The University Of Sheffield.

Department Of Mechanical Engineering

BEng Mechanical Engineering

Differential evolution optimisation on flexible job shop scheduling

May 2017

Qi Jin

Dr Manson, Graeme

Thesis submitted to the University of Sheffield in partial fulfilment of the requirements for the degree of Bachelor of Engineering

ABSTRACT

This paper demonstrates a hybrid differential evolution algorithm (HDE) combining local search algorithm with basic differential evolution algorithm, which intends to solve the flexible job shop scheduling (FJSS) problem. As far as the author aware, there is limited research on applying differential evolution to solve flexible job shop scheduling problem, which reveals that there is deep potential in improving the performance on tackling FJSS problem. Firstly, the description of FJSS problem and two regular representation techniques are introduced where a sample FJSS instance is analyzed throughout the section for understanding. The conversion between problem description and corresponding mathematical model is presented. Secondly, the procedure of basic DE algorithm is clarified, which is selected to be the optimization tool for solving FJSS. Thirdly, the hybrid differential evolution algorithm is purposed where the encoding & decoding mechanism and the local search algorithm are emphasized. The computational simulation results for a set of typical benchmark instance is demonstrated afterwards. Lastly, the analysis on corresponding simulation results and discussion on the controlling parameters of HDE is demonstrated. The research on self-adaptive differential evolution (SADE) is clarified for future improvement.

CONTENTS

1 INTRODUCTION

Engineers are often faced with different challenges of improving their design's performance within the lowest cost. For instance, civil engineers may try to minimize the weight or maximize the strength when they design a bridge and mechanical engineers may concern about how to improve their system efficiency with minimum cost on the parts. Among a range of engineering problems, Flexible Job Shop Scheduling (FJSS) is regarded as one of the most difficult and important optimization problems in the manufacturing field. Due to the intensive competition between manufacturing industries, the rising demand from the market and customers as well as the rapid developing production scale, how to control the production cost by scheduling the production procedure reasonably has always been highly concerned by the industries, which makes it significant for the industries to investigate the solutions to specific FJSS problems. As the possible solutions to FJSS increase exponentially with increasing number of jobs even if there are only two machines and three operations for each job (Garey et al., 1976), FJSS is also known as a NP-hard problem (Jain & Meeran, 1999). NP-hard (Non-deterministic polynomial-hard) problems involves the problems where the solutions to these problems cannot be checked by a specific algorithm containing one or several polynomials. Taking the Travelling Salesman Problem (Applegate et al., 2006), one of the classical NP-hard problems, as an example, if a schedule with total expense less than C is required, all the possible travelling schedules need searching instead of determining the available schedules directly from an effective algorithm.

With the rapid development of technology, various optimization techniques have been developed to tackle these complex optimization problems. In this paper, a hybrid differential evolution (HDE) algorithm is purposed to solve the flexible job shop scheduling problem, where a local search algorithm is implemented to improve the performance of differential evolution algorithm.

According to the original aim and the objective, the self-adaptive differential evolution algorithm was intended to be implemented in the purposed HDE algorithm. However, due to the massive effort on consummating the current HDE algorithm and limited time, the SADE algorithm is not accomplished completely. The research on SADE which has been conducted is concluded in section 6.3 so that further work could be done to proceed the study on SADE based on it in the future.

1.1 Aim and objectives

In this section, aim and objectives of this project are introduced to demonstrate the primary focus of the corresponding research.

1.1.1 Aim

Develop a self-adaptive approach to solve the Flexible Job Shop Scheduling problem by applying the differential evolution optimisation algorithm.

1.1.2 Objectives

a) Understanding the theory of Differential Evolution and be able to demonstrate the basic procedure of Differential Evolution;
b) Be able to encode DE algorithm with simulation tools such as Matlab to solve different optimization testing problem;
c) Be able to analyze the obtained result of optimization and adjust the parameters based on the convergence performance to improve the optimization;
d) Analyzing and understanding the Flexible Job Shop Scheduling problem and be able to build the mathematical model for this case and encode it;
e) Understanding the theory of local search and be able to implement it to speed up the original DE algorithm;
f) Understanding how self-adaptive technique works and be able to implement it in the DE algorithm;
g) Be able to debug and optimize the code applied for the targeting case;
h) Be able to analyzing and comparing the computational results produced by testing the well-known benchmark data, and illustrating the result with appropriate graphs and tables.

1.2 Literature Review on FJSS

As an extension of classical Job Shop Scheduling (JSS) problem, Flexible Job Shop Scheduling (FJSS) was first studied by Bruker and Schlie (1990) who purposed a polynomial algorithm to solve flexible job shop scheduling problems with two jobs. In order to solve real complex instance, a hierarchical approach was developed in several early literatures by Brandimarte (1993) and Tung et al. (1999), where the assignment of operations on the machines and the sequence of operations are considered separately. This approach was then widely regarded as a basic idea of FJSS problem as the complexity is reduced through the decomposition of FJSS. Brandimarte (1993) also started applying taboo search heuristic (TS) to solve scheduling sub-problem., which became a popular techniques afterwards and further developed by Hurink et al. (1994). An integrated approach was then purposed by Dauzere-Peres and Paulli (1997) where the classical disjunctive graph model for FJSS was developed into a neighborhood structure. In this structure the distinction between reassigning and resequencing an operation was eliminated. Based on the study of neighborhood structure, two neighborhood functions and the improved taboo search procedure were purposed and by Mastrolilli and Gambardella (2002) and 120 new better solutions were found in their computational study on 178 FJSS problems, which was outperformed at that time.

After Kacem et al. (2002) purposed a new genetic algorithm (GA) controlled by the assignment model, whose instance set was then widely used as a benchmark to test the performance of new purposed algorithms. The Kacem benchmark is also included in the computational studies in this paper to analyze the purposed HDE algorithm. Meanwhile, a modified GA was proposed by Jia et al. (2003) in order to solve distribute scheduling problems and the FJSS.

In addition, the particle swarm optimization (PSO) was also frequently applied to solve multi-objective FJSS. Xia and Wu (2005) purposed a hybrid algorithm where the machine assignment was relied on PSO while simulated annealing (SA) algorithm was applied to arrange the operation sequence. Moslehi and Mahnam (2011) combined PSO and local search algorithm to improve the performance of multi-objective approach.

Recently, the differential evolution (DE) algorithm purposed by Storn and Price (1997), which is a population-based evolution algorithm, has drawn growing attention to its potential on solving scheduling optimization problems with outstanding performance. In the beginning, Storn (1996) aimed to tackle the signal processing problem by applying DE algorithm, whose applicability was then proved by various real applications with better performance including sensor optimization (Manson & Worden, 2001), neural network training (Ilonen et al., 2003), power systems optimization (Lakshminarasimman et al., 2008), robot path planning (Gonzalez et al., 2009), and even nuclear safety (Di Maio, Baronchelli, & Zio, 2014). Compared to previous algorithms such as genetic algorithm and taboo search, DE algorithm has the advantages including simplicity of parameter controlling, fast convergence speed as well as high compatibility.

As for the recent applications on scheduling problems, for instance, Kazemipoor et al. (2012) purposed an efficient metaheuristic algorithm based on DE to schedule the multi-skilled project portfolio in an organization, where the effectiveness of DE was confirmed compared to taboo search based algorithm. In 2016, Erdem and Ali (2016) optimized the performance of fuzzy logic controller under different traffic scenarios via DE.

When it comes to the application of DE algorithm on FJSS, Yuan and Hua (2013) was the first noticing that there was not enough study on implementing a DE algorithm to tackle FJSS problem. In their paper, two hybrid DE algorithms was developed based on DE framework combining two improved neighborhood structures in the local search algorithm, which has been very competitive algorithms since then.

However, the performance of DE algorithm is highly dependent on the controlling parameters setting (Brest et al., 2006), where abundant simulation is needed to determine the suitable parameter setting. According to Liu and Lampinen (2005), research was conducted on how to allow the controlling parameters to be self-adapted without any intervention and a Self-adaptive Differential Evolution algorithm (SADE) was purposed. According to their simulation results, SADE was able to demonstrate outstanding performance consistently on problems with different complexity.

In conclusion, as far as the author aware, there is limited research on applying differential evolution to solve flexible job shop scheduling problem apart from Yuan and Hua's study, which reveals that there is deep potential in improving the performance on tackling FJSS problem. In order to make a relevant contribution to the development of this field, a SAHDE algorithm is intended to be developed in this paper based on the algorithm purposed by Yuan and Hua to solve FJSS.

2 FJSS PROBLEM ANALYSIS

In this section, the flexible job shop scheduling problem is analyzed based on three parts. In sub-section 2.1, the description of FJSS problem is introduced with the demonstration of a basic FJSS instance. Two sample solutions to FJSS are also illustrated with the evaluation procedure. In sub-section 2.2 & 2.3, two general evaluation methods to FJSS are illustrated, which is Gantt chart and Disjunctive graph,

2.1 Problem description

Flexible job shop scheduling problem is a classical optimization problem which aims to allocate the resource reasonably. In order to analyze the problem clearly, a sample instance of FJSS problem named Instance 1 is the sample normally referred to in this section as shown in Table 2.1. The mathematical model of this problem can be set as follows:

Assuming there are two sets M = {M1, M2, M3....} and J = {J1, J2, J3...} where M_m is the number of the machine and J_n is the number of jobs. For each independent job J_i, there are several operations for the job to complete in a specific sequence composing the operation set {$O_{i,1}$, $O_{i,2}$, $O_{i,3}$, ...} where $O_{i,j}$ refers to the jth operation of the ith job. For each operation the processing time on different machines is independent and variant.

