GENACE: An Efficient Cultural Algorithm for Solving the Flexible Job-Shop Problem

N. B. Ho and J. C. Tay
Intelligent Systems Laboratory
Nanyang Technological University
Blk N4 #2a-32 Nanyang Avenue, Singapore 639798
Email: asjctay@ntu.edu.sg

***Abstract*- This paper presents an efficient methodology called GENACE for solving the flexible job-shop scheduling problem (or FJSP) with recirculation. We show how CDRs are used to solve the FJSP with recirculation by themselves and to provide a bootstrapping mechanism to initialize GENACE. We then adopt a cultural evolutionary architecture to maintain knowledge of schemata and resource allocations learned over each generation. The belief spaces influence mutation and selection over a feasible chromosome representation. Experimental results show that GENACE obtains better upper bounds for 11 out of 13 benchmark problems, with improvement factors of 2 to 48 percent when compared to results by Karcem et al, Brandimarte and of using CDRs alone.**

Keywords- **Flexible Job-Shop, Genetic Algorithm, Cultural Evolution, Dispatching Rules.**

I. INTRODUCTION

The job-shop scheduling problem (or JSP) is an interesting scheduling problem that has been attracting many researchers [1, 2, 3]. In the JSP, there are n jobs and m machines, each job is to be processed on a group of machines satisfying precedence constraints. Each operation of a job is to be processed only on one predetermined machine. It is well known that the JSP is NP-hard [4]. Though the JSP has been well studied, its application to real-world scenarios is often undermined by the constraint of the one-to-one mapping of operations to machines. Hence, the *flexible* job-shop scheduling problem (or FJSP) extends the JSP by allowing each operation to be processed on more than one machine. With this extension, we are now confronted with two subtasks: assignment of each operation to an appropriate machine and sequencing operations on each machine. In addition, for complex manufacturing systems, a job can recirculate among several machines, thereby significantly increasing the complexity of the system requirements [5].

In this paper, we introduce a methodology called GENACE, which uses evolutionary computation to approximate the FJSP efficiently. The general approach is as follows: Composite Dispatching Rules (or CDRs) [6, 7] are used to create the initial chromosomal population. The search space in every generation is then intelligently guided by generalizations of the individual chromosomes that are *culturally evolved* from previous generations. The aim of this algorithm is to minimize the *makespan;* the maximum completion time of all operations. To examine the performance of our proposed method, some common benchmarks from the research field are used. Experimental results of comparing GENACE with [8, 9, 10, 11] show that GENACE obtains better upper bounds for 11 out of 13 benchmark FJSPs.

The paper is organized as follows. Section II gives the formulation of the FJSP *with* recirculation. Section III introduces recent research in solving the FJSP and some lessons learned. Section IV describes bootstrapping with CDRs while Section V presents a new chromosome representation used in GENACE for solving the FJSP with recirculation. Section VI describes the cultural evolution model of GENACE that is used for representing learned information from previous generations; in addition, how the belief-space model is used to adapt the genetic operators. Section VII presents and analyzes the performance results. Finally, Section VIII gives some concluding remarks and directions for future work.

II. PROBLEM FORMULATION

The FJSP with recirculation is formulated as follows:

- There are n jobs J=1, 2,..., n, indexed i.
- Each job consists of a sequence of operations G_i.
- There are m machines M=1, 2,... m, indexed k.
- Each operation, indexed j, can be processed on a set of machines M_{ij}. Therefore, O_{ijk} denotes that operation j of job i is to be processed on machine k.
- Recirculation occurs when a job can visit a machine more than once.
- The objective is to find a schedule that has minimum makespan.

Depending on the size of M_{ij}, the FJSP is classified into two kinds:

- Total FJSP (or T-FJSP): $M_{ij} = M$ for each operation O_{ij}, implies that each operation can be processed on any one machine of set M.
- Partial FJSP (or P-FJSP): $M_{ij} \subset M$ for at least one operation O_{ij}.

Kacem *et al.* [8] mentioned that the P-FJSP makes the problem more difficult by increasing the search space and the computation time. They divide their chromosome representations to address T-FJSP and P-FJSP. In this paper, we will also solve the two kinds of FJSP.

0-7803-8515-2/04/$20.00 ©2004 IEEE

However, the chromosome representation (to be described in Section V) in our algorithm is flexible. We will be able to use one chromosome representation to describe both problem solution types. Therefore, the performance results will also apply to both. We shall also 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.

