Available online at www.sciencedirect.com
ScienceDirect
European Journal of Operational Research 179 (2007) 316–333
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

Discrete Optimization

# An effective architecture for learning and evolving flexible job-shop schedules

Nhu Binh Ho, Joc Cing Tay *, Edmund M.-K. Lai

*Evolutionary and Complex Systems Program, School of Computer Engineering, Nanyang Technological University, Singapore 639798, Singapore*

Received 4 March 2005; accepted 5 April 2006
Available online 12 June 2006

**Abstract**

In recent years, the interaction between evolution and learning has received much attention from the research community. Some recent studies on machine learning have shown that it can significantly improve the efficiency of problem solving when using evolutionary algorithms. This paper proposes an architecture for learning and evolving of Flexible Job-Shop schedules called LEarnable Genetic Architecture (LEGA). LEGA provides an effective integration between evolution and learning within a random search process. Unlike the canonical evolution algorithm, where random elitist selection and mutational genetics are assumed; through LEGA, the knowledge extracted from previous generation by its schemata learning module is used to influence the diversity and quality of offsprings. In addition, the architecture specifies a population generator module that generates the initial population of schedules and also trains the schemata learning module. A large range of benchmark data taken from literature and some generated by ourselves are used to analyze the efficacy of LEGA. Experimental results indicate that an instantiation of LEGA called GENACE outperforms current approaches using canonical EAs in computational time and quality of schedules.


*Keywords:* Genetic Algorithms; Scheduling; Composite Dispatching Rules; Flexible Job-Shop problems

## 1. Introduction

The importance of scheduling has increased in recent years due to the growing consumer demand for variety, reduced product life cycles, changing markets with global competition and rapid development of new processes and technologies. These economic and commercial market pressures emphasize the need for a system which minimizes inventory but is able to maintain customer satisfaction levels of production and delivery specification. Often, this requires an efficient, effective and accurate scheduling plan. The Job-Shop Scheduling Problem (JSP) is one of the most popular scheduling models existing in practice [1]. It has attracted many researchers due to its wide applicability and inherent difficulty [2–5]. It is also well known that JSP is NP-hard [6], and hence deterministic methods of search are in general inefficient. The $n \times m$ classical

* Corresponding author. Fax: +65 6792 6559.
*E-mail address:* asjctay@ntu.edu.sg (J.C. Tay).

0377-2217/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.ejor.2006.04.007

JSP involves $n$ jobs and $m$ machines. Each job is to be processed on each machine in a pre-defined sequence. Each operation of a job is to be processed only on one machine at a time. In practice, the shop-floor setup typically consists of multiple copies of the most critical machines so that bottlenecks due to long operations or busy machines can be reduced [25]. Therefore, an operation may be processed on more than one machine having the same function. This leads to a more complex problem known as the *flexible* job-shop scheduling problem (FJSP). With the new extension, two decisions have to be made: assignment of an operation to an appropriate machine and sequencing the operations on each machine. In addition, for complex manufacturing systems, a job can visit a machine more than once (called *recirculation*). These three features of the FJSP significantly increase the complexity of finding even approximately optimal solutions [7].

The canonical EA has been applied to solve many applications in the real world. However, its results are still limited due to the reliance on randomized natural selection and recombination [8]. Recently, the study of interaction between evolution and learning for solving optimization problems has been attracting much attention from researchers and many promising results [8–12] have been obtained. The diversity and disparity of these approaches has motivated our pursuit for a holistic architecture called LEGA, which supports the interaction between evolution and learning to approximate the FJSP efficiently. This architecture comprises the integration of three modules: Evolutionary Algorithm (EA), schemata learning (SL) and Population Generator. Instead of simply using a randomized search process as in the canonical EA, the search space in every generation is intelligently guided by the SL module towards promising areas while the population generator module is used to generate good initial schedules and in turn, to train the SL module. The SL module is updated by good features of the best schedules during EA execution. The objective is to minimize the makespan; the maximum completion time of all operations. We demonstrate the efficacy of this architecture in the instantiation of GENACE [21] which is applied to common benchmark problems from [13–17], and others generated by ourselves. Experimental results show that using the LEGA architecture, GENACE outperforms previous approaches for solving the FJSP.

The paper is organized as follows. Section 2 gives the formal definition of FJSP. A practical model of the FJSP is also described. Section 3 reviews recent related works for solving the FJSP. Current studies of the interaction between evolutionary and learning for solving optimization problems are also discussed. Section 4 describes the proposed LEGA architecture and the integration of its three modules. The Population Generator module that uses a form of Composite Dispatching Rules (CDRs) and the $k$-nearest neighbor method applied to SL module are also detailed. Section 5 presents and analyzes the performance results of LEGA when applied to solve common benchmarks in literature and others generated by ourselves. Finally, Section 6 gives some concluding remarks and directions for future work.

## 2. Problem formulation

Similar to the classical JSP, the FJSP takes into account the assignment of each operation of each job to a machine and sets its starting and ending times. However, the task is more challenging than the classical one because it requires a proper selection of a machine from a set of machines to process each operation of each job. Generally, the FJSP is formulated as follows:

- Let $J = \{J_i\}_{1 \leqslant i \leqslant n}$, indexed $i$, be a set of $n$ jobs to be scheduled.
- Each job $J_i$ consists of a predetermined sequence of operations. Let $O_{i,j}$ be operation $j$ of $J_i$.
- Let $M = \{M_k\}_{1 \leqslant k \leqslant m}$, indexed $k$, be a set of $m$ machines.
- Each machine can process only one operation at a time.
- Each operation $O_{i,j}$ can be processed without interruption on one of a set of machines $P(O_{i,j})$. Therefore, we denote $p_{i,j,k}$ to be processing time of $O_{i,j}$ on machine $M_k$.
- The objective of this problem is to find a schedule that has minimum time required to complete all operations (minimum *makespan*).

Kacem et al. [15] classified the FJSP into two sub-problems as follows:

- Total FJSP (T-FJSP): each operation can be processed on any machine of $M$.
- Partial FJSP (P-FJSP): each operation can be processed on one machine of subset of $M$.

According to Kacem et al. [15], for the same number of machines and jobs, the P-FJSP is more difficult to solve than the T-FJSP. Therefore, the P-FJSP is transformed to the T-FJSP by adding 'infinite processing times' to the unused machines and to solve the latter instead. However, although the P-FJSP is a generalization of the T-FJSP, in order to compare our experimental results (in Section 5) with the latest results from [15], we will still distinguish between the problem types of T-FJSP and P-FJSP. The flexibility of our representation (to be described in Section 4.3) allows us to use a single representation to describe both problem types. Therefore, the performance results that we obtain will also apply to both.

Constraint information on the FJSP can be illustrated with a table [13,15] where each cell denotes the processing time of that operation on the corresponding machine. Table 1 presents a P-FJSP problem of 2 jobs by 3 machines. Each operation of a job can be processed on more than one machine. The processing time on each machine is predefined. For instance, operation $O_{1,1}$ of job 1 can be processed on machine $M_1$ with processing time 4 units and on machines $M_2$ with processing time 5 units. Machine $M_3$ is unavailable for $O_{1,1}$.

In this paper, we shall assume that:

- All machines are available at time 0.
- All jobs are released at time 0.
- The order of operations for each job is predefined and cannot modified.

### 2.1. Modeling job assignments in the electronic manufacturing service industry

We shall now review the practical use of our FJSP model based on job assignments in the electronics manufacturing service (EMS) industry. The data were collected from CEI Contract Manufacturing Limited Company, Singapore.

Table 1
An example of the P-FJSP

| | | $M_1$ | $M_2$ | $M_3$ |
|---|---|---|---|---|
| $J_1$ | $O_{1,1}$ | 4 | 5 | XXX |
| | $O_{1,2}$ | 9 | 2 | 2 |
| | $O_{1,3}$ | XXX | 6 | 3 |
| $J_2$ | $O_{2,1}$ | 6 | 5 | XXX |
| | $O_{2,2}$ | 3 | 3 | 5 |

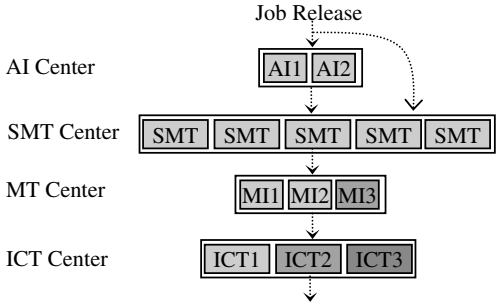


Fig. 1. Typical process flow for PCB assembly.

For job assignments in the EMS industry, printed circuit boards (PCB) generally go though four main centers: Auto Insert center (AI), Surface Mount Technology center (SMT), Manual Insert center (MI), and Independent Component Test center (ICT) [19]. A process flow model of PCB assembly is illustrated in Fig. 1.

