespan has changed proportionally with the change of $p_{ij}$ (between $-20\%$ and 20% of the original value). In other words, the use of machines at each stage has increased as processing time of the jobs increases. This natural tendency in the makespan indicates its high sensitivity against $p_{ij}$ changes. Interpreting these results helps decision-makers make good decisions to improve the environment, for example, by increasing the processing capacity of the machines to result in a decrease in the makespan.

## 6. Conclusion

This paper investigated the Distributed Hybrid Flow Shop scheduling Problem with Multiprocessor Tasks (DHFSP-MT) to minimize the maximum completion time among the factories. In the DHFSP-MT, each factory is a Hybrid Flow Shop scheduling Problem with Multiprocessor Tasks (HFSP-MT). We developed a Conditional Markov Chain Search (CMCS) framework to realize the automated algorithm configuration in the problem under study. Using the characteristics of the HFSP-MT, we introduced two novel concepts of *weight* and *impact* to improve the solution. The developed CMCS has three components: Q-Learning-based Component (QLC), Load Balancer Component (LBC), and Classical Operator Component (COC). For the initial solution of the CMCS, we designed a Three List Scheduling Construction (3LSC) algorithm. In QLC, we first formulated the HFSP-MT as a Markov Decision Process (MDP) and then introduced a hybrid Q-learning-local search approach to find proper scheduling in each HFSP-MT. LBC consists of two steps: applying a strategy for balancing jobs between factories, and improving the makespan of each factory by three local search algorithms. COC included random-based classical methods in order to enhance the quality of obtained solution in DHFSP-MT in both inter-factory and intra-factory aspects.

To evaluate the performance of the developed CMCS, we used a benchmark that was presented in the literature as a test bed. We prepared training sets from the benchmark so that CMCS can learn the environment automatically instead of relying on a handcrafted design. We compared the results of CMCS on 57 instance groups with the results of two state-of-the-art algorithms. According to the results, the CMCS performs better than the two algorithms on 40 of 57 groups of the instances, while in the remaining groups it recorded results similar to both of them.

As a future study, we plan to employ a data mining technique in LBC to analyze the behavior of the stated strategy for job balancing in factories and to extract a series of recurring patterns. In addition, due to the importance of energy consumption optimization in production systems, we intend to consider DHFSP-MT with two objectives of optimization of makespan and energy consumption, and in addition to using a CMCS, customize the concepts of impact and weight for the problem.