III. Previous Works

The canonical GA often creates a number of infeasible chromosomes in each generation because it performs crossover and mutation on randomly selected genes. This leads to repairing each chromosome in order to ensure feasible schedules. Therefore, chromosome representation is important for successful application of the GA. If well designed, no infeasible solutions are produced after recombination, and the algorithm can expect to be more efficient. To evaluate the performance of a chromosome representation, we propose the following criteria:

- Time and space computational complexity of recombination operations.
- The need to maintain solution feasibility.

We will use these criteria to assess recent results from [8, 9, 12]. Mesghouni *et al.* [8] uses two types of chromosome representation for the FJSP: *parallel machine representation* and *parallel job representation*. In the parallel machine representation, a chromosome encodes a list of machines operating in parallel. Similarly, parallel job representation is a list of jobs executed in parallel. Figure 1 illustrates an example of parallel job representation.

Job 1	(M_1,0)	(M_3,4)	(M_3,6)
Job 2	(M_2,0)	(M_1,5)	

Figure 1. An example of a parallel job representation.

The chromosome representation of [9] allows crossover to be easily performed by using *row crossover* and *column crossover*. In row crossover, two chromosomes from the population and one job are randomly selected. The machines that process all operations of the job are unchanged. The remaining jobs with machines from the two parents are swapped. Similarly, column crossover keeps one column unchanged and swaps the remaining ones. However, it needs a repairing mechanism to recalculate the starting times for all operations. Due to the complexity of decoding the representation, this algorithm requires several generations to achieve near-optimal results.

Chen *et al.* [12] divide the chromosome into two parts: *A string* and *B string*. *A string* contains a list of all operations of all jobs and machines selected for corresponding operations while *B string* contains a list of operations that are processed on each machine.

A-string	O_{11}	O_{12}	…	O_{ij}	…	O_{nln}
	M_{O11}	M_{O12}	…	M_{Oij}	…	M_{On1n}

B-string	M_1	M_2	…	M_m
	O_{M1}	O_{M2}	…	O_{Mm}

The crossover and mutation operations are complex as the *B-string* has to be checked to ensure that operations in O_{Mk} consist only of those operations that are assigned to machine M_k by *A-string*. Furthermore, it also needs a repairing mechanism to maintain feasibility.

The best reported performance results so far belong to Kacem *et al.* [8]. Their representation is an assignment table. Figure 2 shows an assignment table of 2 jobs and 3 machines. An overview of their solution is as follows.

		M_1	M_2	M_3
J_1	O_{11}	**1**	0	0
	O_{12}	0	**1**	0
	O_{13}	0	**1**	
J_2	O_{21}	**1**	0	0
	O_{22}	0	0	**1**

Figure 2. An example of chromosome representation in [8].

Each unit value in the assignment table maps a machine to a corresponding operation. After using an assignment algorithm to find a set of feasible schedules (in the form of encoded assignment tables), they are used as an initial population for GA. The idea of the assignment algorithm is to assign and schedule each job one at a time. Dispatching rules [6, 7] are used here to resolve conflicts when more than one operation is scheduled on the same machine. Crossover is performed easily by keeping all the rows of one job and swapping rows of remaining jobs between two tables. Mutation is performed by selecting another machine that can process a chosen operation. All schedules that are generated after applying crossover and mutation are feasible. There is no need to repair chromosomes. 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 GENACE so that the knowledge is *culturally evolved* over each generation and further applied to the genetic operators (be described in Section V) 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.

IV. Bootstrapping GENACE with CDRs

We use *Composite* Dispatching Rules (or CDRs) to initialize GENACE. The CDR-Bootstrap algorithm is described in Figure 3.

```
1. Generate a random array of job orders J
2. Create a Gantt chart data structure G
3. Assign the first operation of each job
   to appropriate machines following the
   orders specified in J to G
4. Update number of running machines n
5. repeat
6.  j := the job with earliest finishing
          operation O_jk
7.  m := machine to process O_jk
8.  S := a set of machines that can
          process O_jk+1
9.  for (i := 1 to size_of(S) ) do
10.   wt   := waiting_time_on_machine_S(i)
11.   m_next  := machine with minimum wt
12. endfor
13. if (m_next is idle) then
14.   O_jk+1 is assigned to be processed on
        machine m_next in G
15.   n := n + 1
16. else
17.   Add O_jk+1 to waiting list of m_next
18. endif
19. if (operations on waiting list of
        machine m != ∅) then
20.     O_lm := next_processing_operation(m)
21. else
22.     n := n - 1
23. endif
24.until (n == 0)
25.return data structure G
```

Figure 3. The CDR-Bootstrap Algorithm in GENACE.