In Table 2.1, the processing time for each operation $O_{i,j}$ on different machines is illustrated. To be emphasized, the processing time in this paper is dimensionless, which only aims to create a timeline for analysis. For example, $O_{1,2}$, the second operation of Job 1, consumes 4 unit time on Machine 1, a unit time on Machine 2, and 3 unit time on Machine 3. Based on this model, it is convenient to transfer the real problem into a logical mathematical instance. However, for some operations the production on the specific machine is not available, which is represented as a tag '-' in the table.

From the mathematical model and Table 2.1 below, the difference between flexible job shop scheduling and the classical JSS problem lies in the fact that FJSS allows operations to be assigned on any available machines while JSS follows a fixed assignment of machine for each operation.

Table 2.1 Processing times table (I_1)

$O_{i,j}$	M_1	M_2	M_3
J_1			
$O_{1,1}$	2	-	3
$O_{1,2}$	4	1	3
J_2			
$O_{2,1}$	-	3	3
$O_{2,2}$	6	5	4
$O_{2,3}$	3	-	2
J_3			
$O_{3,1}$	4	5	2
$O_{3,2}$	3	-	2

Although the operations can be executed randomly, there are several basic constraints in this problem. Firstly, the operations for each job should be proceeded in order. In other words, a new operation cannot start unless the previous operation for the same job has been completed. However, every job is independent so that operations from different jobs can be processed at the same time. Secondly, one machine can only process one operation each time. Thirdly, the operations cannot be paused and substituted with another operation in progress. Based on these constraints, the aim of the optimization is to shorten the manufacturing time by swapping the arrangements for the operations and eliminate the gap between operations which is the spare time of machines. In order to assess the performance of optimization, a cost function is applied to calculate the total processing time, whose result is defined as *makespan*. The makespan is the latest completion time to complete all the jobs. As introduced before, makespan can be optimized by changing the operation assignment in order to process the operations on an available machine with shorter processing time and let the transition time as close to zero as possible.

By producing an operation time table which illustrated a possible solution to the problem, the makespan can be determined. The operation time table is produced to show the earliest start time and earliest completion time for each operation by assigning the each operation $O_{i,j}$ to a specific machine M_m, where the maximum completion time is the expected makespan. Table 2.2 is a sample operation time table of the Instance 1 in Table 2.1.

The first column of Table 2.2 represents the processing sequence of operations, which means the operations will follow the order of $O_{2,1} \rightarrow O_{1,1} \rightarrow O_{3,1} \rightarrow O_{3,2}$

$\rightarrow O_{2,2} \rightarrow O_{2,3} \rightarrow O_{1,2}$. This processing sequence has been corrected to avoid inappropriate sequence within each job i.e. $O_{1,1}$ is always proceeded before $O_{1,2}$. The repaired scheme for the correction and the transfer scheme between $O_{i,j}$ and the fixed ID '1,2,3...ect' can be found in section 4.4.1, which is applied for convenient identification on the operation number for different jobs by the algorithm. The second column of Table 2.2 illustrates the machine assignment for each corresponding operation. Based on the selection of machine and the operation sequence, the start time and end time can be worked out. For instance, for operation 1($O_{1,1}$), it is the first operation on machine 1. According to the processing time table, operation 1 will consume 2 unit time. For operation 3 ($O_{2,1}$), it takes 3 unit time as it is also the first operation on machine 3 which has no effect on the operation on other machines. Moving on to operation 2 ($O_{1,2}$), it follows its previous operation in Job 1 ($O_{1,1}$) and the operation proceeded on machine 3 previously which is the operation 3 ($O_{2,1}$), so it needs to wait for their completion to continue proceeding and the final completion time for operation 2 ($O_{1,2}$) is 6 unit time. Similar to $O_{1,2}$, the gap appears between operation $O_{2,2}$ and operation $O_{2,3}$, owing to the incompletion of operation $O_{3,2}$ which delays the start of operation $O_{2,3}$ a unit time while operation $O_{2,2}$ is complete. After all the operations are completed, the largest ending time, which is 12 unit time according to Table 2.2, is determined as the makespan for this solution. Another optimized solution is illustrated in Table A1 in appendix, where the gap time is eliminated to enhance the schedule and the makespan is reduced to 9 unit time.

Table 2.2 Operation time table for a possible solution to Instance 1

Operation ID	Machine	Start time	End time
3 ($O_{2,1}$)	3	0	3
1 ($O_{1,1}$)	1	0	2
6 ($O_{3,1}$)	1	2	6
7 ($O_{3,2}$)	1	6	9
4 ($O_{2,2}$)	2	3	8
5 ($O_{2,3}$)	1	9	12
2 ($O_{1,2}$)	3	3	6

2.2 Gantt chart

For small scale of machines and operations, the Gantt chart is introduced to represent the process of arrangement. Gantt chart was invented by Henry Laurence in 1917 which is basically a bar chart illustrating the available schedule directly. In the Gantt chart the horizontal axis denotes time axis and the machine assignment lies on the vertical axis. From Figure 2.1 which is a Gantt chart of Table 2.2, the corresponding schedule is illustrated as the bars in each line, and the total time consumed by the longest bar represents the makespan of this schedule.

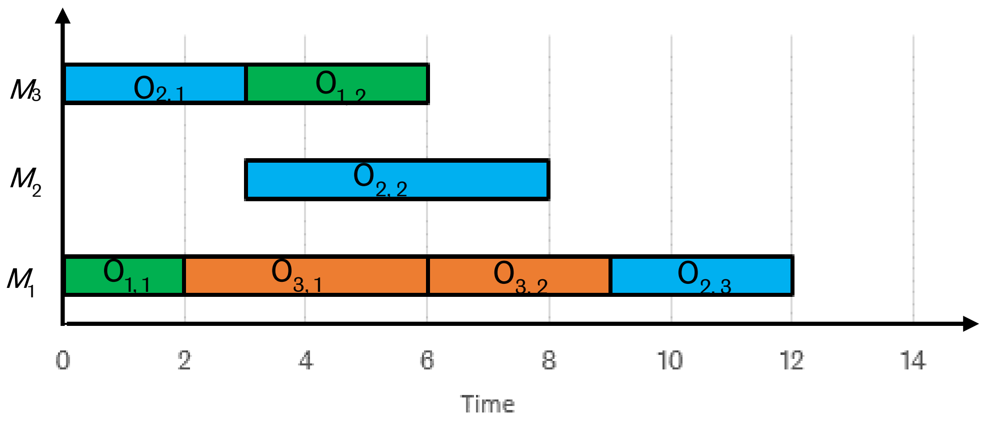


Figure 2.1 Gantt chart of Table 2.2

Through the Gantt chart the manager can easily analyze the whole arrangement and reassign operations to eliminate the bottleneck. In Figure 2.1, there is a clear gap at the beginning of Machine 2 lasting 3 unit time.

2.3 Disjunctive graph

In order to analyze the potential improvement for local schedules, another representation method of schedules commonly used for FJSS is disjunctive graph. Disjunctive graph is a graph of schedule G consisting of nodes and arcs and can be represented as $G = (V, C \cup D)$. In this graph, V denotes a set of nodes which corresponds to all the operations as well as two extra nodes representing the starting node and termination node; C denotes a set of conjunctive arcs connecting the consecutive operations within a same job so the sequence order can be represented on the graph. D is a set of all the disjunctive arcs which illustrates the path of operations proceeded on the same machine. The processing time is generally labelled at the start of each arc i.e. labelled on each node as the weight of corresponding nodes. A sample disjunctive graph is shown in Figure 2.2.

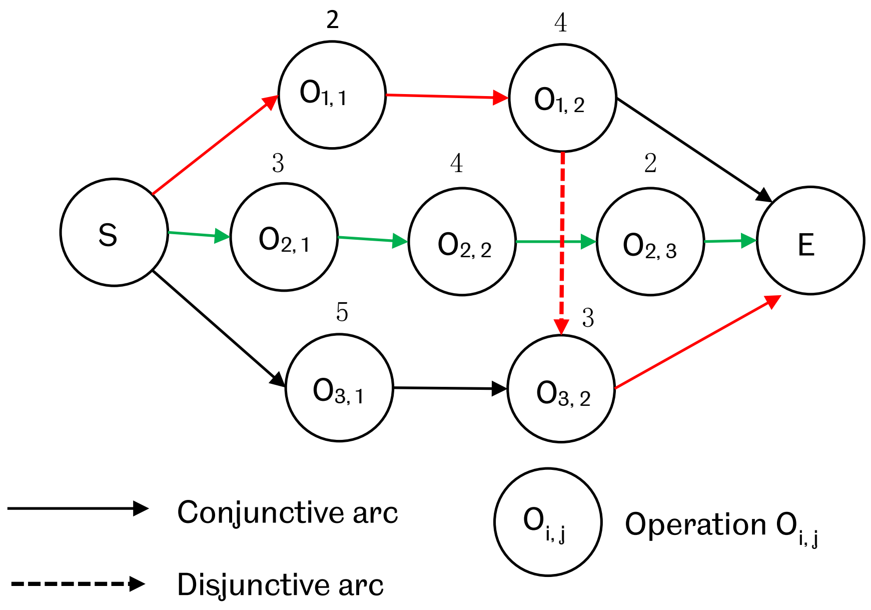


Figure 2.2 Illustration of the disjunctive graph with critical paths

In a disjunctive graph, if there is no cyclic path wherein a node is reachable from itself as shown in Figure A3 in appendix, the corresponding schedule is regarded as a feasible schedule. In a feasible schedule, the longest path in the disjunctive graph connecting the starting node and ending node is regarded as the critical path of this schedule. By summing the weight of nodes lying on this path, the makespan is obtained and the corresponding operations on this path are called critical operations. For example, in Figure 2.2, two paths $O_{1,1} \rightarrow O_{1,2} \rightarrow O_{3,2}$ as well as $O_{2,1} \rightarrow O_{2,2} \rightarrow O_{2,3}$ which have the same makespan 9 can be regarded as the critical path.