**Table 7**
PD and percentage of improvement of CMCS over the best of DSFLA and SIG on large groups.

| $n/k/u$ | PD | | Imp. | $n/k/u$ | PD | | Imp. |
|---|---|---|---|---|---|---|---|
| | Best of SIG&DSFLA | CMCS | | | Best of SIG&DSFLA | CMCS | |
| 20/2/2 | 0.078 (S&D) | 0.078 | – | 50/5/5 | 0.120 (S&D) | 0.105 | 12.50% |
| 20/2/3 | 0.068 (S&D) | 0.068 | – | 50/5/6 | 0.097 (D) | 0.086 | 11.34% |
| 20/2/4 | 0.082 (S) | 0.082 | – | 50/8/2 | 0.027 (S) | 0.025 | 7.41% |
| 20/2/5 | 0.147 (S) | 0.137 | 6.80% | 50/8/3 | 0.088 (D) | 0.055 | 37.50% |
| 20/2/6 | 0.270 (S&D) | 0.226 | 16.30% | 50/8/4 | 0.108 (D) | 0.084 | 22.22% |
| 20/5/2 | 0.093 (S) | 0.093 | 4.76% | 50/8/5 | 0.124 (S) | 0.122 | 1.61% |
| 20/5/3 | 0.180 (S) | 0.160 | 20.69% | 50/8/6 | 0.216 (D) | 0.199 | 7.87% |
| 20/5/4 | 0.233 (S) | 0.212 | 11.11% | 100/2/2 | 0.030 (D) | 0.030 | – |
| 20/5/5 | 0.198 (S&D) | 0.178 | 12.50% | 100/2/3 | 0.072 (D) | 0.070 | 2.78% |
| 20/5/6 | 0.289 (D) | 0.235 | 11.34% | 100/2/4 | 0.035 (D) | 0.026 | 25.71% |
| 20/8/2 | 0.153 (S) | 0.150 | 1.96% | 100/2/5 | 0.057 (D) | 0.050 | 12.28% |
| 20/8/3 | 0.189 (S) | 0.178 | 5.82% | 100/2/6 | 0.053 (D) | 0.049 | 7.55% |
| 20/8/4 | 0.261 (S&D) | 0.240 | 8.05% | 100/5/2 | 0.028 (S&D) | 0.021 | 25.00% |
| 20/8/5 | 0.270 (D) | 0.215 | 20.37% | 100/5/3 | 0.027 (D) | 0.010 | 62.96% |
| 20/8/6 | 0.313 (S) | 0.259 | 17.25% | 100/5/4 | 0.042 (D) | 0.025 | 40.48% |
| 50/2/2 | 0.073 (D) | 0.073 | – | 100/5/5 | 0.098 (S&D) | 0.041 | 58.16% |
| 50/2/3 | 0.066 (S) | 0.063 | 4.55% | 100/5/6 | 0.121 (D) | 0.041 | 66.12% |
| 50/2/4 | 0.050 (D) | 0.048 | 4.00% | 100/8/2 | 0.010 (D) | 0.007 | 30.00% |
| 50/2/5 | 0.067 (S) | 0.065 | 2.99% | 100/8/3 | 0.055 (D) | 0.035 | 36.36% |
| 50/2/6 | 0.102 (S) | 0.093 | 8.82% | 100/8/4 | 0.047 (S) | 0.035 | 25.53% |
| 50/5/2 | 0.021 (S&D) | 0.020 | 4.76% | 100/8/5 | 0.064 (S) | 0.043 | 32.81% |
| 50/5/3 | 0.058 (S&D) | 0.046 | 20.69% | 100/8/6 | 0.103 (D) | 0.073 | 29.13% |
| 50/5/4 | 0.171 (D) | 0.152 | 11.11% | | | | |

Note: S=SIG, D=DSFLA, Imp.=Improvement.