In the algorithm, O_{jk} denotes operation k of job j. `next_processing_operation(m)` selects the next operation on the waiting list of machine m depending on dispatching rules (e.g. SPT, LPT or FIFO). `waiting_time_on_machine_S(i)` simply sums the processing time of the next operation plus the remaining processing times on machine *i*. This algorithm assigns each operation to the machine that reduces the overall workload on each machine *and* minimizes the completion time of all operations. It terminates when all operations are scheduled or when all machines are idle.

The example in Table I has 2 jobs and 3 machines. Each operation of a job can be processed on more than one machine. The processing time on each machine is also predefined. For instance, operation O_{11} of job 1 can be processed on machine M_1 with processing time 4 units and on machine M_2 with processing time 5 units.

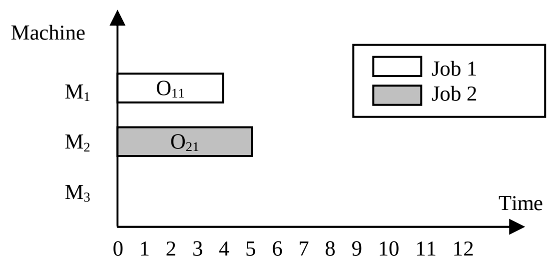
TABLE I. An Example of the P-FJSP

		M_1	M_2	M_3
J_1	O_{11}	4	5	XXX
	O_{12}	9	2	2
	O_{13}	XXX	6	3
J_2	O_{21}	6	5	XXX
	O_{22}	3	3	5

With reference to Figure 1:

Line 1: Creates a random order of jobs: for example, J_1 J_2.

Line 2: Assigns the first operation of the jobs to the array of the machines based on order: J_1 then J_2. This step is illustrated by the following Gantt chart.



Line 4: The number of running machines n is set to 2.

Line 6 to 8: The job with earliest finishing operation, $j = 1$ for machine $m = M_1$, the set of machines that can process O_{12}; $S = \{M_1, M_2, M_3\}$ is computed.

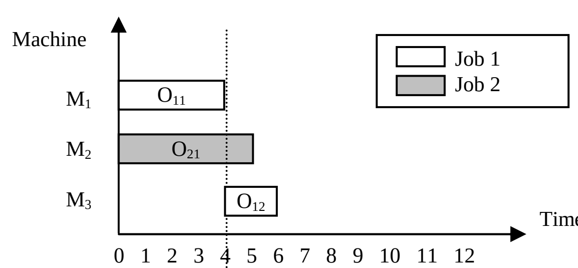
Line 9 to 12: The minimum waiting time of all possible machines $\{M_1, M_2, M_3\}$ to process operation O_{12} is found;

M_1: wt = ∑(processing times on waiting list of machine 1) + processing time of O_{12} on machine 1 + remaining processing time on machine 1 = 0 + 9 + 0 = 9.

M_2: wt = ∑(processing times on waiting list of machine 2)+ processing time of O_{12} on machine 2 + remaining processing time on machine 2 = 0 + 2 + (5 - 4) = 3.

M_3: wt = 2.

Minimum wt = 2 on M_3. Therefore, M_3 is selected to process O_{12} as shown in the following Gantt Chart.



Line 12 to 17: Selection of the next operation on machine 1's waiting list: since there are none, M_1 is idle and the number of running machines is updated to 3. The final result is shown in Figure 4 with a makespan of 9.

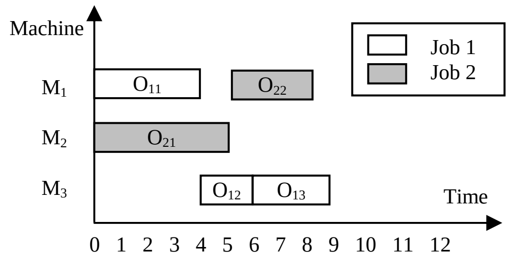


Figure 4. Final result of applying the CDR algorithm

Observe that the final result obtained depends on the orders of job and the rules that prioritize operations in the waiting lists (e.g. using FIFO, SPT, LPT). If we change the order of jobs and the rules, we will obtain different results. For instance, instead of J_1 J_2, by exchanging the order to J_2 J_1, the result will be 10 as shown in Figure 5.