The processing of a lot (or job) which contains a batch of identical PCBs begins when the full-kit of materials is ready and released to the shop floor. Each lot has to be processed on a given set of machines in a predefined route that depends on the lot type. It can be processed on the AI center or go directly to the SMT center. For the former route, one of two machines AI1 or AI2 can be selected. For the latter route, one of five machines can be chosen to process it. The MI center has three lines where two are configured for water-based PCBs and one is for non water-based PCBs. Therefore, if a lot belongs to water-based PCBs, it can select one of two machines MI1 or MI2, otherwise, it can only be processed on one non-water based line. Finally, it is processed on one of three machines of the ICT center. After the ICT center, PCBs go through some manual processes (e.g. quality assurance, visual inspection) before undergoing packaging. Observe from the constraints of lots assigned to machines in the assembly line, that the job assignment task in the EMS is well modeled as a FJSP. Moreover, when a two-sided PCB is scheduled to go through the system twice and visit a machine in a center more than once, this latter scenario can be better modeled as a FJSP with *recirculation*.

## 3. Previous works

EAs have been widely used to solve optimization problems [20]. Among the approaches used, the

FJSP is one that has been tackled effectively by appropriate choices of chromosomal representations [13–16,21,22]. Recently, EAs that integrate some form of learning has encouraged further study in evolutionary computation [8–12]. In this section, we review recent EA approaches for chromosome representation as a means of understanding what type of knowledge can be learnt, followed by a review of approaches to integrating such knowledge into an evolutionary learning framework so as to motivate the design of LEGA.

### 3.1. Chromosomal representations for genetic evolution

Better efficiency of EA-based search can be achieved by modifying the chromosomal representation and its related operators so as to generate feasible solutions and avoiding the use of a repair mechanism. For example, the canonical EA often creates a number of infeasible schedules in each generation as it performs crossover and mutation on randomly selected genes. If the chromosomal representation is well designed, no infeasible schedules will be produced after recombination, and the algorithm can expect to be more efficient. Moreover, the relationship between chromosomal representation and its related operators should also be taken into consideration. The operators are only meaningful in the context of a given representation. To evaluate the performance of some reported chromosomal representations [13–16] and their recombination operators, we consider the twin criteria of time and space computational complexity of the EA's operators, and the need to maintain solution feasibility.

Mesghouni et al. [13] proposed a chromosomal representation for FJSP known as parallel job representation. In this representation, a chromosome is represented by a matrix where each row consists of a set of ordered operations of each job. The disadvantage of this representation is described as follows. After doing crossover, the starting times of each operation on each machine could become invalid. Therefore, the offspring requires a repair mechanism to recalculate the starting times for all operations. Due to the complexity of decoding the representation, this algorithm incurs significant computational cost to achieve even near-optimal results. Chen et al. [14] uses an $A$–$B$ string representation. This representation divides the chromosome into two parts: $A$ string and $B$ string. $A$ string contains a list of all operations of all jobs and the machines selected for the corresponding operations while $B$ string contains a list of operations that are processed on each machine. The crossover and mutation operators are costly as the $B$ string has to be checked to ensure that operations in the $B$ string consist only of those operations that are assigned to machine $M_k$ by $A$ string. In addition, this chromosome representation has to consider the order of operations that is processed on the same machine so that when a disjunctive graph is used to decode it, a deadlock situation does not occur. Therefore, a repair mechanism to maintain feasibility is required. Kacem et al. [15,16] use an assignment table representation. It is more efficient than those described previously because all schedules that are generated after applying crossover and mutation remain feasible. Another important feature is the application of domain knowledge in the mutation operation; achieved with a table which specifies promising machines for operations to be processed on (for instance, in terms of processing times). Unfortunately, this knowledge is static. We have extended this feature in LEGA so that the knowledge is dynamic and learnt over each generation. The genetic operators are directed by this knowledge as well. Another drawback of the assignment table is its space complexity. For instance, in the case of P-FJSP, one operation can be processed on a subset of the machines in system. However, a data structure of the assignment table must necessarily describe the set of all machines. This increases the overall time complexity due to the presence of redundant assignments.

The lessons learnt from the above representations motivate some seemingly disjoint features for a possible integration of evolution and learning, they are presented here to motivate the eventual design of LEGA. They are:

1. The use of a chromosomal representation (and its related operators) that only creates feasible space-efficient *offspring schedules* when applying operators. Our proposed chromosomal representation in LEGA will be discussed in Section 4.3.
2. The use of genotypic knowledge that is dynamically acquired during the search process, and to in turn direct future explorations. We will see how this knowledge can be learnt in the following section.

3. The use of an *operation assignment* based composite dispatching rule algorithm instead of being just *job assignment* based.

### 3.2. Learning for evolution

There are several works reported on the application of the interaction between evolution and learning [8–12]. The general approach taken is to keep good and/or bad features of previous individuals to improve the fitness of individuals and/or avoid infeasibility in the current generation. This subsection will review some prominent classes of learning schemes applied to EA that have been reported in literature.

The idea of cultural evolution by Reynolds [8,9] is typically implemented by an external memory that preserves the beliefs encoded in individuals said to represent good *traits* from each generation. In a cultural-based EA (which we call CEA), domain knowledge is represented, stored and transmitted from one generation to the next. Though the quality of schedules improve, our initial experiments show that large amounts of processing time is incurred as learning is applied on every generation. Similarly, Branke [10] uses a memory to keep good individuals. Whenever the environment changes, the new population is created by merging the old population and the individuals in the memory. Louis and McDonnell [11] introduce case-based reasoning methods to select individuals from the case-based memory to be injected into the current generation by using a *similarity* measure. More recently, Michalski [12] presents Learnable Evolution Model (LEM), a learning scheme that uses Darwinian-type evolutionary recombination to generate offspring, where machine learning is used purely to generate a new population. In the current version of LEM, the AQ-type learning system is used to generate the inductive hypotheses.

The above approaches for integrating learning into the evolution of job-shop schedules reveal an interaction between evolution and learning that is implemented in three different ways based on specific problems. In general, they have used version space [8], case-based memory [11] or an AQ-learning system [12]. The common and retaining feature of these approaches is the use of good and/or nogood individuals from previous generation/population(s) to train a learning module or to simply populate a memory. The reported performance results indicate that these approaches can significantly outperform the standard EA on the same benchmarks.

## 4. Integrating genotypic knowledge, learning and population evolution

The canonical EA has been applied successfully to solve many applications [20]. However, the processes are based on random natural selection and recombination. By starting each generation without considering conserved schematic parts of good traits, it may take a long time to converge to the global optimum. As seen earlier in Section 3.2, an integration of proper genotypic knowledge representation, learning methodology and population generation (through an *operation*-based assignment scheduling approach) can yield significant improvements in schedule quality and overall computational time. We therefore propose the LEGA architecture for solving the FJSP. Section 4.1 describes the architecture. Section 4.2 presents the population generator module and its instantiation. Section 4.3 introduces the EA module and our proposed chromosomal representation for FJSP. The SL module and how it integrates with the other modules are discussed in Section 4.4. In particular, changes to the recombination operators are presented in Sections 4.4.3–4.4.5.

### 4.1. The proposed LEGA architecture

The proposed LEGA architecture is functionally divided into three modules; namely, a Population Generator module, EA module and a SL module (see Fig. 2).

The results of the Population Generator module are used to improve the quality of an initial population for subsequent evolution in the EA module. They are also employed to train the SL module. In the GENACE instance of LEGA [21], the CDR population generator algorithm (CDR-PopGen) is used to generate a set of $s$ schedules, which are subsequently encoded into $s$ chromosomes. In Section 5, half of the initial population was generated by CDR-PopGen, the remaining half was generated

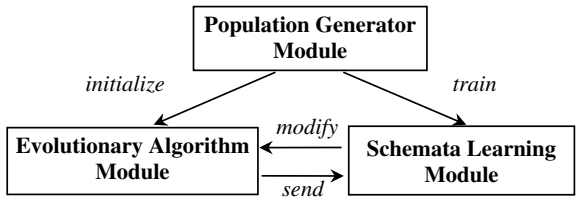


Fig. 2. The proposed LEGA architecture.

randomly. The representational structure of these chromosomes ensures solution feasibility under recombination (selection, crossover, and mutation). During genetic evolution, the SL module will *modify* offspring schedules to improve their *fitness* (or *makespan*) values based on a memory of conserved schemas resolved from sampled schedules *sent* dynamically from the EA module.

In the GENACE instance of LEGA, we use a simple $k$-nearest neighbor method ($k$-NN) [26] for the dynamic learning of schemas. The idea is to find the $k$ most similar solutions to a solution and classify the latter based on a majority vote. The structure of the SL module is shown in Fig. 3. Unlike the previous approaches in [8,11] where knowledge is extracted at every generation, we reduce the computational time and avoid the same good chromosomes that may appear in each generation by updating the SL module only after $k$ generations. Predefined numbers of best chromosomes (like High-group in [12]) after every $k$th generation are selected to be *inserted* to a set of elite chromosomes. The knowledge from these chromosomes is then extracted to *update* the Chromosomal Memory and Operational Memory. The remaining sub-sections will provide algorithmic details on each module of the LEGA architecture.