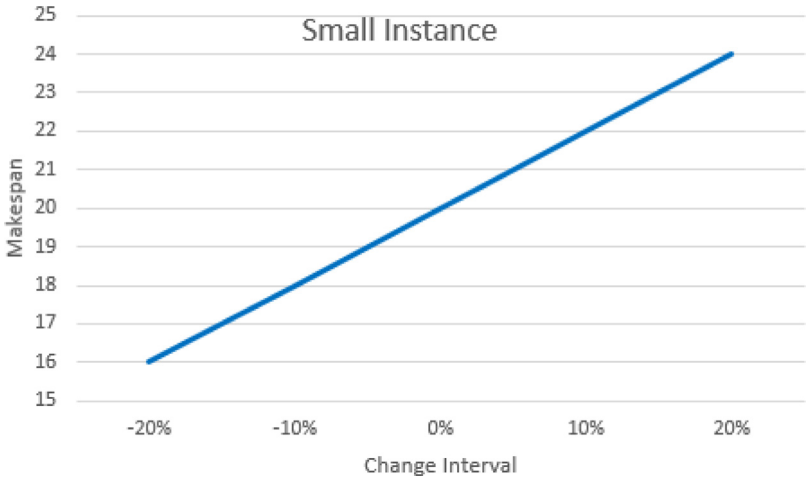



**Fig. 10.** Sensitivity analysis on the small instance.

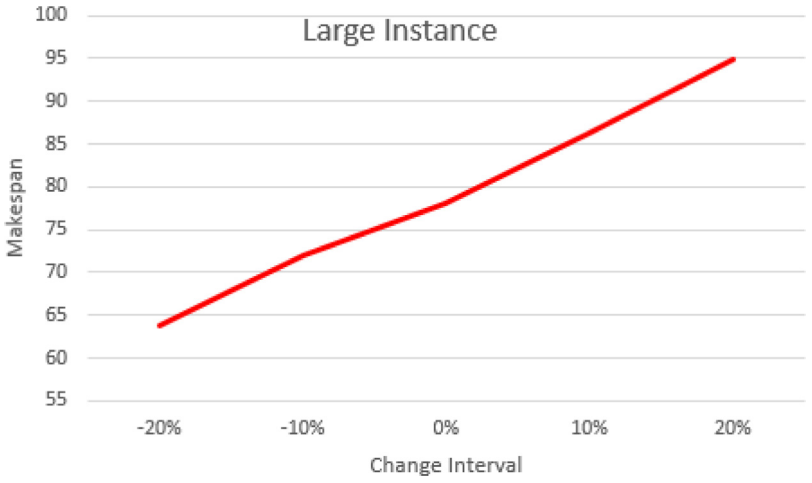



**Fig. 11.** Sensitivity analysis on the large instance.

## CRediT authorship contribution statement

**Hadi Gholami:** Conceptualization, Methodology, Software, Investigation, Writing – original draft. **Hongyang Sun:** Supervision, Validation, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

The data that support the finding of this study are openly available in http://swlin.cgu.edu.tw/data/DHFSP.zip.

## References


[1] H. Gholami, R. Zakerian, A list-based heuristic algorithm for static task scheduling in heterogeneous distributed computing systems, in: 2020 6th International Conference on Web Research, ICWR, IEEE, 2020, http://dx.doi.org/10.1109/ICWR49608.2020.9122306.

[2] H. Sun, P. Stolf, J.-M. Pierson, Spatio-temporal thermal-aware scheduling for homogeneous high-performance computing datacenters, Future Gener. Comput. Syst. 71 (2017) 157–170, http://dx.doi.org/10.1016/j.future.2017.02.005.

[3] K.-C. Ying, S.-W. Lin, Minimizing makespan for the distributed hybrid flowshop scheduling problem with multiprocessor tasks, Expert Syst. Appl. 92 (2018) 132–141, http://dx.doi.org/10.1016/j.eswa.2017.09.032.

[4] C. Oğuz, M.F. Ercan, A genetic algorithm for hybrid flow-shop scheduling with multiprocessor tasks, J. Sched. 8 (4) (2005) 323–351, http://dx.doi.org/10.1007/s10951-005-1640-y.

[5] Z. Min, et al., A Self-Adaptive Load Balancing Approach for Software-Defined Networks in IoT, in: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS, IEEE, 2021, http://dx.doi.org/10.1109/ACSOS52086.2021.00034.

[6] J.-j. Wang, L. Wang, A cooperative memetic algorithm with learning-based agent for energy-aware distributed hybrid flow-shop scheduling, IEEE Trans. Evol. Comput. (2021) http://dx.doi.org/10.1109/TEVC.2021.3106168.

[7] Q. Yan, W. Wu, H. Wang, Deep reinforcement learning for distributed flow shop scheduling with flexible maintenance, Machines 10 (3) (2022) 210.

[8] B. Xi, D. Lei, Q-learning-based teaching-learning optimization for distributed two-stage hybrid flow shop scheduling with fuzzy processing time, Complex Syst. Model. Simul. 2 (2) (2022) 113–129, http://dx.doi.org/10.23919/CSMS.2022.0002.

[9] F. Zhao, et al., A cooperative water wave optimization algorithm with reinforcement learning for the distributed assembly no-idle flowshop scheduling problem, Comput. Ind. Eng. 153 (2021) 107082, http://dx.doi.org/10.1016/j.cie.2020.107082.

[10] F. Zhao, et al., A reinforcement learning-driven brain storm optimisation algorithm for multi-objective energy-efficient distributed assembly no-wait flow shop scheduling problem, Int. J. Prod. Res. (2022) 1–19, http://dx.doi.org/10.1080/00207543.2022.2070786.

[11] W. Meng, R. Qu, Automated design of search algorithms: Learning on algorithmic components, Expert Syst. Appl. 185 (2021) 115493, http://dx.doi.org/10.1016/j.eswa.2021.115493.

[12] C. Oğuz, et al., Hybrid flow-shop scheduling problems with multiprocessor task systems, European J. Oper. Res. 152 (1) (2004) 115–131, http://dx.doi.org/10.1016/S0377-2217(02)00644-6.

[13] C.-T. Tseng, C.-J. Liao, A particle swarm optimization algorithm for hybrid flow-shop scheduling with multiprocessor tasks, Int. J. Prod. Res. 46 (17) (2008) 4655–4670, http://dx.doi.org/10.1080/00207540701294627.