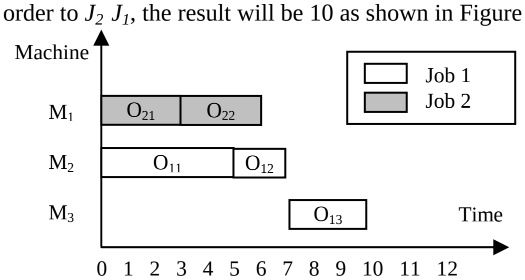


Figure 5. Alternative result by changing job orders

Consequently, we are able to create a set of random jobs orders and apply the CDRs to construct the initial population for bootstrapping GENACE.

V. GENACE ALGORITHM

A. Chromosome Representation

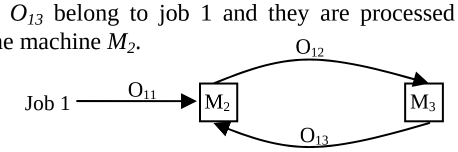
The canonical GA [13] often creates infeasible solutions during each generation after randomized crossover and mutation. Therefore, the effort to enforce feasibility must tradeoff against search efficiency. There are different kinds of chromosome representations for solving the FJSP. For instance, Mesghouni *et al.* [9] used *parallel machine representation* and *parallel job representation*, Chen *et al.* [12] use *A string* (operations) and *B string* (machines), while Kacem *et al.* [8] used *an assignment table*. The representations of [9] and [12] have to be adjusted for feasibility, while in the tabular representation of [8], rows present all operations of jobs and columns present all machines. The latter is not suitable for some P-FJSP in which an operation is only processed by a small subset of all machines. In this paper, we introduce an effective chromosomal encoding of the FJSP to ensure feasible solutions that are invariant under crossover and mutation. Theoretical and empirical perfomance results [14] have show favourable support. Another design objective is to facilitate the learning of good schemata in order to modify the cultural belief space (to be discussed in Section V.*C.* and Section V.*D.*). Our chromosome representation has two parts:

- Operation order part. We adopt the operation order representation from Ramiro et al. [15]. Consider the problem in Table I. Job 1 has 3 operations O_{11},O_{12},O_{13}; job 2 has 2 operations O_{21} ,O_{22}. One possible schedule could be (O_{21} O_{11} O_{22} O_{12} O_{13}), where O_{mn} is the operation of job *m* with order *n*. An individual is obtained from this schedule by replacing each operation by the corresponding job index; giving the result (2 1 2 1 1). This encoding prevents the creation of infeasible schedules.
- Machine selection part. We use an array of binary values to present machine selection. For the problem in Table I, one possible encoding of the machine selection part is shown in Figure 6.

O_{11}		O_{12}			O_{13}		O_{21}		O_{22}		
M_1	M_2	M_1	M_2	M_3	M_2	M_3	M_1	M_2	M_1	M_2	M_3
0	1	0	0	1	1	0	0	1	0	1	0

Figure 6. Machine selection part

A unit value denotes machine selection while 0 denotes non-selection. For example, M_2 is selected to process O_{11} because the value in the M_2 is selected. Only one machine can be selected per operation. For example, operation O_{12} can be processed on 3 machines M_1, M_2, M_3, so the valid values are 001, 010 or 100. This demonstrates a FJSP with recirculation if more than one operation of the same job is processed on the same machine. For instance, O_{11} and O_{13} belong to job 1 and they are processed on the same machine M_2.