3 BASIC DE ALGORITHM

Differential evolution is a population-based stochastic optimization algorithm which aims to minimize or maximize the result of an objective function $f(\vec{x})$ where $\vec{X} = [x_1, x_2, \dots x_D]^T$ is a D dimensional vector randomized in a specific range $[x_{j,min}, x_{j,max}]$, which is the lower bound and upper bound of each variable x_j, forming a candidate solution to the problem. As shown in Figure 3.1, several basic steps of DE process iteratively until a set generation is reached. The detailed process of each stage is introduced as follows based on the evaluation of Rosenbrock function, which is shown as Equation 3.1, and a flow chart for the evaluation process with a sample population is illustrated in figure in appendix.

$$f(x) = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2) + (x_i - 1)^2)$$

Equation 3.1

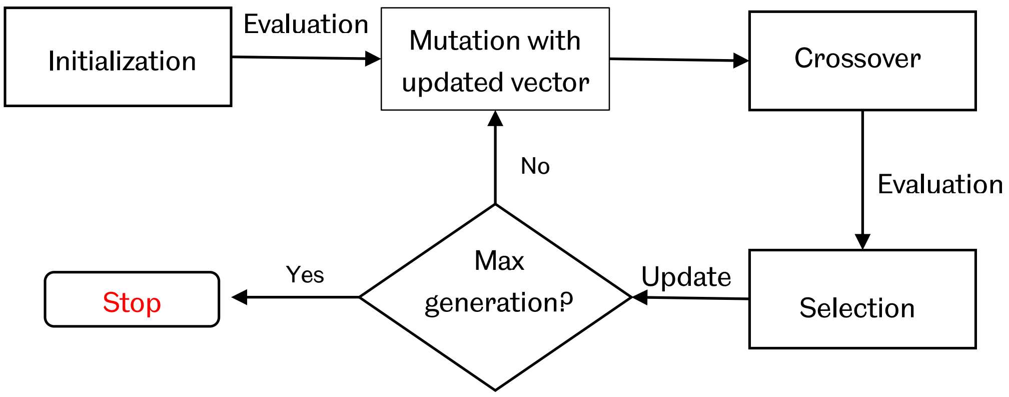


Figure 3.1 Main stages of the basic DE algorithm

3.1 Initialization

Initially, a random D dimensional matrix, also known as a chromosome, $\vec{X} = [x_{1,i,G}, x_{2,i,G}, \dots x_{D,i,G}]^T$ is generated randomly, i= 1, 2, 3....NP where *NP* is the population size and G represents generation number. The range of random number $[x_{j,min}, x_{j,max}]$ is set as constraints i.e. random real numbers within the bounds will be returned to fill the population matrix. For example, if NP is set as 5 and the matrix has a dimension of 2, and the bound is set as [-5, 5], 5 vectors which consists of 2 real numbers between -5 and 5 will be generated randomly to compose a 5*3 matrix. A sample matrix is generated as shown in Table A2 in appendix, where the numbers are rounded to the nearest integer for convenience. This population matrix will be used as the sample in the following stages. At this stage, the scaling factor *F* and the crossover ratio *CR* are set to control the performance of mutation and crossover, whose detailed effect is discussed in section 8. In addition, an initial evaluation on the population matrix by the cost function, which is Rosenbrock function in this case, is proceeded to select the target vector in mutation. As Rosenbrock function is a minimization optimization problem, the vector which obtains the lowest cost value becomes the target vector.

3.2 Mutation

After the initial evaluation on the population matrix, a target vector, or a parent vector, with best fitness value calculated by the objective function will be selected to mix with other two random vector in the population matrix to proceed the evolution and a mutant vector is produced after this process. Similar with the environmental effect in the evolutionary theory, the mutation process is an unpredictable factor which creates more probability of a better solution occurring in the population matrix in order to accelerate the progress of convergence. A mutant vector $\vec{V}_{i,G} = [v_{1,i,G}, v_{2,i,G}, \dots v_{D,i,G}]^T$ is constructed by applying scheme *DE/best/1*, which is selected for this problem among the existing different mutation schemes of DE algorithms. The difference between those schemes is that different mutation functions are applied. For *DE/best/1*, the mutation function is illustrated as:

$$\vec{V}_{i,G} = \vec{X}_{best,G} + F \times (\vec{X}_{r_1^i,G} - \vec{X}_{r_2^i,G})$$

Equation 3.2

where $\vec{V}_{i,G}$ is the parent vector, $\vec{X}_{best,G}$ is the target vector with best fitness i.e.

the vector with the lowest cost value for Rosenbrock function. $\vec{X}_{r_1^i,G}$ and $\vec{X}_{r_2^i,G}$ are two random vector selected from the population matrix, where $r1$ and $r2$ are two different random integers from [1, NP], and both of them are also distinct from the base index i. F is the scaling factor which is between [0, 2] and usually selected from 0.6~0.8 in normal DE optimization problems to process enough global search during the evolution of the population. According to equation 3.2, a large scaling factor will force $\vec{V}_{i,G}$ becoming diverse and moving away from $\vec{X}_{best,G}$. On the contrary, a small scaling factor will generate a mutant vector $\vec{V}_{i,G}$ close to $\vec{X}_{best,G}$. In addition, the selection of scaling factor highly depends on the cooperation of crossover ratio, which will both affect the performance of the algorithm so a proper choose of parameter will improve the computational efficiency of the algorithm searching for the optimal solutions. According to the simulation results of Rosenbrock function, *F* = 0.7 and *Cr* = 0.6 is a good choice in this case. Moreover, if the value of the initial mutant exceeds the boundary of [-5, 5], the exceeded value will be adjusted to the closest boundary value.

As shown in Figure A2 in appendix, the target vector is [-1 1] which has the lowest cost value 4 from Rosenbrock function. In order to form the mutant matrix, two distinct vector [-4 2] and [4 -3] are selected from the population matrix randomly and equation 3 is applied. As one of the initial result jumps over the bound, the vector is adjusted to [-5 5]. This process is repeated 5 times to generate a mutant matrix with the same size as the original matrix,

3.3 Crossover

For every mutant vector, crossover is applied to produce a set of vector with individual characteristics. In the crossover phase, a crossover vector $\vec{V}_{i,G} = [v_{1,i,G}, v_{2,i,G}, \dots v_{D,i,G}]^T$,also called trial vector, is generated by combining the elements from the parent vector $\vec{X}_{i,G}$ and its mutant vector $\vec{V}_{i,G}$ with a fixed probability *Cr* following equation 3.3.

$$u_{j,i,G} = \begin{cases} v_{j,i,G}, & \text{if rand}(0,1) \leq \text{Cr or j} = \text{q} \\ x_{j,i,G}, & \text{otherwise} \end{cases}$$

Equation 3.3

where *Cr* is the crossover ratio selected within [0, 1] and q is an integer index selected from [1, D] randomly which aims to guarantee that the trial vector contains at least one element from the original parent vector. The crossover ratio controls whether the characteristic of the child vector will inherit from the parent vector or the mutant vector. To extend the searching area, the crossover ratio is usually chosen from between 0.6 and 0.8. The child matrix is then generated by crossover in Figure A2.

3.4 Selection

As an important phase in evolutionary process, in selection, the trial vector and the parent vector will compete against each other in order to guarantee that the child vector $\vec{X}_{i,G+1}$,which stands for the updated population vector in the next generation, has the best fitness for all the element in this vector, which can be described as

$$\vec{X}_{i,G+1} = \begin{cases} \vec{U}_{i,G}, & \text{if } f(\vec{U}_{i,G}) \leq f(\vec{X}_{i,G}) \\ \vec{X}_{i,G}, & \text{otherwise} \end{cases}$$

Equation 3.4

where $f(x)$ is the objective function for the problem. Assuming the problem is a minimizing optimization problem, the elements from the trial vector $\vec{U}_{i,G}$ will replace the original ones $\vec{X}_{i,G}$ if the fitness value calculated from $\vec{U}_{i,G}$ is better. Otherwise the child vector remains the same as the parent vector. In Figure A2, the third vector in the population is replaced from the child vector with a better cost value.

4 HDE ALGORITHM FOR FJSS

In this section, a hybrid differential evolution algorithm is proposed in order to solve the FJSS. In sub-section 4.1, the framework of HDE algorithm is clarified and a flow chart of HDE is illustrated in Figure 4.2. From sub-section 4.2 to 4.6, each key stage of HDE algorithm is demonstrated respectively.