### 4.2. A CDR-based population generator

In this section, we introduce the CDR-PopGen algorithm as the Population Generation module (in LEGA) used to generate good initial candidates for evolving into fitter flexible job-shop schedules. Its preliminary results are compared with other methods, and the integration with the EA module and the SL module is also discussed.

#### 4.2.1. The CDR-PopGen algorithm

The localization approach by Kacem et al. [15] uses the order of jobs to generate new schedules. It is quite similar to the first-in-first-out (FIFO) rule

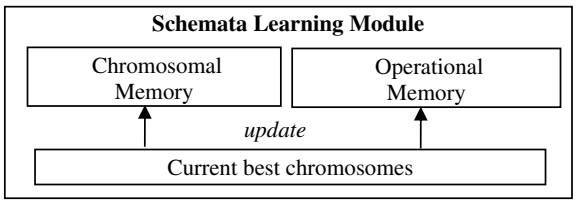


Fig. 3. Structure of the schemata learning module.

[24] where each job is assigned one at a time to machines following a predetermined order. However as mentioned earlier, this *job assignment* may not be efficient because it assigns all operations of one job at a time. For instance, if job $J_m$ precedes job $J_n$ in the predetermined order, all operations of the former job $J_m$ are assigned to the machines before all the operations of the latter job $J_n$. Therefore, $J_m$ has more available machine choices with short processing times and finishes necessarily earlier than the latter. In the CDR-PopGen algorithm, an *operation-based* assignment is used instead to build a schedule. Fig. 4 gives the CDR-PopGen algorithm.

With reference to Fig. 4, note that by the end of Step (0), all first operations of each job would be scheduled. From Step (1), the last operation $O_{i,j}$ on the machine $M_s$ (with earliest stop time) is selected to be evaluated. Note that in Step (1), the machine with minimum processing time for $O_{j,k+1}$ is not necessarily the selected machine. It will depend on the total time that this operation has to wait to be processed in the worst case. Therefore, the workloads of the machines are reduced by balancing operations to be assigned. In the current version of LEGA, at Step (2), three basic dispatching rules are used: shortest processing time (SPT), longest processing time (LPT), or first-in-first-out (FIFO). With one random array $J$ and three basic dispatching rules, we can generate three different schedules. Each of the rules is applied to one random array $J$ at a time. This algorithm can also be extended by using other dispatching rules that are described in [23,24]. The algorithm terminates when all operations are scheduled. The result is a Gantt chart that satisfies all constraints of the FJSP. In Fig. 4, operations are selected one at a time to be scheduled. Therefore, it has a complexity of $O(nm)$, $n$ being the total number of operations of the problem, and $m$ the total number of machines.

#### 4.2.2. Empirical performance of the CDR-PopGen algorithm

In order to test the effectiveness of the CDR-PopGen algorithm, some benchmark T-FJSPs and P-FJSPs were selected from [13,15–17]. In each instance, 20 random orders of jobs were generated. By applying three dispatching rules SPT, LPT, and FIFO to one order of jobs, three different schedules were obtained. Therefore, we have a total of 60 schedules. Another 60 random schedules were also generated to compare to the results from the

Step 0) *Initialize:*
  a) Create an array for a random job order $J$ (e.g. $J_2\ J_3\ J_1$).
  b) Create a Gantt chart data structure $G$ to insert operations.
  c) Assign the first operation $O_{x,1}$ of each job to a suitable machine following the order specified in $J$ to $G$ (e.g. $O_{2,1}$, $O_{3,1}$, then $O_{1,1}$). This machine is selected by calling Step 1) b) to process $O_{x,1}$. Note that when calling Step 1) b), $O_{x,1}$ is assigned directly to the selected machine. If there is no operation assigned to the selected machine, $O_{x,1}$ starts from time 0. Otherwise, it starts from the stop time of the last operation on the selected machine.

Step 1) *Find a suitable machine to process an operation:*
  a) Identify the machine with earliest stop time $M_s$ in $G$ and the last operation $O_{i,j}$ on $M_s$. If more than one machine satisfied, a random machine of them is selected. Let $t$ be the stop time of $M_s$.
  b) Find a suitable machine to process $O_{i,j+1}$ by the following procedure:
    - Identify all the machine that can process $O_{i,j+1}$: $P(O_{i,j+1})$
    - Calculate the waiting time of each machine $M_k \in P(O_{i,j+1})$: if $M_k$ doesn't process any operation at time $t$, it is equal to processing time of $O_{i,j+1}$ on $M_k$. Otherwise, it is processing an operation $O_{z,y}$. Its waiting time is equal to total processing times of operations on $M_k$'s waiting list + (remaining time of $O_{z,y}$ on $M_k$) + processing time of $O_{i,j+1}$ on machine $M_k$.
    - The machine with shortest waiting time is chosen to process $O_{i,j+1}$. If this machine doesn't process any operation at time $t$, assign $O_{i,j+1}$ to it to be processed. Otherwise, add $O_{i,j+1}$ to the waiting list of this machine.

Step 2) *Find a suitable order of operations on a machine:* from Step 1) a), we identified $M_s$ and $O_{i,j}$, then assign next operation $O_{i,j+1}$ to an appropriate machine. Now we find an operation on the waiting list of $M_s$ to be processed on $M_s$. If there is a number of operations on the waiting list of $M_s$, select one of them depending on one of the following dispatching rules: Shortest Processing Time (SPT), Longest Processing Time (LPT), or First In First Out (FIFO).

Step 3) Return to Step 1)

Fig. 4. The CDR-PopGen algorithm for initial population generation in LEGA.

CDR-PopGen algorithm. The best, average makespan results and average processing times of 50 runs of the CDR-PopGen algorithm on a Pentium IV running at 2 GHz are compared.

Table 2 compares the results that were obtained by [13,15,16], random schedule generation, and the CDR-PopGen algorithm. The values in bold-face identify those instances in which the CDR-PopGen algorithm is observed to fare better than the others. The first column presents the size of each problem instance. The second column shows the best makespans obtained from [13,15,16]. Note that the makespans from [15] are obtained by considering a single-criterion objective while the makespans collected from [16] are the best makespans among all the multi-objective results for each instance. The third column identifies where the instance is taken. The fourth column presents the best makespan of the shortest schedule out of 60 schedules that were generated randomly in 50 runs. The fifth column shows the best makespans obtained by the CDR-PopGen algorithm in 50 runs. In the next column, the best makespan of each run is collected. The average result of 50 runs is then reported. The last column shows the average computational times in 50 runs. For the following tables, the method to calculate the average result and the average processing time for each instance is similar to that in Table 2.

Table 2
CDR-PopGen for solving small sized T-FJSPs and P-FJSPs instances

| Job × Mach | Best results | Taken from | Random | CDR-PopGen best | CDR-PopGen avg. | Avg. time (second) |
|---|---|---|---|---|---|---|
| 4 × 5 | 16 | [16] | 11 | **11** | 11.00 | 0.002 |
| 10 × 7 | 15 | [16] | 16 | **11** | 11.00 | 0.006 |
| 8 × 8 | 14 | [15] | 20 | 16 | 16.00 | 0.005 |
| 10 × 10 | 7 | [13,15] | 9 | 8 | 08.00 | 0.006 |
| 15 × 10 | 23 | [16] | 15 | **12** | 12.24 | 0.011 |

We can see that the CDR-PopGen algorithm works well on T-FJSPs. It achieves better results than [16] on 3 instances. In the 10 jobs × 10 machines, in which they obtained the best value of 7 unit times; CDR-PopGen gets a near-optimal value of 8 unit times. In the 8 jobs × 8 machines, CDR-PopGen obtains 16 unit times while the best value from [15] is 14. In these small problems, the makespans of the best random schedules are near to the makespans of the schedules obtained by the CDR-PopGen algorithm. By using a simple Gantt chart structure, this algorithm can be extended to solve the P-FJSP with constraints of *start date* and *due date* of each job and machine maintenance periods.

Table 3 compares the results obtained by Brandimarte [17], random schedule generation and the CDR-PopGen algorithm. The results show that when the problem size increases, CDR-PopGen gets better results than schedules generated randomly. For instance, random generation and CDR-PopGen obtain 49 and 42 unit times, respectively, in solving 10 jobs and 6 machines whereas the results are 338 and 234 unit times in solving 20 jobs and 15 machines. However, in some problems, such as the 20 jobs and 10 machines problem, CDR-PopGen still cannot achieve better results than the Tabu Search algorithm by Brandimarte [17].