[14] K. Ying, An iterated greedy heuristic for multistage hybrid flowshop scheduling problems with multiprocessor tasks, J. Oper. Res. Soc. 60 (6) (2009) 810–817, http://dx.doi.org/10.1057/palgrave.jors.2602625.

[15] C. Kahraman, et al., Multiprocessor task scheduling in multistage hybrid flow-shops: A parallel greedy algorithm approach, Appl. Soft Comput. 10 (4) (2010) 1293–1300, http://dx.doi.org/10.1016/j.asoc.2010.03.008.

[16] H.-M. Wang, F.-D. Chou, F.-C. Wu, A simulated annealing for hybrid flow shop scheduling with multiprocessor tasks to minimize makespan, Int. J. Adv. Manuf. Technol. 53 (5) (2011) 761–776, http://dx.doi.org/10.1007/s00170-010-2868-z.

[17] K.-C. Ying, Minimising makespan for multistage hybrid flowshop scheduling problems with multiprocessor tasks by a hybrid immune algorithm, Eur. J. Ind. Eng. 6 (2) (2012) 199–215, http://dx.doi.org/10.1504/EJIE.2012.045605.

[18] Y. Xu, et al., An effective immune algorithm based on novel dispatching rules for the flexible flow-shop scheduling problem with multiprocessor tasks, Int. J. Adv. Manuf. Technol. 67 (1) (2013) 121–135, http://dx.doi.org/10.1007/s00170-013-4759-6.

[19] A.D.C. Rani, B. Zoraida, Multistage multiprocessor task scheduling in hybrid flow shop problems using discrete firefly algorithm, Int. J. Adv. Intell. Paradigms 8 (4) (2016) 377–391, http://dx.doi.org/10.1504/IJAIP.2016.080191.

[20] M. Kurdi, Ant colony system with a novel non-DaemonActions procedure for multiprocessor task scheduling in multistage hybrid flow shop, Swarm Evol. Comput. 44 (2019) 987–1002, http://dx.doi.org/10.1016/j.swevo.2018.10.012.

[21] H. Gholami, M.T. Rezvan, A memetic algorithm for multistage hybrid flow shop scheduling problem with multiprocessor tasks to minimize makespan, Int. J. Ind. Eng. Manag. Sci. 7 (1) (2020) 127–145, http://dx.doi.org/10.22034/IJIEMS.2020.118105.

[22] Z. Shao, D. Pi, W. Shao, Hybrid enhanced discrete fruit fly optimization algorithm for scheduling blocking flow-shop in distributed environment, Expert Syst. Appl. 145 (2020) 113147, http://dx.doi.org/10.1016/j.eswa.2019.113147.

[23] W. Shao, Z. Shao, D. Pi, Modeling and multi-neighborhood iterated greedy algorithm for distributed hybrid flow shop scheduling problem, Knowl.-Based Syst. 194 (2020) 105527, http://dx.doi.org/10.1016/j.knosys.2020.105527.

[24] W. Shao, Z. Shao, D. Pi, Multi-local search-based general variable neighborhood search for distributed flow shop scheduling in heterogeneous multi-factories, Appl. Soft Comput. 125 (2022) 109138, http://dx.doi.org/10.1016/j.asoc.2022.109138.

[25] J. Zheng, L. Wang, J.-j. Wang, A cooperative coevolution algorithm for multi-objective fuzzy distributed hybrid flow shop, Knowl.-Based Syst. 194 (2020) 105536, http://dx.doi.org/10.1016/j.knosys.2020.105536.

[26] S. Chen, et al., A population-based iterated greedy algorithm to minimize total flowtime for the distributed blocking flowshop scheduling problem, Eng. Appl. Artif. Intell. 104 (2021) 104375, http://dx.doi.org/10.1016/j.engappai.2021.104375.

[27] F. Zhao, et al., A Population-Based Iterated Greedy Algorithm for Distributed Assembly No-Wait Flow-Shop Scheduling Problem, IEEE Trans. Ind. Inform. (2022) http://dx.doi.org/10.1109/TII.2022.3192881.