4.1 Framework introduction

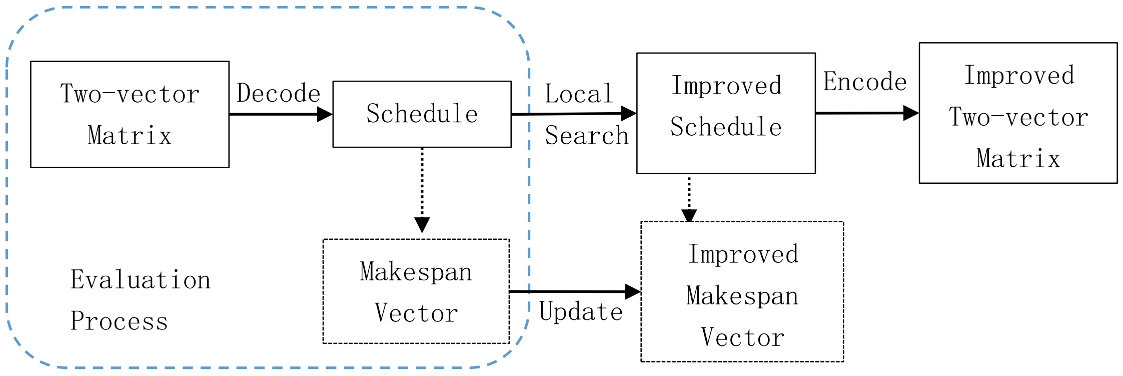


Figure 4.1 Flow Chart of Evaluation process & Local search

The framework of HDE algorithm is established based on the basic DE algorithm combined with a heuristic method called local search. As it is shown in Figure 4.1, the algorithm will keep searching for the better solution until the fitness value converges so the maximum generation number will be adopted by the performance of convergence. By adjusting the parameters appropriately, the convergence performance can be improved.

Two sets of data are required to input into the algorithm: processing time table and job operation vector. The job operation vector represents the number of operation for each job which is used for the algorithm to recognize the operation's ID. The detailed theory is clarified in section 4.3.

Compared to the basic DE algorithm which concentrates on searching for optimal solutions within the global region, the HDE employs the local search algorithm to enhance the exploitation of local improvement i.e. improve the vector's corresponding schedule directly to minimize its makespan rather than generating new vectors to check whether their makespan are improved. In addition, the local search is only applied on the trial vector $\vec{V}_{i,G}$ to avoid the possible cyclic search, which will be discussed in section 4.6.3.

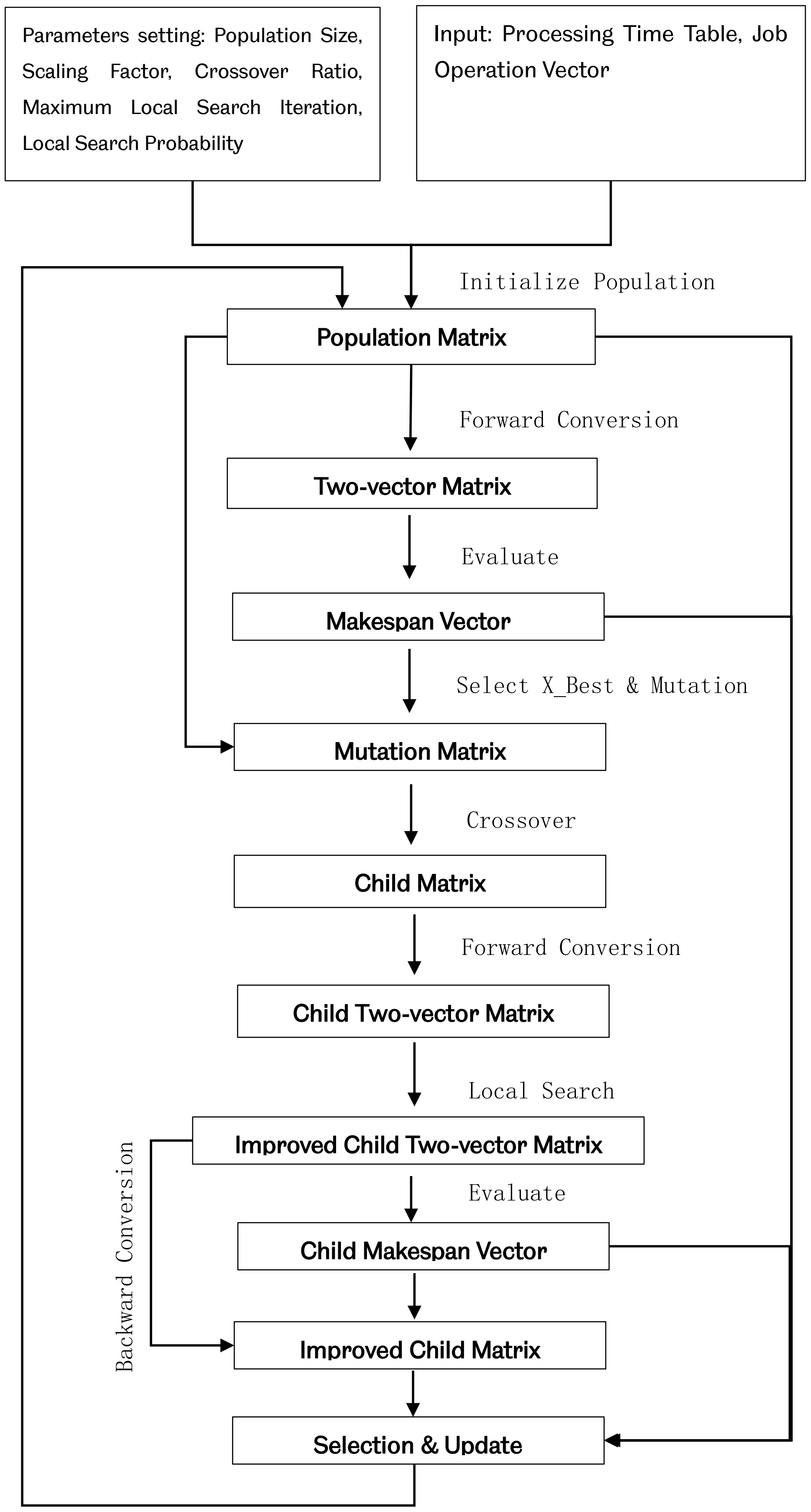


Figure 4.2 Flow chart of the HDE algorithm

Besides, two parameters $Iter_{max}$, the maximum iteration, and P_l, the local search probability are set to control the performance of local search.

After encoding the original population into two vector code through forward conversion, the evaluation process, which also performs a key role in DE as well as the implementation of local search, is proceeded in order to generate the schedule and record the makespan by decoding the two vector code.

Once the available schedule corresponding to the trial vector is generated through the evaluation process, local search is implemented iteratively to further improve the schedule to obtain a minimum makespan which can be decoded to generate a new two vector matrix. A backward conversion is then applied to convert the two vector matrix into an improved chromosome which enters the new generation instead of the original one with worse makespan.

4.2 Initialization

As preparation for the two-vector code, the chromosome $\vec{X} = [x_{1,i}, x_{2,i}, \dots x_{D,i}]^T$ is generated with a dimension of the chromosome D which satisfies $\mathrm{D} = 2\mathrm{d}$ where d is the total number of operations of the corresponding FJSS instance. The chromosome is divided into two parts to represent the two key information separately in the problem, i.e. machine assigning and sequencing. The first part $\vec{X} = [x_{1,i}, x_{2,i}, \dots x_{d,i}]^T$ denotes the assignment of machine for each operation. The second half $\vec{X} = [x_{d+1,i}, x_{d+2,i}, \dots x_{D,i}]^T$ describes the sequence of operations. According to the basic DE initialization, the bound in the proposed algorithm is set as [0, 1] and the population is initialized randomly within the bound.

4.3 Two vector code

The two vector code is encoded from the chromosome and consists of two vectors: machine assignment vector and operation sequence vector (Yuan & Hua, 2013).

In order to describe the operation number conveniently, a fixed ID scheme is applied on all the operations as shown in Table 4.1 for the Instance 1. For example, operation 4 refers to operation $O_{2,2}$.

Table 4.1 ID transfer scheme for operations

$O_{i,j}$	$O_{1,1}$	$O_{1,2}$	$O_{2,1}$	$O_{2,2}$	$O_{2,3}$	$O_{3,1}$	$O_{3,2}$
Fixed ID	1	2	3	4	5	6	7

Although the algorithm now can identify the operations under this ID scheme, the operations still cannot be classified to their corresponding jobs. The job operation vector, which representing the number of operation for each job, is required inputting to guide the algorithm to classify the IDs. According to the scheme in Figure 4.3 a start operation vector can be figure out from job operation vector. As shown in Figure 4.3, the first operation of each job is recognized as an interval between different jobs, so the operations before (not including) Operation 3 and after Operation 1 are classified in Job 1.