The results in Table 3 show that CDR-PopGen obtains reasonable results for large-sized P-FJSPs. A weakness of the CDR-PopGen algorithm is its localized consideration of each machine apart from its surroundings. This has worked well for small-sized problems that are T-FJSPs or P-FJSPs. However, in practice, most scheduling problems are P-FJSPs and they have large sizes. Thus, we believe that the CDR-PopGen algorithm should be cast as an input to other methods to further improve the results.

### 4.3. Evolutionary algorithm module

#### 4.3.1. Chromosomal representation

The canonical EA [20] often creates infeasible solutions during each generation after doing randomized crossover and mutation. Therefore, the effort to enforce feasibility must tradeoff against search efficiency. In the evolutionary computation community, the definitions of *genotype* and *phenotype* are used to distinguish between heritable and non-heritable (or behavioural) traits. EA's recombinant operations are applied to the genotype and the result maps to a phenotype that allows the calculation of a fitness value. However, this is not a bijective mapping from genotype to phenotype. In the classical JSP problem, many mapped instances produce infeasible, redundant and *inactive* schedules [27]. Therefore, additional computational effort is required to convert them to feasible ones. To avoid this complexity, we design a chromosomal representation for solving the FJSP that not only creates feasible chromosomes after genetic recombination but also generates *active* schedules upon decoding. We had earlier seen that Mesghouni et al. [13] used a *parallel machine* and *parallel job representation*, Chen et al. [14] used *A–B string* representation, while Kacem et al. [15,16] used *an assignment table* (described in Section 3). We now present our proposed chromosomal representation for the FJSP. The objectives of our design are primarily:

- To ensure feasible solutions that remain feasible under crossover and mutation.
- To facilitate knowledge update in the SL module.

Our chromosomal representation (called OOMS) has two components: operation order and machine selection (see Fig. 5).

Table 3
CDR-PopGen for solving big sized P-FJSPs instances

| Name | Job × Mach | Brandimarte [17] | Random | CDR-PopGen best | CDR-PopGen avg. | Avg. time (second) |
|---|---|---|---|---|---|---|
| Mk1 | 10 × 6 | 42 | 49 | 42 | 42.00 | 0.009 |
| Mk2 | 10 × 6 | 32 | 36 | 30 | 30.00 | 0.010 |
| Mk4 | 15 × 8 | 81 | 147 | 68 | 68.00 | 0.014 |
| Mk5 | 15 × 4 | 186 | 235 | 179 | 179.30 | 0.017 |
| Mk6 | 10 × 15 | 86 | 102 | 69 | 69.20 | 0.024 |
| Mk7 | 20 × 5 | 157 | 217 | 153 | 153.88 | 0.017 |
| Mk8 | 20 × 10 | 523 | 587 | 527 | 528.44 | 0.035 |
| Mk9 | 20 × 10 | 369 | 455 | 326 | 328.78 | 0.039 |
| Mk10 | 20 × 15 | 296 | 338 | 234 | 236.12 | 0.042 |

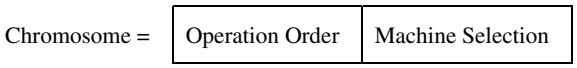


Fig. 5. Structure of proposed OOMS chromosome.

- *Operation order component*. We adopt the operation order representation from [27–29]. Consider the problem in Table 1. Job 1 (or $J_1$) has 3 operations $O_{1,1}$,$O_{1,2}$,$O_{1,3}$; job 2 (or $J_2$) has 2 operations $O_{2,1}$,$O_{2,2}$. One possible schedule could be ($O_{2,1}O_{1,1}O_{2,2}O_{1,2}O_{1,3}$), where $O_{i,j}$ is the operation of job $i$ with order $j$ (decoding the string by reading the data from left to right). However, if we apply GA's operators to this chromosome, we may receive an infeasible solution. For instance, ($O_{2,2}O_{1,1}O_{2,1}O_{1,2}O_{1,3}$) is an infeasible schedule because it violates the precedent constraint; that the second operation of job 2 (or $O_{2,2}$) is to be processed before the first operation of job 2 (or $O_{2,1}$). To avoid creating an infeasible solution, an individual is obtained from this schedule by replacing each operation by the corresponding job index (see Fig. 6). By reading the data from left to right and increasing operation index of each job, a feasible schedule is always obtained.
- *Machine selection component*. We use an array of binary values to present machine selection. For the problem in Table 1, one possible encoding of the machine selection part is shown in Fig. 7.

A boolean value denotes machine selection. For example, $M_2$ is selected to process $O_{1,1}$ because the value in the $M_2$ is a unit value. Only one machine can be selected per operation. For example, operation $O_{2,2}$ can be processed on 3 machines $M_1$, $M_2$, $M_3$, so the valid values are 001, 010 or 100. This

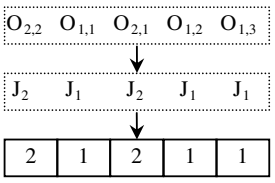


Fig. 6. OOMS: operation order component.

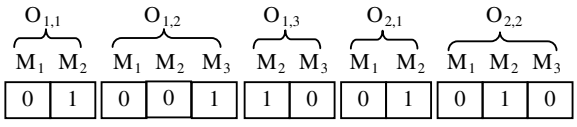


Fig. 7. OOMS: machine selection component.

demonstrates *a FJSP with recirculation* if more than one operation of the same job is processed on the same machine. For instance, $O_{1,1}$ and $O_{1,3}$ belong to $J_1$ and they are processed on the same machine $M_2$. Note that the machines that can process an operation are not redundant as in [15]. For instance, $O_{1,1}$ (in Fig. 7) can select to be processed on one of two machines $M_1$ or $M_2$. Therefore, the OOMS representation is flexible enough to encode the FJSP. It can also represent two problems, T-FJSP and P-FJSP, with the same structure. This property improves the search process by requiring less memory space and ignoring unused data. Practical results of this representation can be found in [21]. Its empirical performance in comparison to other representations has also been shown to be very good [22].

#### 4.3.2. Decoding an OOMS chromosome to a feasible and active schedule

Schedules are often categorized into three classes [18]. A *semi-active* schedule is a feasible schedule where no operation can be started earlier without changing the order or violating the constraints on any one of the machines. An *active* schedule is a feasible schedule where no operation can be started earlier without delaying at least one other operation or violating the constraints on any one of the machines. Finally, a *non-delay* schedule is a feasible schedule where no machine is kept idle while an operation is waiting for processing. It has been verified in [18] that active schedules are a subset of semi-active schedules, non-delay schedules are a subset of active schedules, and active schedules also contain optimum solutions. Therefore, instead of searching the large search space of feasible schedules or semi-active schedules, in our decoding algorithm, only active schedules will be constructed. Since active solutions contain both non-delay and optimal solutions, our decoding algorithm reduces the search space size and still guarantees that an optimal solution can be found.

Fig. 8 gives the algorithm for decoding an OOMS to a feasible and active schedule for a FJSP. Notice that in Step (1) (d), a left-most time gap is detected between two operations processed on the same machine so as to insert a new operation where possible. Therefore, an active schedule is always created when applying this algorithm. Otherwise, a semi-active schedule can be generated by simply inserting the operation to the end of the last operation on the same machine. But the semi-active schedule does

Step 0) *Initialize:* Generate a Gantt chart data structure $G$ to insert operations.
Step 1) *Build active schedule:*
For each integer value reading from left to right in *operation order part*:
a) Identify its corresponding operation $O_{i,j}$.
b) Refer to *machine selection part* to know the machine $M_k$ that processes $O_{i,j}$ and its processing time $p_{i,j,k}$ on $M_k$.
c) Let $t_1$ be stop time of previous operation of $O_{i,j}$ (that is $O_{i,j-1}$) and let $t_2$ be the stop time of the last operation so far on $M_k$ in $G$. If $O_{i,j}$ is the first operation (i.e. $O_{i,1}$), set $t_1$ to be 0. If $M_k$ hasn't processed any operation, set $t_2$ to 0.
d) If $t_2$ is smaller or equal to $t_1$ (i.e. $M_k$ is idle at $t_1$), assign $O_{i,j}$ to $M_k$ starting at $t_1$. Otherwise, from time $t_1$ to time $t_2$ on $M_k$, find time gap between two consecutive operations. The time gap is identified by the stop time of the first operation and the start time of the second operation. If this time gap is equal or larger than $p_{i,j,k}$, assign $O_{i,j}$ to this time gap starting from the left of the time gap. If $O_{i,j}$ cannot insert to any time gaps from $t_1$ to $t_2$, insert $O_{i,j}$ to the end of the last operation on $M_k$ (i.e. it starts at $t_2$).

Fig. 8. Decoding an OOMS chromosome to a flexible and active schedule.

not guarantee to obtain optimal results and its *makespan* is still larger or equal to the active one. The active schedule decoding algorithm has a complexity of $O(n^2)$ while the semi-active decoding algorithm has a complexity of $O(n)$, $n$ being the total number of operations.