B. Computing the makespan

Figure 7 gives the algorithm for computing the makespan from a given chromosome. A left-most time gap is detected between two operations processed on the same machine so as to insert a new operation where possible (line 9 to line 13). In the algorithm, *m* is the length of *operation order*, `operation_order(i)` returns the index of the job at position *i* in *operation order*, `operation_index(j)` returns the index of the operation of job *j*, `get_machine_index(j,k)` returns the index of the machine performing O_{jk}, `get_processing_time(j, k,l)` returns processing time of O_{jk} on machine *l* and `M` is a set of machines in the scheduling problem.

```
1.  for (i:=1 to m) do
2.    j := operation_order(i)
3.    k := operation_index(j)
4.    l := get_machine_index(j, k)
5.    p := get_processing_time(j, k, l)
6.    L := set of operations were
           processed on machine l
7.    middleList := false
8.    for (i:=1 to size_of(L)-1 ) do
9.      if (p <= the time between
          L(i) and L(i+1)) then
10.         insert operation k between
               L(i) and L(i+1)
11.         middleList := true
12.         break
13.     endif
14.   endfor
15.   if (!middleList) then
16.    insert operation k into the end of L
17.   endif
18. endfor
19. max:= maximum finishing time of a
           machine in a set of machine M
20. return max
```

Figure 7. Makespan Computation Algorithm

The algorithm is applied to the example in Figure 8. O_{11} finishes at time t and the next operation O_{12} is to be processed on M_1 for 3 time units. The Makespan Computation Algorithm can detect the left time gap between O_{11} and O_{23} (ie. 4 units). The left time gap is equal or greater than processing time of O_{12}. Therefore, O_{12} is inserted to this period. The result is in Figure 9.

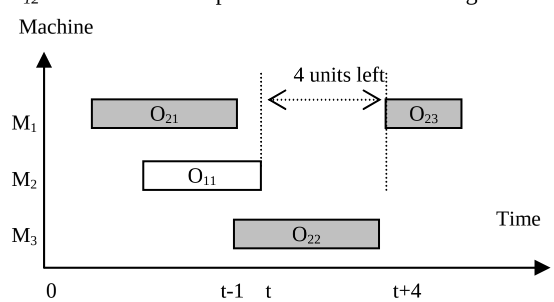


Figure 8. Insert O_{12} to machine M_1

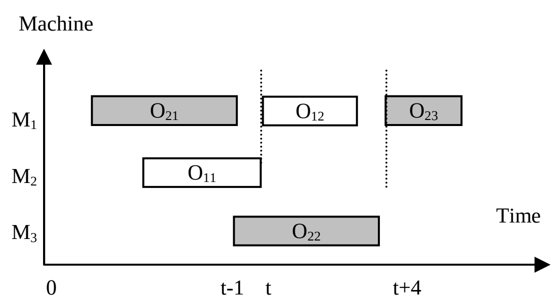


Figure 9. Result after inserting O_{12} to machine M_1

C. Learning from the problem domain

We use cultural evolution [16] to preserve beliefs of effective schedules that contain good schemata from each generation. In a cultural-based GA (or CGA), domain knowledge is used to represent, store and transmit knowledge from one generation to the next. It helps to reduce the search space by pruning useless parts and promote desirable parts. CGA models such 'cultural influences' through two spaces; *population space* and *belief space*. The population space is implemented by an extended GA, in that it uses belief-revised[1] operators of selection, crossover and mutation, while the belief space models the *behavioral traits* of the current population. The application of the belief-space model is similar to the memories of the Tabu Search [2] in which the good features of previous solutions are used to influence current solution. Based on the new representation of chromosome for the FJSP (as described in Section V.*A*.), we now present the GENACE method of integrating CDRs into a CGA.

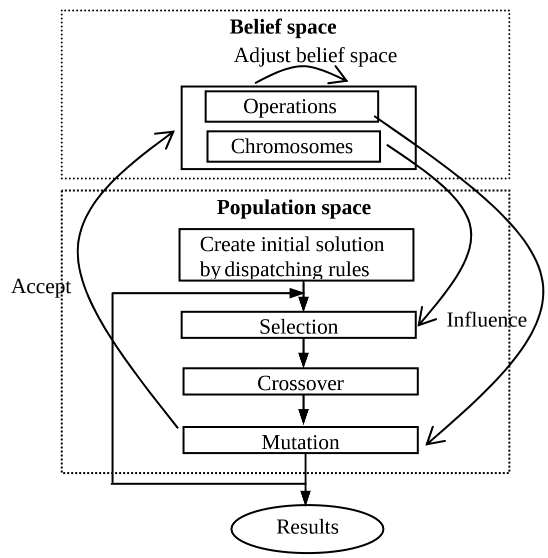


Figure 10. GENACE Model for Integrating CDRs into a CGA

Figure 10 gives the overall control-flow of GENACE. The belief space contains the possible schemata that can be produced over generalizations on population space. In our algorithm, we divide it into two parts: *operation belief space* and *chromosome belief space*. The Population space is implemented by an extended GA.