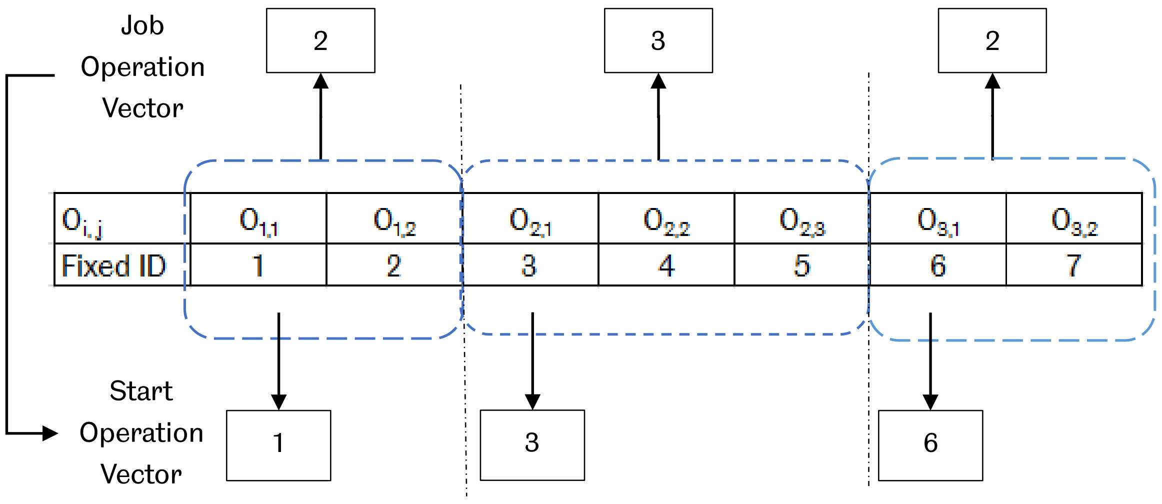


Figure 4.3 Illustration of classification of operations under ID scheme

4.3.1 Machine assignment vector

The machine assignment vector $\vec{R} = [r_1, r_2, \dots r_d]^T$ describes that for an operation j ,where j = 1,2,3...d, the r_jth machine in the available machine set is selected to perform this operation. Based on Instance 1, a sample machine assignment vector is illustrated in Figure 4.4. For instance, $r_2 = 3$ indicates that the operation $O_{1,2}$ is performed on the third machine in the available machine matrix generated from Table 2.1 , which refers to machine M_3 in the corresponding shadowed block.

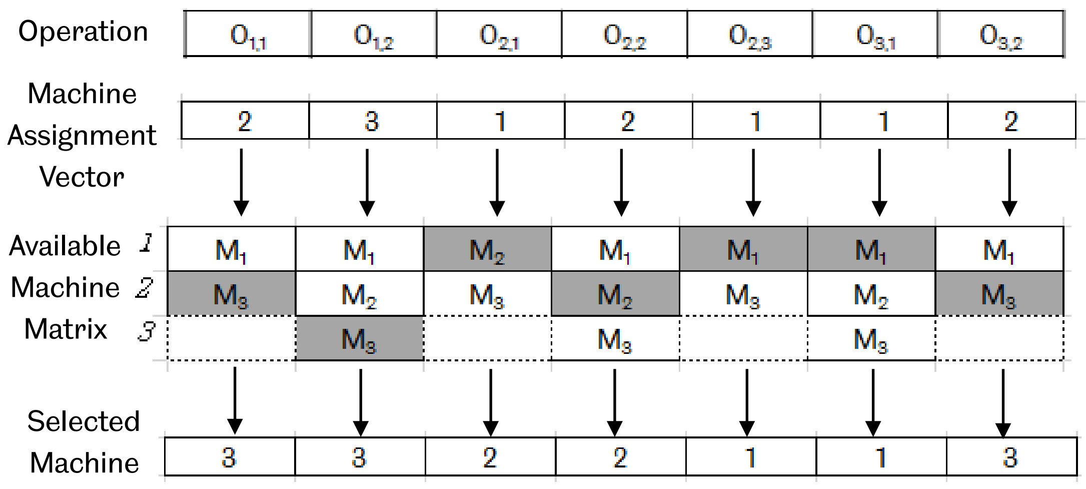


Figure 4.4 Sample illustration of machine assignment vector

4.3.2 Operation sequence vector

The operation sequence vector, represented as $\vec{S} = [s_1, s_2, \dots s_d]^T$ is the machining order of each operation's corresponding ID. Each operation is arranged with priority in $\vec{S}$ so the higher the operation locates, the earlier it proceeds on the machines. According to an example shown in Figure 4.5, this operation sequence vector represents that the operations will follow the order of $O_{2,1} \rightarrow O_{1,1} \rightarrow O_{3,1} \rightarrow O_{3,2} \rightarrow O_{2,2} \rightarrow O_{2,3} \rightarrow O_{1,2}$. It needs to be emphasized that all the jobs are independent which means that operations of other jobs such as $O_{1,1}$ and $O_{3,1}$ can be processed together with $O_{2,1}$ if they are assigned on different machines from $O_{2,1}$'s machine (see the 'start time' column in Figure A8).

Operation Sequence Vector	3	1	6	7	4	5	2
Operation	$O_{2,1}$	$O_{1,1}$	$O_{3,1}$	$O_{3,2}$	$O_{2,2}$	$O_{2,3}$	$O_{1,2}$

Figure 4.5 Sample illustration of operation sequence vector

Operation Sequence Vector	3	1	5	4	7	6	2
Operation	$O_{2,1}$	$O_{1,1}$	$O_{2,3}$	$O_{2,2}$	$O_{3,2}$	$O_{3,1}$	$O_{1,2}$

Figure 6.6 Sample illustration of an unrepaired operation sequence vector

However, due to the existing priority of operations within a job i.e. $O_{1,1}$ is always proceed earlier than $O_{1,2}$, some ID sequences are not feasible so adjustment is needed during the forward conversion process. Figure 6.6 shows a possible sample of unrepaired operation sequence vector, where operation $O_{2,2}$ should precedes operation $O_{2,3}$, and operation $O_{3,1}$ should precedes operation $O_{3,2}$.

4.3.3 Encoding and decoding

It is convenient to process the conversion between the two vector matrix and the operation time table as shown in Figure A8. In order to produce a feasible schedule, according to the theory in the previous sections, the operations are assigned to corresponding machines in machine assignment vector $\vec{R}$ firstly and then the processing time for each operation on their assigned machine are allocated following the order of operation sequence vector $\vec{S}$.

In contrast, by recording the machine assignment for each operation and sorting the operations based on their earliest start time, the operation schedule can be encoded into a two vector code. To be aware that the operation sequence vector may be different from the original one after encoding the schedule and it is acceptable because operations of different jobs are independent and the sequence of these operations is irrelevant with the schedule only if they are processed on different machines.

4.4 Conversion techniques

In this sub-section, two conversion techniques are introduced in order to perform the conversion between the chromosome and the two vector code.

4.4.1 Forward conversion

Forward conversion converts a population matrix $\vec{X}$ containing real numbers within [0, 1] into a two-vector matrix composed of integer. The first half of two-vector matrix $\vec{R}$ refers to the machine assignment for each operation and the second half $\vec{S}$ refers to the ordering of the operations.

For the first half of conversion, $\vec{X_1} = [x_{1,i}, x_{2,i}, \dots x_{d,i}]^T$ is converted to $\vec{R}$. By inputting the available machine vector $\vec{L} = [l_1, l_2, \dots l_d]^T$ which denotes number of available machines for each operation to be assigned, the machine assignment vector is obtained by equation 4.1

$$r_j = ceil\ (l_j \times x_j\ )$$

Equation 4.1

where r_j is the nearest integer towards the positive infinity for the obtained real number. Taking Instance 1 as an example, the available machine vector will be [2 3 2 3 2 3 2]. The conversion process is shown in Figure 4.7, if a sample population vector is given as $\vec{X}_S$.

For the second part of the conversion, the operation sequence vector $\vec{S}$ is converted from $\vec{X}_2 = [x_{d+1,i}, x_{d+2,i}, \dots x_{D,i}]^T$ by the largest position value (LPV) rule (Wang et al., 2010). After applying LPV rule, the ID sequence of corresponding operations sorted in descend order is obtained. However, the ID sequence may contradict the existing operation order where a repair scheme is required to repair the priority of operations within each job.

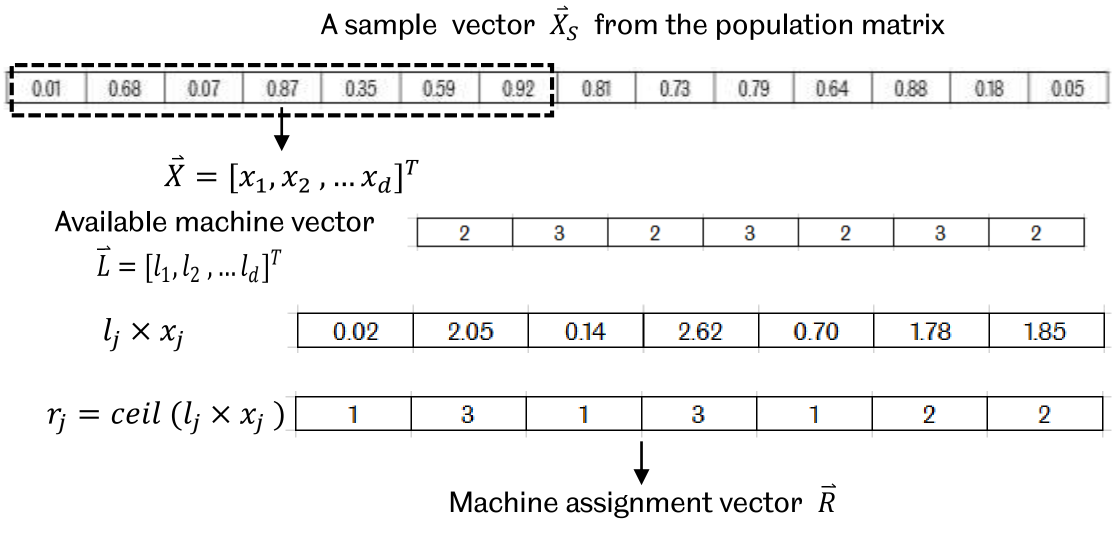

A sample vector $\vec{X}_S$ from the population matrix
0.01 0.68 0.07 0.87 0.35 0.59 0.92 0.81 0.73 0.79 0.64 0.88 0.18 0.05
$\vec{X} = [x_1, x_2, \dots x_d]^T$
Available machine vector $\vec{L} = [l_1, l_2, \dots l_d]^T$
2 3 2 3 2 3 2
$l_j \times x_j$
0.02 2.05 0.14 2.62 0.70 1.78 1.85
$r_j = ceil\ (l_j \times x_j\ )$
1 3 1 3 1 2 2
Machine assignment vector $\vec{R}$

Figure 4.7 Illustration of forward conversion process of $\vec{X}_1$

Assuming the vector is still $\vec{X}_S$ in Figure 4.7, the conversion process is illustrated in Figure 4.8. After sorting the $\vec{X}_2$ by applying LPV rule, the original position of

$\vec{X}_2$ is recorded as the ID sequence of this population vector which needs repairing because the original operation precedence within job 2 is overturned. By swapping the position of highlighted IDs following the initial operation precedence (3, $O_{2,1} \rightarrow$ 4, $O_{2,2} \rightarrow$ 5, $O_{2,3}$), the operation sequence vector is repaired.