#### 4.3.3. Integrating CDR-PopGen to the evolutionary algorithm and schemata learning modules

The CDR-PopGen results are used in LEGA to generate the initial population for the EA module. The characteristics of the initial population will determine the diversity of promising subspaces in our search for optimality. Therefore, to increase the diversity of the first generation, we combine the results of the CDR-PopGen algorithm with randomly generated chromosomes via random permutation of operations and random assignment of machines to each operation. Our tests on different proportions of these combined schedules found 0.5 to be a suitable ratio. This ratio was used throughout the experiments reported in Section 5. The results from the CDR-PopGen algorithm are also selected to train the SL module upon system initialization. The promising machines that process operations to obtain good schedules are extracted to update the operational memory while good schedules are chosen to update the chromosomal memory. A detailed structure of the SL module and how to update it will be described in Section 4.4.

### 4.4. Schemata learning module

Based on the OOMS chromosome, we now describe how to update the chromosomal memory and the operational memory (as seen in Fig. 3). An instantiation of the LEGA architecture named GENACE is given in Fig. 9. The communication between the SL module and the EA module is shown by Step (2), Step (4) and Step (5). First of all, the CDR-PopGen algorithm is used to generate an initial population (the process that encodes from a CDR-PopGen result to an OOMS).

In each generation, chromosomes in the chromosomal memory that are *similar* to the best chromosomes will survive to the next generation. After performing crossover, the knowledge in the operational memory will influence the mutation operator towards improved results. To reduce computational time and to avoid reusing the same good schedules, the SL module is updated only after every $q$th step using an elitist strategy. The structure of the SL module of LEGA is described in the following subsections.

#### 4.4.1. Chromosomal memory

The *schemata theory* [20] implies that building blocks are used to create good chromosomes which derive from bit patterns in the schemata. The schemata of a binary code 11*10* can instantiate four chromosomes; 11**0**10**0**, 11**0**10**1**, 11**1**10**0** and 11**1**10**1**. We use a similarity template when performing crossover on these chromosomes; as a result, chromosomes with enduring characteristics of higher fitness are more likely to survive into the next generation. In order to inherit the similarity template of these chromosomes and influence the current population; a predefined number of good chromosomes in the current generation are selected. Each selected chromosome is then compared to the

```
Step 0) Initialize:
       a) Generate the initial population: ½ individuals by CDRs and ½ individuals by
          random.
       b) Initialize Chromosomal Memory by m best individuals in the initial
          population.
       c) Initialize Operational Memory by the machines that process an operation in
          the shortest time.
Step 1) Selection: Apply Stochastic Universal Sampling method to the current
        population.
Step 2) Influence the population by Chromosomal Memory: select n best chromosomes from
       current population. Each of them is compared to those in the Chromosomal Memory
       to find k similar chromosomes. The procedure to compute similarity between two
       chromosomes and those in the Chromosomal Memory will be described in Section
       4.4.1. Then these k chromosomes are inserted to the current population by
       replacing the k worse ones.
Step 3) Crossover: apply crossover operator to the population.
Step 4) Mutation: influence mutation operator by Operational Memory.
Step 5) Update: update Chromosomal Memory and Operational Memory after every q^th step.
Step 6) Stop condition: the algorithm terminates if it exceeds number of generations.
       Otherwise, go to Step 1).
```

Fig. 9. GENACE: an instantiation of the LEGA architecture.

$k$-nearest neighbor chromosomes in memory. Two chromosomes (when $k$ equals to 2) with the highest similarities in the memory are then copied to the current generation. Hence, the chromosomal memory is constructed with a list of *m good chromosomes* from the initial population. It will be updated by *m good chromosomes* from the population after every $q$th step.

We define the similarity between two chromosomes based on the difference in their operations. In this respect, the similarity between two schedules depends on the *genotype* of the chromosome, not the *phenotype* of the chromosome. The *similarity* can be calculated by following function:

$$S(Chm1, Chm2) = \sum_{i=1}^{n} Opt1(i) \otimes Opt2(i) + \sum_{j=1}^{m} Mch1(j) \otimes Mch2(j),$$

where *Chm1* and *Chm2* are two chromosomes, *Opt1* and *Opt2* are operation order parts of *Chm1* and *Chm2*, $Opt1(i)$, $Opt2(i)$ and $n$ represents are the values at position $i$ and length of operation order part, respectively. Similarly, *Mch1* and *Mch2* are machine order parts of *Chm1* and *Chm2*, $Mch1(j)$, $Mch2(j)$ and $m$ are the values at position $j$ and length of machine order part. The operator $\otimes$ returns 0 if two values are equal, otherwise it returns 1. The value of $S$ is the *Hamming distance*, the number of values that are different. If the value of $S$ is small, two chromosomes *Chm1* and *Chm2* are close. This measure takes $O(n + m)$ computational time. In order to save computational time, in Section 5, we set the number of chromosomes in chromosomal memory to 5, the number of the best chromosomes in the population to compare to the memory to 3.

#### 4.4.2. Operational Memory

We saw how the CDR-PopGen algorithm finds a good schedule by selecting the most suitable machine that can process an operation. In order to improve the mutation operation to achieve this, we control it by using the operational memory. This mutation operator helps to guide the search process towards the promising schedules. The integration between the mutation operator and the operational memory will be described in detail in Section 4.4.5. The operational memory contains a set of possible machines to process an operation of a job and therefore constrains the mutation operator to create a better chromosome. The structure of the operational memory is an array of bits. In Step (0) (c) in Fig. 9, the machines that process an operation with the shortest processing times are selected. Their corresponding positions with these operations in the operational memory are then set to 1. The remaining positions in the operational memory are set to 0. The other suitable machines to process an operation can be updated after the $q$th step by a set of $n$ best chromosomes ($q$ and $n$ are two predefined parameters). Fig. 10 shows the construction of an operational memory for the example given in Table 1. A machine's suitability for processing the corresponding operation is denoted by a value of 1 and a 0 for the contrary. Consider operation $O_{1,2}$. It

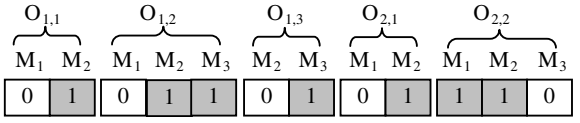


Fig. 10. An example of a schemata learning module's operational memory.

can be processed on machine $M_1$ within 9 unit times, machine $M_2$ within 2 unit times or machine $M_3$ within 2 unit times. The unit values at column $M_2$ and $M_3$ indicate that $M_2$ (2 unit times) or $M_3$ (2 unit times) can be selected to process $O_{1,2}$ to create a shorter schedule rather than select $M_1$ (9 unit times). The other suitable machines are updated after $q$th step by setting its suitability value to 1. Therefore, in later generations, an operation will have more suitable machines for selection.

*4.4.3. Selection operator*

After decoding all chromosomes in the current population to obtain fitness values, linear scaling and stochastic universal sampling are used to select efficient individuals [20].

*4.4.4. Crossover operator*

As described in Section 4.3, the representation of the chromosome has two parts. Therefore, crossover is applied on each part of the chromosome.

*Operation order part*: two-point crossover is applied [29]. To explain its operation, consider two parents: (2 1 2 1 1) and (1 1 1 2 2). A substring is selected randomly from parent 1: (2 1 2 1 1). Reading the string from left to right, we know that operations in the selected substring are the first operation of job $J_1$ ($O_{1,1}$), the second operation of job $J_2$ ($O_{2,2}$), and the second operation of job $J_1$ ($O_{1,2}$). The corresponding positions of the characters in this string are then found and deleted in the second parent: (1 1 1 2 2). The substring is inserted to the second parent at the same position in the first parent to create a new child: (1 1 2 1 2).

*Machine selection part*: two random numbers (for the two loci) are selected: $2 \leqslant r_1 \leqslant r_2 \leqslant (n-1)$, ($n$ is the length of machine selection part). Two partial parts of the parents between the two loci are exchanged. For example, consider two machine selection parts of two parent chromosomes in Fig. 11.

If two random numbers $r_1 = 2$ and $r_2 = 4$ are generated, then the two parts of two parents between position 2 and position 4 are exchanged (see Fig. 12).

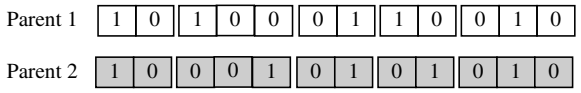


Fig. 11. An example of two machine selection parts of two parents.

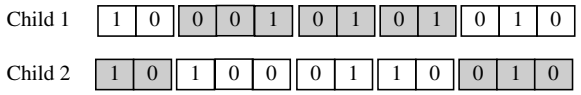


Fig. 12. Two machine selection parts of two offspring.