Good schemata are selected after a predefined number of generations and are used to modify the belief space so as to improve the search. The current state of the belief

[1] A simplified version of nogood subsumption [17].

space is then used to modify the performance of current individuals in the population by using belief revisioned selection and mutation operators.

Learning from machine allocations

We saw how the CDR-Bootstrap algorithm enables GENACE to find a good schedule by finding the most suitable machine that can process an operation. In order to improve the mutation operation to find a suitable machine, we control it by using an operation belief space. The operation belief space (or OB-Space) contains a set of possible machines to process an operation of a job. It must therefore represent a constraint that will help the mutation operator to create a fitter chromosome. The OB-Space is updated after k generations by a set of n best chromosomes in the k^{th} generation (k and n are two predefined parameters). The structure of the OB-Space is an array of bits. Figure 11 shows the construction of the OB-Space for the example in Table I.

O_{11}		O_{12}			O_{13}		O_{21}		O_{22}		
M_1	M_2	M_1	M_2	M_3	M_2	M_3	M_1	M_2	M_1	M_2	M_3
1	0	0	1	1	0	1	0	1	1	1	0

Figure 11. Example of an Operation Belief Space

Consider operation O_{12}. It 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_{12} to create a shorter schedule rather than select M_1 (9 unit times).

Learning from effective schedules

The *schemata theory* [18] implies that building blocks are used to create good chromosomes which derive from bit patterns in the schemata. Consider the schemata of a binary coding: 11*10*. Four chromosomes can be instantiated: 11**0**10**0** and 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 be reproduced into the next generation. Hence, a chromosome belief space (or CB-Space) is constructed with a list of m good chromosomes from the initial population (see Figure 12).

```
1. for (i:=1 to number of generations) do
2.      if ( (i mod k) == 0) then
3.        quicksort(current_population);
4.            update(current_population,
              chromosome_belief_space, n);
5.      end if
6. end for
```

Figure 12. CB-Space Updating Algorithm

The CB-Space is updated after k generations by a set of n good chromosomes from the current generation using an elitist strategy.

In order to inherit the similarity template of these chromosomes and influence the current population; after each generation, a predefined number of good chromosomes in current generation are selected. Then, each selected chromosome is compared with the chromosomes in CB-Space, two chromosomes with the highest similarities in CB-Space are copied to the current generation.

D. Designing the GA

In the GENACE algorithm, the representation of the chromosome has two parts. Therefore, crossover has to be performed on each part of the chromosome.

- Operation order: using 2-point crossover [15].
- Machine selection: two random numbers (for two loci) are selected: $2 \le r_1, r_2 \le (l - 1)$, ($l$ is the length of machine selection part). Two partial parts the parents between the two loci are exchanged.

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

- Operation order: two random numbers are selected: $2 \le r_1, r_2 \le (l - 1)$, ($l$ is the length of operation order part). The values in substring in between two positions are inverted.
- Machine selection: the mutation operator performs in this part is influenced by the OB-Space. The procedure of this influence is presented in Figure 13.

The new chromosome representation always produces feasible schedules when performing crossover and mutation operations. As there is no need to repair the schedules, overall computational costs can be reduced.

```
1. for ( i:=0 to  number of operations) do
2.  Read machine M that processes operation
      i in machine selection part;
3.  Refer to operation belief space
      to know possible machines can
      process this operation;
4.  if (there is one possible machine and
    this machine is equal to machine M) then
5.   Do not change machine that processes
       this operation;
6.  else
7.   Select randomly one of possible
        machines except machine M;
8.  end if
9. end for
```

Figure 13. Influence of OB-Space on Mutation

The characteristics of the initial population will determine the diversity of promising subspaces in our search for optimality. In GENACE, we maintain diversity of the first generation by creating a half of it randomly and the remaining half by CDRs.

VI. Emprical Results

We apply GENACE to solve some benchmark problems presented in literature. Two sets of sample problems are used: test sample 1 is the T-FJSP instances taken from [8, 9, 10] and test sample 2 is the P-FJSP instances taken from [8, 11]. Though partial flexibility makes the problem more difficult [8], the performance of GENACE is not affected because the chromosome representation for partial flexibility and total flexibility are the same. The GENACE algorithm was implemented in C++ on a Pentium IV running at 2 GHz and experiments described below were the best results selected after 5 runs.