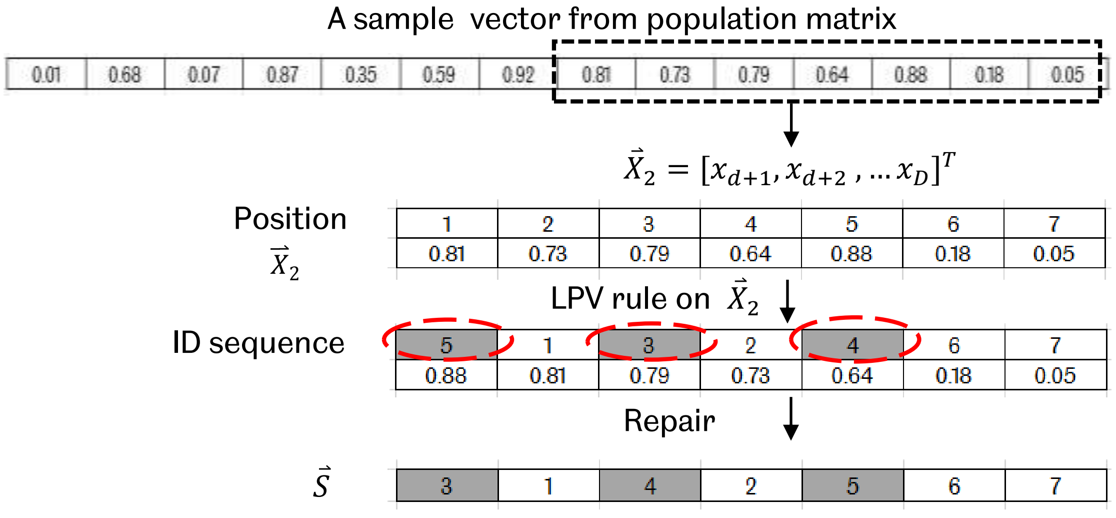


Figure 4.8 Illustration of forward conversion process of $\vec{X}_1$

4.4.2 Backward conversion

Due to the local improvement on the schedule after local search, the two vector code as well as the chromosome needs updating. As we discussed in 4.3.3, the improved schedule can be directly encoded into two-vector code and the procedure is illustrated in Figure A8. After we get a new two-vector code, backward conversion is employed to convert it to the chromosome by two steps similar to forward conversion.

Firstly, the machine assignment vector $\vec{R}$ is decoded to vector $\vec{X}_1$. By analyzing equation 4.1, it is discovered that the bound of generating random elements for chromosome can be calculated through inverse linear transformation of equation as follows

$$x_j = (r_j - 1 + rand(0,1)) / \, l_j$$

Equation 4.2

where x_j is the new $\vec{X}_1$ vector, and rand(0, 1) represents random real number between 0 and 1. A sample calculation process is illustrated in Figure 4.9.

Available machine vector	2	3	2	3	2	3	2
$\vec{X}_1$	1	3	1	3	1	2	2
$r_j - 1 + rand(0,1)$	0.04	2.85	0.93	2.68	0.76	1.74	1.39

$$x_j = (r_j - 1 + rand(0,1))/\ l_j$$

New $\vec{X}_1$	0.02	0.95	0.47	0.89	0.38	0.58	0.70

Figure 4.9 Illustration of backward conversion process of $\vec{X}_1$

Secondly, the vector $\vec{X}_2$ is transformed by applying LPV rule on the original chromosomes and rearranging the obtained modified vector according to $\vec{S}$, which denotes the position of the modified vector in the new vector. A sample conversion is depicted in Figure 4.2. The new $\vec{X}_2$ is obtained by rearranging $\vec{X}_2$ in descending order and then modifying the position of ordered $\vec{X}_2$ based on $\vec{S}$ i.e. s_1 = 3 indicates that the first value in modified $\vec{X}_2$ 0.88 should be at the third place in the new $\vec{X}_2$.

$\vec{S}$	3	1	4	2	5	6	7
Original $\vec{X}_2$	0.81	0.73	0.79	0.64	0.88	0.18	0.05
Modified $\vec{X}_2$	0.88	0.81	0.79	0.73	0.64	0.18	0.05
New $\vec{X}_2$	0.81	0.73	0.88	0.79	0.64	0.18	0.05

Figure 4.2 Illustration of backward conversion process of $\vec{X}_2$

4.5 Evaluation

In the main steps of DE, evaluation is an important step which influence the determination of the target vector and update of the population matrix. The evaluation procedure is processed twice in order to obtain the cost value of chromosomes in the parent matrix and the child matrix respectively from the cost function. In the case of FJSS, the makespan vector for each chromosome is obtained through evaluation by decoding the two vector matrix and generating the operation schedule time table. A sample procedure of evaluation is illustrated in Figure A1 in the appendix.

4.6 Local search

In this sub-section, the local search algorithm is introduced in detail in order to enhance the overall efficiency of HDE algorithm by improving the operation schedule. At first the neighborhood structure is introduced as the platform for local search. Secondly, the procedure as well as the detailed algorithm for local search is clarified. Lastly, cyclic search is discussed as a particular circumstance which has a negative effect on the performance of local search.

4.6.1 Neighbourhood structure

Neighborhood is a set of solutions generated after one of the operations in the schedule is inserted to another available place or assigned to a different machine. An example of neighborhood is illustrated in Figure A6 in appendix. By searching the optimal solution from a neighborhood, the schedule is updated and a new neighborhood structure can be constructed. This process of searching and updating the best local optimum iteratively is defined as local search. However, in job shop scheduling problems, it is not effective to consider all the possible neighborhood structures for all the operations because of higher possibility of cyclic schedules. Therefore, only the neighborhood structures on the critical path are considered in FJSS.

4.6.2 Procedure of local search

To clarify the procedure of local search, a corresponding flow chart is illustrated in Figure A3 in the appendix. The detailed procedure is introduced as following:

In order to perform local search in the neighborhood structure, assuming the operations in the critical path is S-co_1-co_2...-co_w-E, an operation co_x, x = 1, 2...w, on the critical path is moved in two steps (Mastrolilli and Gambardella, 2000):

Step 1: Deletion. From the first operation in the critical path co_x, delete the node v representing co_x and remove all the disjunctive arcs connecting with it in the disjunctive graph. The weight of this node is set to 0.

Step 2: Reassign the deleted operation co_x to another machine M_k and insert the node v following the order of existing operations on this machine M_k in order to generate a new disjunctive graph by adding its related disjunctive arcs. Define this position as $O_{i,j}$ and set the weight of node v as $p_{i,j,k}$, which is the processing time for the operation $O_{i,j}$ on machine M_k.

According to these two steps, assuming G is the current schedule, the neighborhood structure N_1 (G) becomes the new schedule without co_x.

The local search will keep updating the improved schedule G' and the neighborhood structure N_1 (G) iteratively while G', which belongs to N_1 (G), always satisfies $C_{max}(G')<=C_{max}(G)$ and G' is acyclic. A speed-up method is introduced by Hua and Yuan (2013) to find an acceptable schedule more quickly instead of searching the whole N_1 (G). This method also avoid the repeated calculations for the new makespan to determine whether G' satisfies the criterion and decrease the probability of trapping in cyclic solutions.

In order to apply the speed-up method, several variables are defined as shown in the Table A3 in appendix as well as the pseudocode for this method.

In the Table A3, ES and EC is obtained from the active schedule. If we set the makespan of G ,$C_{max}(G)$, as the target makespan, LC for the last operation on each machine is obtained by calculate the latest completion time based on $C_{max}(G)$ and then LS for these operations is obtained by LC=LS + $p_{i,j,k}$. The LC and LS for the remaining operations are derived by generating the operation schedule inversely based on the operation sequence. The derivation process of LC& LS is illustrated in Figure A5.

According to Hua and Yuan, 'if there is a position before $O_{i,j}$ in G^- on machine M_k allowing co_x to be inserted in which creates a new schedule G' satisfying $C_{max}(G')<=C_{max}(G)$, it should guarantee co_x starting as early as EC and finishing as late as LS without delaying $C_{max}(G)$.' which refers to the equation 4.3.

$$\begin{aligned} & max\left\{EC^{G^-}\left(PM(O_{i,j})\right), EC^{G^-}\left(PJ(co_x)\right)\right\} + p_{co_x,k} \\ & \quad < min\left\{LS^{G^-}\left(O_{i,j}\right), LS^{G^-}\left(SJ(co_x)\right)\right\} \end{aligned}$$

Equation 4.3

In other words, because the insertion of co_x will not affect the schedule before it, the earliest start time of co_x in the schedule G' can be determined as the earliest completion time of the latest operation before co_x. By adding the processing time of co_x on the selected machine k, the earliest completion time of co_x in G' is determined. Only if this completion time is earlier than the latest completion time of the operation succeeding co_x, the corresponding schedule G' will be regarded as an improved schedule.