*4.4.5. Mutation operator*

Similar to crossover, mutation is also performed on two parts of a chromosome.

*Operation order*: two random numbers $r_1$ and $r_2$ are selected such that $2 \leqslant r_1 \leqslant r_2 \leqslant (m-1)$, where $m$ is the length of the operation order part. The values in substring in between two positions are inverted. For instance, consider an operation part of a chromosome: (1 2 2 1 1). Two random numbers are generated: $r_1 = 2$ and $r_2 = 4$. The substring between position 2 and 4, (1 2 2 1 1), are converted. The result is (1 1 2 2 1).

*Machine selection*: the mutation operator is influenced by the operational memory according to the algorithm given in Fig. 13.

Step (1) (b) in Fig. 13 shows how to find a *suitable* machine to process an operation. If there are more than one machine suitable to process operation $O_{i,j}$, the mutation operator would select a machine that is different from the current machine. This increases the chances of finding better results by exploiting other promising parts of the search space. To illustrate this process, consider an example of an operational memory of the current generation and the *machine selection part* of a chromosome as given in Fig. 14.

Consider the operation $O_{1,1}$ processed by machine $M_1$. From the operational memory, this operation can be processed by machine $M_1$ or machine $M_2$. Therefore, machine $M_2$ is selected by mutation operator to process this operation. Applying similar step to $O_{1,2}$, $O_{1,3}$, $O_{2,1}$, $O_{2,2}$, we obtain the result after applying mutation in Fig. 14. The above algorithm takes O($n$) where $n$ is total number of operations. An advantage of our proposed chromosomal representation is that it always produces feasible offsprings when performing crossover and mutation operations, thereby removing the need for costly repairs.

```
Step 0) For each operation O_{i,j} in machine selection part of a chromosome:
     a) Identify machine M_k that is currently selected to process O_{i,j}.
     b) Identify a set of machines that can process O_{i,j}: P(O_{i,j}).
     c) Identify a set of suitable machines in Operational Memory that can process
        O_{i,j}: P_M(O_{i,j}).
Step 1) Generate a random number r ∈ [0,1]:
     a) If r is equal or smaller than 0.5, apply random mutation operator: select a
        random machine in P(O_{i,j}) (except M_k) to process O_{i,j}.
     b) Otherwise, influence mutation operator: if P_M(O_{i,j}) contains only M_k, then
        still keep M_k to process O_{i,j}. Otherwise, select a random machine in P_M(O_{i,j})
        (except M_k) to process O_{i,j}.
Step 2) Go to Step 0).
```

Fig. 13. Influence of operational memory on mutation operator.

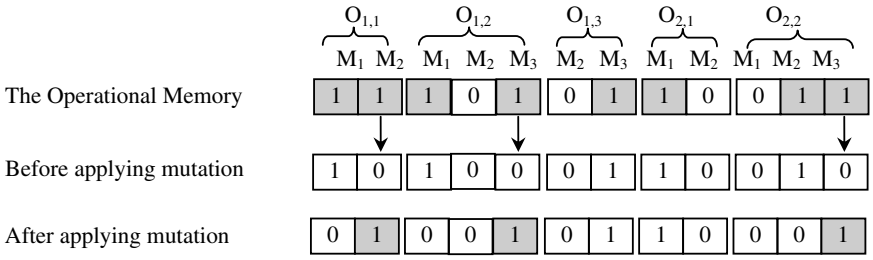


Fig. 14. An example of influence of operational memory on mutation operator.

## 5. Experimental results

Two sets of sample problems are used to analyze this instantiation. Test sample I includes both T-FJSP instances taken from [13,15,16] and randomly generated T-FJSPs. Test sample II includes both the P-FJSP instances taken from [17] and randomly generated P-FJSPs. The dimensions of the problems presented in literature typically range from 4 jobs × 5 machines to 20 jobs × 15 machines. In addition, we examine the current instantiation in solving large randomly generated FJSPs; ranging from 40 jobs × 15 machines to 200 jobs × 20 machines for each test sample. The number of operations for each job ranges from 20 to 40. The processing time for each operation ranges from 10 to 100 unit times. In the P-FJSP random instances in Test sample II, less than 50% machines are randomly selected to process an operation. Furthermore, in practice, the difference in processing times between two machines that can process an operation is small due to similar configurations of machines that belong to the same group. Therefore, we set the maximum deviation of two machines to process one operation to be 5 unit times. As mentioned earlier, although partial flexibility makes the problem more difficult, one approach is to convert it to the T-FJSP [15]. This can sometimes lead to larger time and space complexities, however, in our experiments, the performance for solving the P-FJSPs is not affected because the chromosomal representation for partial flexibility always matches the P-FJSPs. The current instantiation was implemented in C++ on a Pentium IV running at 2 GHz and experiments described below were the best and average results obtained after 50 runs. The average computational times after 50 runs were also reported.

Through experimentation, the parameter values were chosen as follows: population size: 200, crossover probability 0.75, mutation probability 0.3, number of generations 200, number of generations to perform learning 5 ($q = 5$), number of best parents copies to next generation 3, number of chromosomes in chromosomal memory 5, number of chromosomes in the population to update chromosomal memory 5, number of chromosomes in the population to compare to chromosomal memory 3, number of chromosome to update operational memory 10. Some of these values are noticeably different (by an order of magnitude) from those usually chosen for binary encoded GAs; such as the mutation probability.

### 5.1. Test sample I

The first test sample comes from [13]. The problem is a 10 jobs by 10 machines T-FJSP. The current instantiation obtains the optimal result of 7 time units after 20 generations (or 0.11 seconds). Mesghouni et al. [13] also obtained this optimum result but only after 1500 generations. The reason could be due to the complexity of decoding their chromosome representation. It requires a repair mechanism to recalculate the starting times for all operations when a new chromosome is created. In our algorithm, the new chromosomes are always feasible after applying these operators. Furthermore, the search process is guided by the learning module towards optimal results. Therefore, our processing time should be shorter. Fig. 15 shows the fitness distribution of all 200 generations. To emphasize the fitness differences, only 50% of the best individuals from each generation are selected to be visualized. The colormapped and sorted fitness values (or *makespan* values) are drawn along the vertical axis. Darker colors indicate smaller fitness values. The figure shows that the best-fit individuals increase as part of the population as soon as generation 20. As mentioned in Section 4.4, the SL module is used to enhance the population of EA by updating the similar templates of the best individuals. In order to see how the good templates can influence the fitness values of the schedules, the individuals in the last generation are drawn in Fig. 16.

The fitness values of the individuals in Fig. 16 are sorted in descending order. Each individual is drawn along the horizontal axis and each gene is assigned a gray scale. Darker color indicates smaller

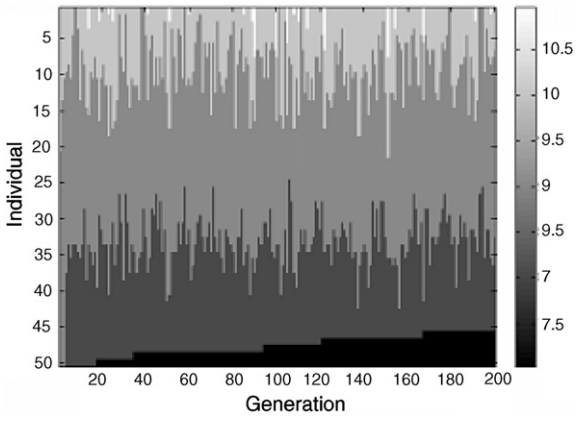


Fig. 15. Fitness distribution of 10 jobs × 10 machines T-FJSP (50% best).

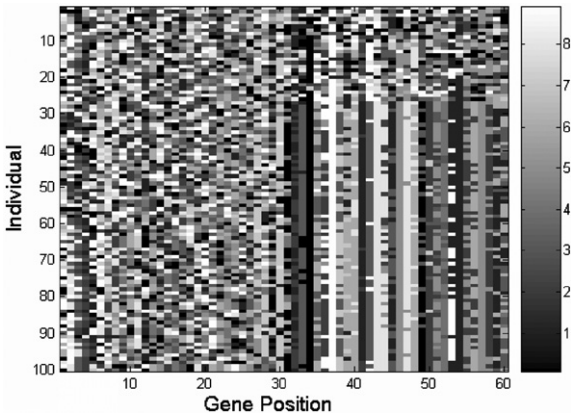


Fig. 16. Individuals of 10 jobs × 10 machines FJSP at generation 200.

gene value. The figure shows that the best individuals contain similar blocks after gene 30. These positions visualize the machine selection part of the individuals and indicate that good schedules contain the same machine selection for each operation. This result verifies the importance of assigning a suitable machine to process an operation for solving FJSP problems. The remaining genes of the individuals from position 1 to position 30 that represents the operation order parts of the individuals are not similar. However, the fitness values of the individuals are still good. This can be explained as follows. In the classical JSP and the FJSP, the order of operations on each machine determines the fitness value of each schedule. As described in Section 4.3, our proposed representation has two parts: operation order part and machine selection part. In the operation order part, the index of job is used to represent the individual. It is decoded by reading the operation order part from left to right. Therefore, the different operation order parts of individuals can be decoded to the same result if the order of operations on each machine after decoding remains unchanged. Table 4 shows the comparison between the results of applying Kacem et al. [15] and GENACE.