Through experimentation, the parameter values are chosen as follows: population size: 100, crossover probability 0.75, mutation probability 0.3, number of generations 200, chromosome belief space size 10, number of generations to perform learning 5, number of chromosomes in current generation that influence the chromosome belief space 5, number of chromosomes in current generation that influence the operation belief space 10.

A. Test Results I

Figure 14 below shows the result when applying the GENACE algorithm for solving the 10 jobs by 10 machines T-FJSP taken from [8, 9]. It obtains the optimal result of 7 time units after 20 generations (or 0.11 seconds). This result was also obtained by Mesghouni *et al.* [9], but only after 1500 generations. After 20 generations, the mean makespan reaches equilibrium.

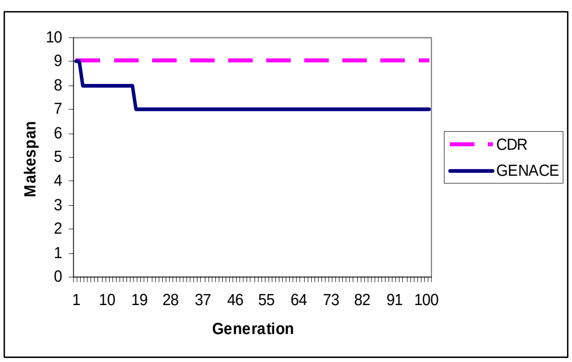


Figure 14. Results of 10 jobs by 10 machines T-FJSP

Table II shows the comparison between the results of applying CDRs alone, GENACE and Kacem *et al.* [10]. The first column is the number of jobs and the number of machines of each instance; the next four columns give the results of applying the corresponding algorithms to solve instances in first column and the last column is the deviations between [10] and the GENACE algorithm (((2nd column – 4th column)/ 2nd column) * 100%). The results indicate that our algorithm outperforms the algorithms of [10] and those of applying CDRs alone.

TABLE II. MAKESPAN OF 4 T-FJSP INSTANCES

JobxMach	Kacem *et al.* [10]	CDR	GENACE	Improve
4x5	16	11	11	31.25%
10x7	15	13	12	20%
10x10	7	9	7	0%
15x10	23	12	12	47.82%

B. Test Results II

Figure 15 below shows the result when applying the GENACE algorithm for solving the 8 jobs by 8 machines P-FJSP from [8]. It gains the optimal result 14 unit times after 28 generations (or 0.3 seconds).

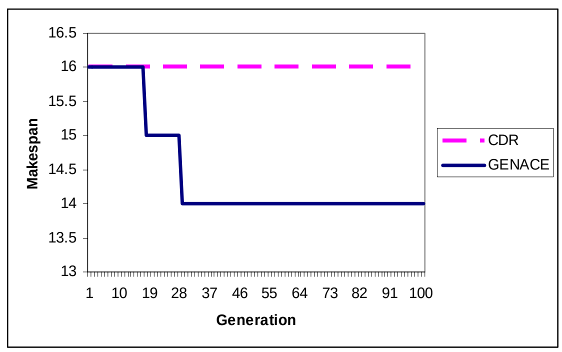


Figure 15. Results of 8 jobs by 8 machines P-FJSP

Table III below gives the results of applying the GENACE algorithm to solve the 9 P-FJSP problems taken from Brandimarte [11].

TABLE III. MAKESPAN OF 9 P-FJSP INSTANCES

Name	Size	Brandimarte [11]	GENACE	Improve
Mk1	10x6	42	41	2.38%
Mk2	10x6	32	29	9.37%
Mk4	15x8	81	67	17.28%
Mk5	15x4	186	176	5.37%
Mk6	10x15	86	68	20.93%
Mk7	20x5	157	148	5.73%
Mk8	20x10	523	523	0%
Mk9	20x10	369	328	11.11%
Mk10	20x15	296	231	21.95%

The first and second columns give the problem specifications; the next two columns give the results of applying corresponding algorithms and the last column is the deviations of [11] and the GENACE algorithm (((3^{rd} column – 4^{th} column)/ 3^{rd} column) * 100%). The results indicate that the GENACE algorithm outperforms Brandimarte's approach in 8 out of 9 problems.