Here Hua and Yuan also suggested that by using the "<" not the "<=" in equation 4.3, co_x can be guaranteed not becoming the critical operation in G' so that cyclic search is avoided as much as possible.

Although this method enhances the speed of approaching the optimal region, there is still possibility for the yielded G' causing cyclic search. In Hua and Yuan's paper, a developed theorem is introduced to filter the available positions for co_x to insert in order to ensure the schedule G' is always feasible. The detailed proof of this theorem can be referred in Mastrolilli and Gambardella (2000).

4.6.3 Cyclic search

As mentioned previously, there is possibility of cyclic search occurring in local search which would cause deadlock in the searching process. In purposed local search algorithm, a new schedule, which is regarded as an 'improved' schedule, is generated only if equation 4.3 is meet, rather than reviewing the makespan of the new schedule first. Therefore, a schedule might be adjusted mistakenly if the condition of Equation 4.3 is satisfied, where the makespan is increased afterwards. This worse solution will be improved by local search in the next iteration and then stuck in this loop until the maximum iteration is approached, which wastes computational ability. A sample of cyclic search is illustrated in Figure A7. As far as it is known, apart from this two-solution cyclic search, it is also possible for the algorithm falling into a loop with a set of three or four solutions. In this purposed algorithm, local search will be interrupted if a sign of cyclic search is detected and the best solution in cyclic search will be regarded as the improved schedule entering the subsequent stages. The developed theorem of avoiding cyclic search is introduced in the previous section.

5 COMPUTATIONAL RESULT

In this section, computational simulations based on a sample instance and two well-known benchmark instances in previous FJSS literature were carried out to evaluate the performance of the proposed HDE algorithm. Firstly, the simulation setup and assessment criterion for the results are introduced. Secondly, the simulation results for selected instances are demonstrated. Lastly, the performance comparison with previous algorithms is conducted.

5.1 Simulation setup

Through reviewing previous FJSS literatures, it is hard to determine a precise assessment criterion on the computational efficiency among different algorithms on different FJSS instances. In this paper, the detailed setting of programming platform, CPU equipped as well as average CPU computational time consumed by each algorithm are enclosed in order to analyze and roughly understand the efficiency difference. This method is generally applied in previous literature (Nasiri and Kianfar, 2012; Zhang et al., 2008) concerning assessment problem. The comparison with recent algorithms based on metrics consisting of the best makespan within the population (Best), the average makespan of the population (AVG) and the standard deviation of makespan of the population (SD) is summarized in Table 5.2 to assist the assessment.

The algorithm is implemented in Matlab on an Intel 2.60 GHz Core CPU with 8GB of RAM. The CPU computational time for solutions of each instance is recorded in separate tables. Instance 1, as a 3 jobs with 7 operations on 3 machines instance was analyzed to demonstrate the detailed simulation process. Several benchmark instance from Kacem *data* were tested in order to analyze the performance of the proposed algorithm in complex problem. The processing time tables and solutions for Kacem Instances are recorded in the appendix due to the massive size of data.

5.2 Simulation results

In order to obtain the best performance on different instances, the parameters are customized to adapt to each instance as shown in Table 5.1. The effect of these parameters is discussed in section 6.

Table 5.1 Parameter settings of HDE algorithms

Parameter	Description	Instance 1	Kacem Instance 1	Kacem Instance 2
NP	Population size	10	20	30
F	Scaling factor	0.1	0.1	0.1
CR	Crossover ratio	0.3	0.3	0.3
G_{max}	Maximum number of generations	30	50	200
P_l	Probability of local search	0.7	0.7	0.7
$iter_{max}$	Maximum local iterations	80	80	80
δ	Bound matrix	[0,1]	[0,1]	[0,1]

5.2.1 Instance 1 I_1 (3 jobs/7 operations/3 machines)

Instance 1 is a small size sample instance used in the previous sections to illustrate the process of the HDE algorithm. The processing time table for I_1 is shown in Table 2.1. Several possible schedules are obtained in Table A5, where M_m represents machine assignment, $O_{i,j}$ represents operation, C_{max} is makespan, S is start time and E is end time. As the population size is 10, 10 schedules with a makespan vector consists of 10 makespan value is obtained every generation. By recording the mean makespan and the minimum makespan within makespan vector in every generation, a convergence graph is plotted in order to illustrate the convergence performance as shown in Figure 5.1. In Figure 5.1, the blue curve demonstrates change of mean makespan and the red line is the minimum makespan in every generation. The circled data point (7, 9) is the convergence point i.e. all the solutions in this population have the same makespan 9.

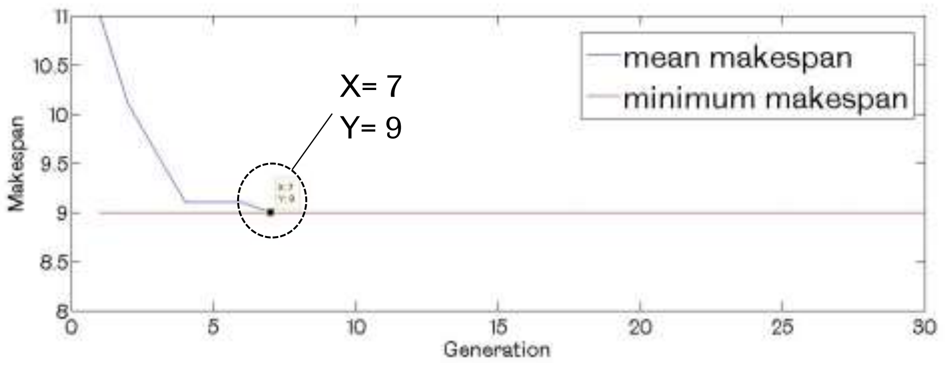


Figure 5.1 Illustration of convergence rate curve of Instance 1

As the two curves keep flat and coincident from the convergence point, the makespan 9 and its corresponding solutions are regarded as the best solutions to Instance 1.

5.2.2 Kacem Instance 1 (4 jobs/12 operations/5 machines)

This data set is a small size instance from Kacem et al. (2002), where the processing time table for Kacem Instance 1 & 2 can be found, with 4 jobs containing 12 operations in total which can be assigned on 5 machines. Several sample solutions are listed in the appendix.

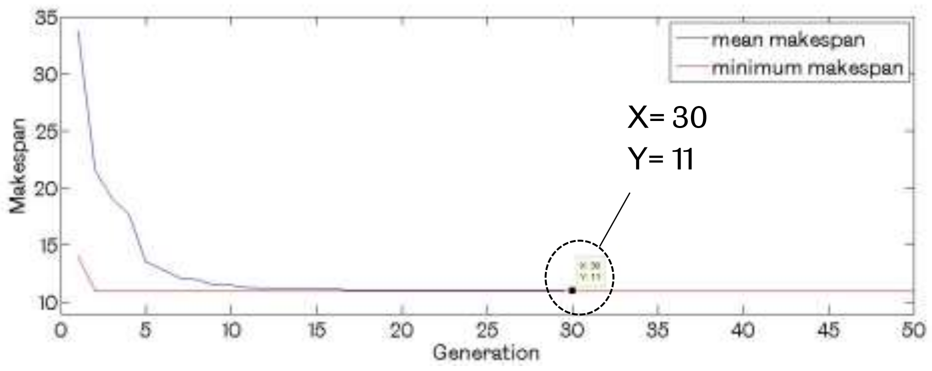


Figure 5.2 Illustration of convergence rate curve of Kacem Instance 1

From Figure 5.2, the best makespan is determined as 11 unit time. The curve converges at 30th generation where all the solutions in the population obtain the same makespan 11, which proves this set of solutions are reliable.

5.2.3 Kacem Instance 2 (10 jobs/29 operations/7 machines)

The Kacem Instance 2 is a normal size instance from Kacem et al. (2002). As a complex instance, the performance of the HDE algorithm is not stable. From several sample solutions listed in the appendix, the makespan does not always converges to the known lowest makespan 11. According to the second sample solution whose makespan is 12, a convergence rate curve is plotted in Figure 5.3. From Figure 5.3, it is found that the mean makespan curve does not converge with the minimum makespan curve, where the lowest value of the mean makespan is 13.13 and the best makespan is 12. In other words, most of solutions in this population has a makespan of 13 and some of them are even worse. Few solutions obtain the makespan of 12 which demonstrates that the proposed algorithm needs to search wider for a better solution as the potential for the local improvement appears to be very limited.

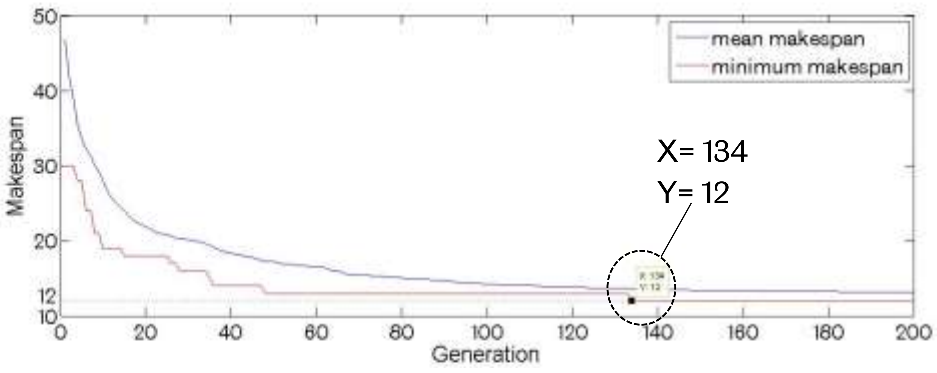


Figure 5.3 Illustration of convergence rate curve of Kacem Instance 2

5.3 Comparison with related algorithms

In order to have a clear understanding on the accuracy and efficiency of this HDE algorithm, four recently proposed algorithms including TSPCB of Li et al. (2011), BEDA of Wang et al. (2012), HDE-N1 and HDE-N2 of Hua and Yuan (2013) are compared with the purposed HDE algorithm and the results are listed in Table 5.2. To be noted, in the table, the size in the second column represents the instance size i.e. number of jobs * number of machines, and the BKS represents best known makespan of the corresponding instance from recent literatures.