[28] K. Karabulut, et al., An evolution strategy approach for the distributed permutation flowshop scheduling problem with sequence-dependent setup times, Comput. Oper. Res. 142 (2022) 105733, http://dx.doi.org/10.1016/j.cor.2022.105733.

[29] F. Zhao, et al., An effective water wave optimization algorithm with problem-specific knowledge for the distributed assembly blocking flow-shop scheduling problem, Knowl.-Based Syst. 243 (2022) 108471, http://dx.doi.org/10.1016/j.knosys.2022.108471.

[30] J.-j. Wang, L. Wang, A cooperative memetic algorithm with feedback for the energy-aware distributed flow-shops with flexible assembly scheduling, Comput. Ind. Eng. 168 (2022) 108126, http://dx.doi.org/10.1016/j.cie.2022.108126.

[31] Z. Shao, W. Shao, D. Pi, Effective constructive heuristic and iterated greedy algorithm for distributed mixed blocking permutation flow-shop scheduling problem, Knowl.-Based Syst. 221 (2021) 106959.

[32] Y. Li, et al., A discrete artificial bee colony algorithm for distributed hybrid flowshop scheduling problem with sequence-dependent setup times, Int. J. Prod. Res. 59 (13) (2021) 3880–3899, http://dx.doi.org/10.1080/00207543.2020.1753897.

[33] W. Shao, Z. Shao, D. Pi, Effective constructive heuristics for distributed no-wait flexible flow shop scheduling problem, Comput. Oper. Res. 136 (2021) 105482, http://dx.doi.org/10.1016/j.cor.2021.105482.

[34] A. Khare, S. Agrawal, Effective heuristics and metaheuristics to minimise total tardiness for the distributed permutation flowshop scheduling problem, Int. J. Prod. Res. 59 (23) (2021) 7266–7282, http://dx.doi.org/10.1080/00207543.2020.1837982.

[35] J. Cai, R. Zhou, D. Lei, Dynamic shuffled frog-leaping algorithm for distributed hybrid flow shop scheduling with multiprocessor tasks, Eng. Appl. Artif. Intell. 90 (2020) 103540, http://dx.doi.org/10.1016/j.engappai.2020.103540.

[36] W. Shao, Z. Shao, D. Pi, Multi-objective evolutionary algorithm based on multiple neighborhoods local search for multi-objective distributed hybrid flow shop scheduling problem, Expert Syst. Appl. 183 (2021) 115453, http://dx.doi.org/10.1016/j.eswa.2021.115453.

[37] J.-H. Hao, et al., Solving distributed hybrid flowshop scheduling problems by a hybrid brain storm optimization algorithm, Ieee Access 7 (2019) 66879–66894, http://dx.doi.org/10.1109/ACCESS.2019.2917273.

[38] D. Lei, T. Wang, Solving distributed two-stage hybrid flowshop scheduling using a shuffled frog-leaping algorithm with memeplex grouping, Eng. Optim. 52 (9) (2020) 1461–1474, http://dx.doi.org/10.1080/0305215X.2019.1674295.