VII. CONCLUSION

In this paper, we proposed an efficient methodology to solve the FJSP with recirculation. First, we introduce how CDRs are used to solve the FJSP and provide a bootstrapping mechanism to initialize GENACE. Next, we apply cultural evolution to maintain knowledge of schemata and resource allocations learned over each generation. The belief spaces in turn influence mutation and selection of individuals. As the results of crossover and mutation always maintain solution feasibility, a repair process is avoided and hence computation time is reduced. Experimental results show that GENACE obtains better upper bounds for 11 out of 13 benchmark FJSP problems, with improvement factors of 2% to 48%. It validates the 'no-free lunch theorem' [19] that knowledge of the problem domain can help the search process to get better results.

In this paper, we have assumed that all machines are available at time 0 and all jobs can be processed at time 0. However, in practice, each job always has a release date and a due date, each machine may not be available at time 0 and machines need to be maintained. Therefore, the immediate future will focus on extending this method to solve FJSPs with additional constraints such as jobs with release and due dates, machines are unavailable at time 0 and maintenance schedules.

ACKNOWLEDGMENTS

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

REFERENCES

[1] M. Kolonko, "Some New Results on Simulated Annealing Applied to the Job Shop Scheduling Problem," *European Journal of Operational Research*, vol. 113, pp. 123-136, 1999.

[2] E. Nowicki and C. Smutnicki, "A Fast Taboo Search Algorithm for the Job Shop Problem," *Management Science*, vol. 42(6), pp. 797-813, 1996.

[3] D.C. Mattfeld, *Evolutionary Search and the Job Shop: Investigations on Genetic Algorithms for Production Scheduling*, Physica/Springer Verlag, Heidelberg, 1996.

[4] M.R. Garey, D.S. Johnson, R. Sethi, "The complexity of flow shop and job-shop scheduling," *Mathematics of Operations Research*, vol. 1(2), pp. 117-129, 1976.

[5] M. Pinedo. and X. Chao, *Operations scheduling with applications in manufacturing and services*, Chaper 3, MCGraw-Hill , 1999.

[6] H. Oliver and R. Chandrasekharan, "Efficient dispatching rules for scheduling in a job shop," *International Journal of Production Economics*, vol. 48(1), pp. 87-105, 1997.

[7] S. Panwalkar and I. Wafik, "A Survey of Scheduling Rules," *Operations Research*, vol. 25(1), pp. 45-61, 1977.

[8] I. Kacem, S. Hammadi, and P. Borne, "Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 32(1), pp. 1-13, 2002.

[9] K. Mesghouni, S. Hammadi, P. Borne, "Evolution programs for job-shop scheduling," *Proc. IEEE International Conference on Computational Cybernetics and Simulation*, vol. 1, pp. 720-725, 1997.

[10] 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*, vol. 60, pp. 245-276, 2002.

[11] P. Brandimarte, "Routing and scheduling in a flexible job shop by tabu search," *Annals of Operations Research*, vol. 22, pp. 158-183, 1993.

[12] H. Chen, J. Ihlow, and C. Lehmann, "A genetic algorithm for flexible job-shop scheduling," *Proc. IEEE International Conference on Robotics and Automation*, vol. 2**,** pp. 1120 -1125, 1999.

[13] D. .E.Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addition-Wesley , 1989.

[14] Tay J.C. and Wibowo D., "An Effective Chromosome Representation for Evolving Flexible Job Shop Schedules," accecpted for publication in *Genetic and Evolutionary Computation Conference* 2004.

[15] V. Ramiro, R. Camino, J.P. Vela, and G. Alberto, "A knowledge-based evolutionary strategy for scheduling problems with bottlenecks," *European Journal of Operational Research*, vol. 145(1), pp. 57-71, 2003.

[16] R. G. Reynolds, "An introduction to Cultural Algorithms," *Proc. 3rd Annual Conference on Evolutionary Programming*, River Edge, NJ World Scientific, pp. 131-139, 1994.

[17] J. Doyle, "A truth maintenance system," *Artificial Intelligence*, vol. 12, pp. 231-272, 1979.

[18] J.H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, reprinted by MIT Press, 1992.

[19] D.H. Wolpert, W.G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1(1), pp. 67-82, 1997.