The dimensions of the problems range from 4 jobs by 5 machines to 15 jobs by 10 machines. The first column is the number of jobs and the number of machines of each instance; the second column shows the results from [15]; the next three columns give the best, average results and standard deviation of applying the corresponding algorithms to solve instances in the first column. The sixth column gives the relative improvement between the best result of

Table 4
Makespan of T-FJSP instances from Kacem et al. [15]

| Job × Mach | Kacem et al. [15] | Best | Avg. | SD | R.Improvement (%) | Avg. time (second) |
|---|---|---|---|---|---|---|
| 4 × 5 | 16 | 11 | 11.00 | 0.0 | 31.25 | 1.226 |
| 10 × 7 | 15 | 11 | 11.00 | 0.0 | 26.66 | 2.610 |
| 10 × 10 | 7 | 7 | 7.56 | 0.501 | 0 | 2.803 |
| 15 × 10 | 23 | 12 | 12.04 | 0.197 | 47.82 | 4.948 |

[15] and GENACE for each instance (((2nd column − 3rd column)/2nd column) × 100%). The observation from Table 3 is that the quality of GENACE is sufficient for solving some small-sized T-FJSPs. The results also indicate that our algorithm outperforms the algorithms of [15].

Unfortunately, we are not aware currently of large T-FJSP benchmarks in literature. Hence, to test for large problem instances, we randomly generated 6 large T-FJSP instances shown in Table 5.

Table 5 shows the best results of EA alone and the combination of three modules: EA, CDR, and SL in 50 runs. Similar to that in Table 2, the average values of the best results of EA + CDR + SL in 50 runs for each instance are reported. Furthermore, standard deviation values of the best results in 50 runs are also given. It is observed that the results obtained by EA and CDR are better the results obtained by EA and SL. The combination of three modules gives the best ones. The reason can be explained by the efficiency of CDR, it is used to improve the results of GENACE by generating a more diverse initial population. Additionally, the SL module also helps to find the suitable assignments of operations to machines.

### 5.2. Test sample II

For 8 jobs by 8 machines P-FJSP from [15]. The current instantiation quickly obtains the result of 14 unit times after only 28 generations (or 0.3 seconds). This result is equal to the result of [15]. Similar to the test sample I, Figs. 17 and 18 illustrate the fitness distribution and individuals at generation 200 of the 8 jobs by 8 machines P-FJSP, respectively. They indicate an increase in the number of new best-fit individuals after 20 generations and that the good individuals are composed of the same building blocks.

Table 6 gives the results of applying GENACE to solve the 9 P-FJSP problems taken from Brandimarte [17]. The dimensions of the problems in Table 6 range from 10 jobs by 6 machines to 20 jobs by 15 machines. The first and second columns give the problem specifications; the third column shows

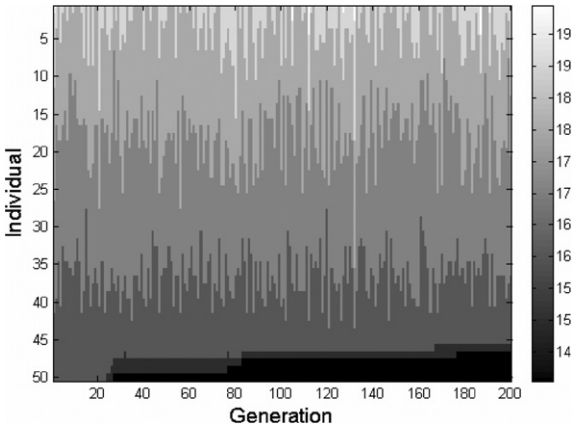


Fig. 17. Individuals of 8 jobs × 8 machines FJSP at generation 200.

Table 5
Makespan of large T-FJSP instances

| Job × Mach | EA best | EA + SL best | EA + CDR best | EA + CDR + SL best | EA + CDR + SL avg. | EA + CDR + SL SD | Avg. time (second) |
|---|---|---|---|---|---|---|---|
| 40 × 15 | 5652 | 5499 | 4606 | 4559 | 4585.12 | 11.98 | 37.304 |
| 60 × 15 | 7960 | 7912 | 7099 | 7045 | 7075.16 | 12.69 | 109.35 |
| 80 × 15 | 9732 | 9635 | 8845 | 8796 | 8818.62 | 12.13 | 174.57 |
| 100 × 20 | 9455 | 9373 | 8532 | 8463 | 8498.60 | 14.98 | 257.94 |
| 150 × 20 | 13242 | 13135 | 12400 | 12336 | 12356.66 | 12.30 | 602.89 |
| 200 × 20 | 17369 | 17274 | 16511 | 16474 | 16494.64 | 8.50 | 1110.33 |

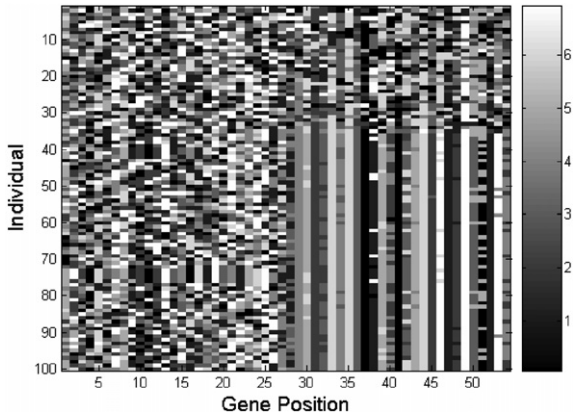


Fig. 18. Fitness distribution of 8 jobs × 8 machines P-FJSP (50% best).

the results from [17]; the next three columns give the best, average results and standard deviation of applying corresponding algorithms; the next column gives the relative improvement of [17] and GENACE (((3rd column − 4th column)/3rd column) × 100%) and the last one shows the average computational time. The test examples of P-FJSP seem to be more difficult than the test example of T-FJSP because P-FJSP is less flexible. The observation from Table 6 is that the relative improvement is small compared to the results in test sample I. The results indicate that GENACE can also obtain good results and it outperforms Brandimarte's approach in 8 out of 9 problems. In the remaining one, GENACE obtains the same result with Brandimarte's algorithm.

In order to test larger P-FJSP instances, we generated 6 different P-FJSP problems at random. Table 7 shows the results of applying GENACE to solve them.

Similar to the results of solving the large T-FJSP problems, the results of applying different types of combination of three modules in LEGA are reported. The results of EA and CDR are better than the results of combining EA and SLM. The combination of three modules also gives the best results. However, because of fewer machine availabilities in the P-FJSP, relative improvements between them are small compared to the T-FJSP benchmarks in test sample I.

## 6. Conclusions

The empirical results in Section 5 show that GENACE is more efficient than the other approaches for

Table 6
Makespan of P-FJSP instances from Brandimarte [17]

| Name | Job × Mach | Brandimarte [17] | Best | Avg. | SD | R.improvement (%) | Avg. time (second) |
|---|---|---|---|---|---|---|---|
| Mk1 | 10 × 6 | 42 | 40 | 41.5 | 0.543 | 4.67 | 3.328 |
| Mk2 | 10 × 6 | 32 | 29 | 29.1 | 0.303 | 9.3 | 3.556 |
| Mk4 | 15 × 8 | 81 | 67 | 67.34 | 0.478 | 17.28 | 6.068 |
| Mk5 | 15 × 4 | 186 | 176 | 178.1 | 1.164 | 5.37 | 7.173 |
| Mk6 | 10 × 15 | 86 | 67 | 68.82 | 0.849 | 22.09 | 10.698 |
| Mk7 | 20 × 5 | 157 | 147 | 152.9 | 1.843 | 6.36 | 6.893 |
| Mk8 | 20 × 10 | 523 | 523 | 523.34 | 0.871 | 0 | 19.098 |
| Mk9 | 20 × 10 | 369 | 320 | 327.74 | 3.498 | 13.28 | 20.413 |
| Mk10 | 20 × 15 | 296 | 229 | 235.72 | 2.927 | 22.63 | 27.697 |