[39] H. Li, X. Li, L. Gao, A discrete artificial bee colony algorithm for the distributed heterogeneous no-wait flowshop scheduling problem, Appl. Soft Comput. 100 (2021) 106946, http://dx.doi.org/10.1016/j.asoc.2020.106946.
[40] J. Cai, D. Lei, M. Li, A shuffled frog-leaping algorithm with memeplex quality for bi-objective distributed scheduling in hybrid flow shop, Int. J. Prod. Res. 59 (18) (2021) 5404–5421, http://dx.doi.org/10.1080/00207543.2020.1780333.
[41] J.-Q. Li, et al., A hybrid Pareto-based Tabu search for the distributed flexible job shop scheduling problem with E/T criteria, IEEE Access 6 (2018) 58883–58897.
[42] Q. Luo, et al., A distributed flexible job shop scheduling problem considering worker arrangement using an improved memetic algorithm, Expert Syst. Appl. 207 (2022) 117984, http://dx.doi.org/10.1016/j.eswa.2022.117984.
[43] D. Karapetyan, A.P. Punnen, A.J. Parkes, Markov chain methods for the bipartite boolean quadratic programming problem, European J. Oper. Res. 260 (2) (2017) 494–506, http://dx.doi.org/10.1016/j.ejor.2017.01.001.
[44] K. McClymont, E.C. Keedwell, Markov chain hyper-heuristic (MCHH) an online selective hyper-heuristic for multi-objective continuous problems, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, 2011, http://dx.doi.org/10.1145/2001576.2001845.
[45] M. López-Ibáñez, et al., The irace package: Iterated racing for automatic algorithm configuration, Oper. Res. Perspect. 3 (2016) 43–58, http://dx.doi.org/10.1016/j.orp.2016.09.002.
[46] M. Carnein, et al., Confstream: Automated algorithm selection and configuration of stream clustering algorithms, in: International Conference on Learning and Intelligent Optimization, Springer, 2020, http://dx.doi.org/10.1007/978-3-030-53552-0_10.
[47] D. Karapetyan, B. Goldengorin, Conditional Markov chain search for the simple plant location problem improves upper bounds on twelve Körkel–Ghosh instances, in: Optimization Problems in Graph Theory, Springer, 2018, pp. 123–147.
[48] H. Sun, Y. Cao, W.-J. Hsu, Fair and efficient online adaptive scheduling for multiple sets of parallel applications, in: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, IEEE, 2011.
[49] H. Sun, W.-J. Hsu, Y. Cao, Competitive online adaptive scheduling for sets of parallel jobs with fairness and efficiency, J. Parallel Distrib. Comput. 74 (3) (2014) 2180–2192, http://dx.doi.org/10.1016/j.jpdc.2013.12.003.
[50] M.T. Rezvan, H. Gholami, R. Zakerian, A new algorithm for solving the parallel machine scheduling problem to maximize benefit and the number of jobs processed, J. Qual. Eng. Prod. Optim. 6 (2) (2021) 115–142, http://dx.doi.org/10.22070/jqepo.2021.14209.1182.
[51] S. Zhou, et al., A self-adaptive differential evolution algorithm for scheduling a single batch-processing machine with arbitrary job sizes and release times, IEEE Trans. Cybern. 51 (3) (2019) 1430–1442, http://dx.doi.org/10.1109/TCYB.2019.2939219.
[52] H. Sun, et al., Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2018, http://dx.doi.org/10.1109/IPDPS.2018.00029.
[53] L. Perotin, H. Sun, P. Raghavan, Multi-resource list scheduling of moldable parallel jobs under precedence constraints, in: 50th International Conference on Parallel Processing, 2021, http://dx.doi.org/10.1145/3472456.3472487.
[54] H. Gholami, M.T. Rezvan, A cooperative multi-agent offline learning algorithm to scheduling IoT workflows in the cloud computing environment, Concurr. Comput.: Pract. Exper. 34 (22) (2022) e7148, http://dx.doi.org/10.1002/cpe.7148.
[55] A. Gainaru, et al., Speculative scheduling for stochastic HPC applications, in: Proceedings of the 48th International Conference on Parallel Processing, 2019, http://dx.doi.org/10.1145/3337821.3337890.
[56] D.A. Lifka, The anl/ibm sp scheduling system, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 1995, http://dx.doi.org/10.1007/3-540-60153-8_35.
[57] F. Zhao, S. Di, L. Wang, A hyperheuristic with Q-learning for the multiobjective energy-efficient distributed blocking flow shop scheduling problem, IEEE Trans. Cybern. (2022) http://dx.doi.org/10.1109/TCYB.2022.3192112.
[58] M. Sewak, Temporal difference learning, SARSA, and Q-learning, in: Deep Reinforcement Learning, Springer, 2019, pp. 51–63.
[59] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[60] Y.-C. Wang, J.M. Usher, Application of reinforcement learning for agent-based production scheduling, Eng. Appl. Artif. Intell. 18 (1) (2005) 73–82, http://dx.doi.org/10.1016/j.engappai.2004.08.018.
[61] Z. Liu, et al., A Graph Neural Networks-based Deep Q-Learning Approach for Job Shop Scheduling Problems in Traffic Management, Inf. Sci. (2022) http://dx.doi.org/10.1016/j.ins.2022.06.017.