Table 7
Makespan of large P-FJSP instances

| Job × Mach | EA best | EA + SL best | EA + CDR best | EA + CDR + SL best | EA + CDR + SL avg. | EA + CDR + SL SD | Avg. time (second) |
|---|---|---|---|---|---|---|---|
| 40 × 15 | 5261 | 5232 | 4588 | 4480 | 4561.52 | 17.70 | 33.90 |
| 60 × 15 | 7724 | 7675 | 6824 | 6778 | 6806.56 | 9.70 | 103.94 |
| 80 × 15 | 9878 | 9834 | 9079 | 9001 | 9043.5 | 18.44 | 167.89 |
| 100 × 20 | 9131 | 9076 | 8238 | 8176 | 8210.7 | 12.59 | 234.18 |
| 150 × 20 | 13351 | 13270 | 12571 | 12522 | 12542.28 | 10.13 | 595.05 |
| 200 × 20 | 17888 | 17826 | 17170 | 17109 | 17133.1 | 15.14 | 1272.41 |

solving the T-FJSPs and the P-FJSPs compared with GA approaches by Kacem et al. [15,16], Mesghouni et al. [13], and Tabu Search approach by Brandimarte [17]. Some conclusions are drawn as follows:

- The higher the flexibility (i.e. the number of equivalent machines available to process an operation), the better the results. In our LEGA architecture, the SL module can help to find a suitable machine to process an operation. If the set of these machines is large, more selections and better assignments can be made. Therefore, better results can be explored.
- The performance of GENACE with a randomized initial population is poorer than GENACE with an initial population partly created by the CDR-PopGen module. By combining diverse individuals with fit ones generated by the CDR-PopGen module, GENACE is able to get better results.
- The CDR-PopGen module is efficient in solving the small FJSP instances. The experimental results in Section 4.2 indicate that the CDR-PopGen can achieve better results than the other approaches in almost all small problem instances.
- The chromosomal representation in GENACE is able to express the FJSP in general form. It does not require transforming the P-FJSP to the T-FJSP as with the GA approaches of [15]. It also does not require any repair mechanisms to maintain feasibility, such as in [13] or [14]. Therefore, the computational time for solving P-FJSPs can be reduced.

In this paper, the LEGA architecture that integrates an EA, a SL module and a population generator is proposed. Past experience from previous generations of schedules are examined, sampled and stored in the SL module to influence the current generation to obtain better schedules. The CDR-PopGen algorithm was proposed to perform initial population generation so as to solve FJSPs. It assigns an operation to a suitable machine to reduce the workload on each machine and minimize the completion time of all operations. When applying these rules to solve some benchmark problems, in some situations, these rules can obtain near optimal solutions. To incorporate a learning mechanism, a new chromosomal representation is introduced. Reviewed representations typically consume a large amount of computing time for checking constraints when performing crossover or mutation. The efficient coding of our proposed chromosomal representation not only satisfies the different constraints of the FJSP, it also always creates feasible and active schedules after genetic recombination, and is adapted to learn good properties of the best chromosomes. Experimental results show that the GENACE instantiation of the LEGA architecture obtains better upper bounds for 11 out of 13 benchmark FJSP problems, with improvement factors of 2–48%. It validates the 'no-free lunch theorem' [30] that knowledge of the problem domain can help the search process to get better results.

In GENACE, the SL module is simply a memory keeping good features from previous generations. Therefore, we intend to focus on enhancing the LEGA architecture via explorations in machine learning (such as truth maintenance system [31] or AQ learner [32]) so as to improve the efficiency of the SL module. Improvements in the population generator module will include an investigation into how the rules can be automatically designed (via genetic programming) to cater to specific properties of the problems in question. In this paper, the single objective FJSP was discussed. The extension to multi-objective FJSP will be investigated in the near future.

## Acknowledgments

This research was funded in part by Nanyang Technological University and CEI Contract Manufacturing Limited, Singapore.

## References

[1] A.S. Jain, S. Meeran, Deterministic job-shop scheduling: Past, present and future, European Journal of Operation Research 113 (2) (1998) 390–434.
[2] J. Carlier, E. Pinson, An algorithm for solving the job-shop problem, Management Science 35 (2) (1999) 164–176.
[3] M. Kolonko, Some new results on simulated annealing applied to the job shop scheduling problem, European Journal of Operational Research 113 (1999) 123–136.
[4] E. Nowicki, C. Smutnicki, A fast Taboo Search Algorithm for the job shop problem, Management Science 42 (6) (1996) 797–813.
[5] T. Yamada, R. Nakano, A fusion of crossover and local search, in: Proceedings of the IEEE International Conference on Industrial Technology, 1996, pp. 426–430.
[6] M.R. Garey, D.S. Johnson, R. Sethi, The complexity of flow shop and job-shop scheduling, Mathematics of Operations Research 1 (2) (1996) 117–129.
[7] M. Pinedo, X. Chao, Operations Scheduling with Applications in Manufacturing and Services, McGraw-Hill, New York, 1999 (Chapter 3).

[8] R.G. Reynolds, An introduction to Cultural Algorithms, in: Proceedings of the Third Annual Conference on Evolutionary Programming, River Edge, NJ, World Scientific, Singapore, 1994, pp. 131–139.
[9] C.J. Chung, R.G. Reynolds, A testbed for solving optimization problems using cultural algorithm, in: Proceedings of the Fifth Annual Conference on Evolutionary Programming, 1996, pp. 225–236.
[10] J. Branke, Memory-enhanced evolutionary algorithms for dynamic optimization problems, in: Proceedings of Congress on Evolutionary Computation, CEC99, 1999, pp. 1875–1882.
[11] S.J. Louis, J. McDonnell, Learning with case-injected Genetic Algorithms, IEEE Transactions on Evolutionary Computation 8 (4) (2004).
[12] R.S. Michalski, Learnable Evolution Model: Evolution process guided by Machine Learning, Machine Learning 38 (2000) 9–40.
[13] K. Mesghouni, S. Hammadi, P. Borne, Evolution programs for job-shop scheduling, in: Proceedings of the IEEE International Conference on Computational Cybernetics and Simulation, vol. 1, 1997, pp. 720–725.
[14] H. Chen, J. Ihlow, C. Lehmann, A genetic algorithm for flexible job-shop scheduling, in: Proceedings of the IEEE International Conference on Robotics and Automation, vol. 2, 1999, pp. 1120–1125.
[15] I. Kacem, S. Hammadi, P. Borne, Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems, IEEE Transactions on Systems, Man and Cybernetics 32 (1) (2002) 1–13.
[16] I. Kacem, S. Hammadi, P. Borne, Pareto-optimality approach for flexible job-shop scheduling problems: Hybridization of evolutionary algorithms and fuzzy logic, Mathematics and Computers in Simulation 60 (2002) 245–276.
[17] P. Brandimarte, Routing and scheduling in a flexible job shop by tabu search, Annals of Operations Research 22 (1993) 158–183.
[18] M. Pinedo, Scheduling Theory, Algorithms, and Systems, Prentice-Hall, Englewood Cliffs, NJ, 2002 (Chapter 2).
[19] N.B. Ho, J.C. Tay, The model of job assignments in Electronic Manufacturing Service in CEI Contract Manufacturing Company, Technical Report, 2002.
[20] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addition-Wesley, Reading, MA, 1989.
[21] N.B. Ho, J.C. Tay, GENACE: An efficient cultural algorithm for solving the flexible job-shop problem, in: Proceedings of the Congress on Evolutionary Computation CEC2004, 2004, pp. 1759–1766.
[22] J.C. Tay, D. Wibowo, An effective chromosome representation for evolving flexible job-shop schedules, in: Proceedings of the Genetic and Evolutionary Computation GECCO2004, 2004, pp. 210–221.
[23] H. Oliver, R. Chandrasekharan, Efficient dispatching rules for scheduling in a job shop, International Journal of Production Economics 48 (1) (1997) 87–105.
[24] S. Panwalkar, I. Wafik, A survey of scheduling rules, Operations Research 25 (1) (1997) 45–61.
[25] Ph. Fortemps, Ch. Ost, M. Pirlot, J. Teghem, D. Tuyttens, Using metaheuristics for solving a production scheduling problem in a chemical firm. A case study, International Journal of Production Economics 46 (1996) 13–26.
[26] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, Springer, New York, 2001 (Chapter 13).
[27] R. Cheng, M. Gen, Y. Tsujimura, A tutorial survey of job-shop scheduling problems using genetic algorithms, Part I: Representation, Computers and Industrial Engineering 30 (4) (1996) 983–997.
[28] Ch. Bierwirth, A generalized permutation approach to job shop scheduling with genetic algorithms, OR Spektrum 17 (1995) 87–92.
[29] R. Varela, C.R. Vela, J. Puente, A. Gómez, A knowledge-based evolutionary strategy for scheduling problems with bottlenecks, European Journal of Operational Research 145 (1) (2003) 57–71.
[30] D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 67–82.
[31] J. Doyle, A truth maintenance system, Artificial Intelligence 12 (1979) 231–272.
[32] R.S. Michalski, K. Kaufman, Learning patterns in noisy data: The AQ approach, in: G. Paliouras, V. Karkaletsis, Spyropoulos (Eds.), Machine Learning sand Applications, Springer, Berlin, 2001.