It can be discovered that HDE algorithm has a stable performance compared to other four algorithms when they process a normal size instance. The HDE algorithm even has a better efficiency compared to TSPCB and BEDA when they are processing the complex instance. However, the accuracy and reliability of HDE algorithm on the large size instance is not satisfactory compared with other algorithms. All five algorithms are able to find the BKS of two Kacem instances.

Table 5.2 Comparison between HDE with other well-known algorithms on Kacem instances

			HDE				TSPCB			
Instance	Size	BKS	Best	AVG	SD	CPU_{av}	Best	AVG	SD	CPU_{av}
Case 1	4 * 5	11	11	11.00	0	0.11	11	11.00	-	0.05
Case 2	10 * 7	11	11	13.09	0.61	0.30	11	11.00	-	5.21

BEDA				HDE-N_1				HDE-N_2			
Best	AVG	SD	CPU_{av}	Best	AVG	SD	CPU_{av}	Best	AVG	SD	CPU_{av}
11	11.00	0	0.01	11	11.00	0	0.06	11	11.00	0	0.09
11	11.00	0	0.30	11	11.00	0	0.19	11	11.00	0	0.46

6 DISCUSSION

In this section, the further research on several problems is demonstrated. In section 6.1, two algorithms were tested separately in order to reflect their effect in HDE algorithm. Then, further analysis was conducted to demonstrate the effect of different parameters in HDE and assess the performance of HDE based on the criterion of stability & reliability. In section 6.3, the research on self-adaptive algorithm is demonstrated for further improvement. Lastly, the limitations of purposed HDE algorithm is concluded.

6.1 Pure DE versus pure local search

In order to understand the individual effect of DE and local search, pure DE algorithm and pure local search was implemented respectively. At first, pure DE was implemented to solve Instance 1. As shown in Figure 6.1, the convergence rate decreases rapidly compared to the convergence speed in Figure 5.1. However, DE is still capable of finding the optimal solution and converging the solutions to the best solution although more generations are needed compared to HDE. It also takes 4 generations for DE to figure out the best makespan, which is inefficient compared with performance of pure local search shown in Figure 6.2.

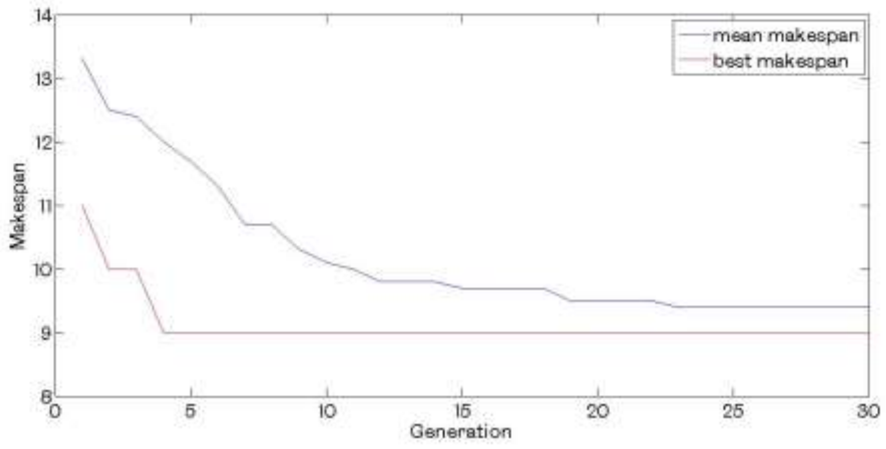


Figure 6.1 Illustration of convergence rate curve under pure DE

On the contrary, the local search is only responsible for local improvement on the schedules as mentioned before, so the solutions keep constant after the 2nd generation i.e. every solution in the population represents the best possible schedule local search can generate. When it comes to a local optimum, an alternative strategy such as DE algorithm should be applied in order to perturb the population for further possible improvement through local search to achieve the global optimum.

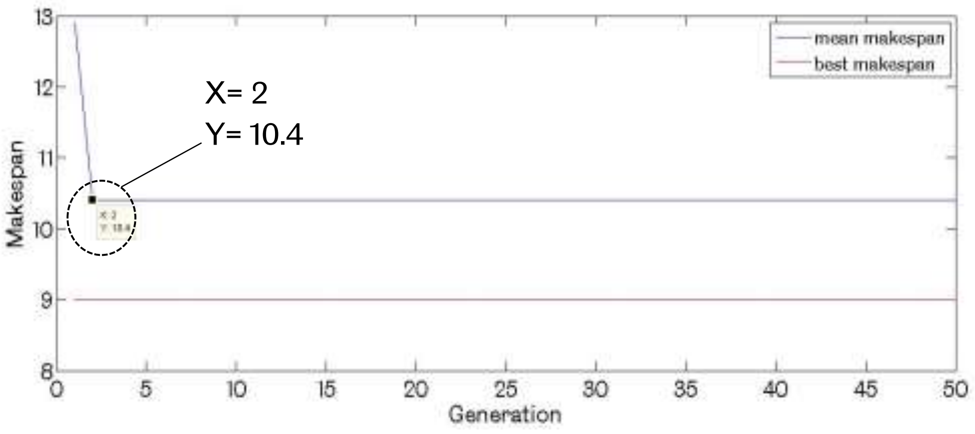


Figure 6.2 Illustration of convergence rate curve under pure local search

6.2 Further performance analysis

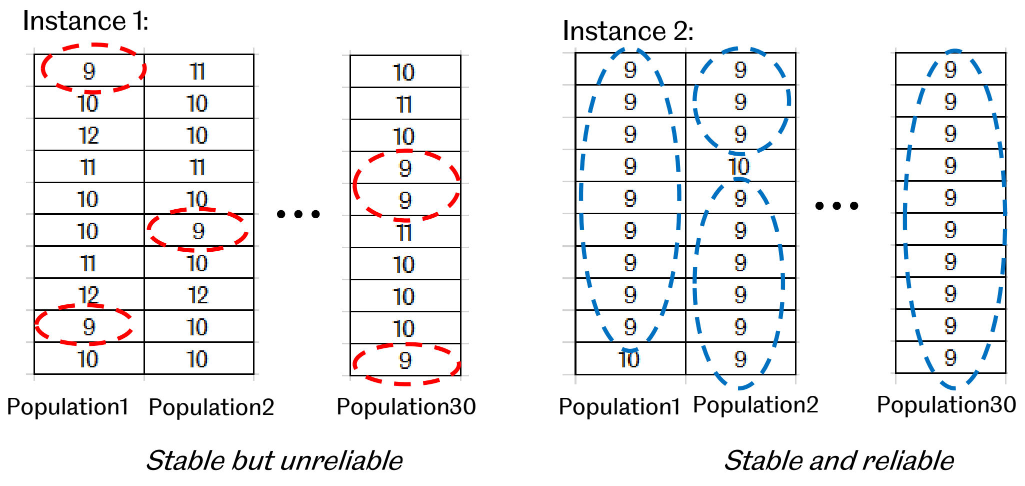
In this sub-section, further analysis on the performance of HDE algorithm is preformed based on the assessment of stability and reliability. In addition, the influence of different controlling parameters in HDE is also discussed.

6.2.1 Stability & Reliability

According to the randomness of the initialization of population, the stability and reliability of algorithm on different instances become an important criterion to assess the performance of the algorithm. The definition of stability of the algorithm in this paper is whether the algorithm can always find the optimal solution over a specified iterations with an appropriate parameter setting, despite the frequency of finding it in each population. On the contrary, the reliability of the algorithm depends on how often the algorithm can obtain the optimal solution in each population. A brief illustration of the two criterions is shown in Figure 6.3. For instance 1 in Figure 6.3, the best known makespan 9 appears in all the population but not frequently so the algorithm for instance 1 is stable but unreliable. For instance 2 in Figure 6.3, almost all the solutions obtain the makespan of 9, which demonstrates the algorithm is stable and reliable for instance 2.

After the purposed HDE algorithm run 30 independent times for each instance, the average value of lowest mean makespan and the average best makespan of 30 iterations are involved in Table 6.1 to represent the reliability and stability respectively. The metric of 'Best Known makespan' is the best known solution to the corresponding instance from reviewing recent FJSS literatures. Besides, the size of instance is listed to distinguish the complexity of different instances.

Assume two instances with a same BKM of **9**. After the algorithm runs 30 independent times to solve these instances with standard setting, following set of solutions* are obtained:



*27 sets of solutions are omitted for convenience.

Figure 6.3 Illustration of stability & reliability of algorithm

Table 6.1 Comparison between results of proposed HDE algorithm with BKM

Instance	Size	Average Mean makespan	Average Best makespan	Best Known makespan (BKM)
Instance 1	3 jobs 7 operations 3 machines	9	9	9
Kacem Instance 1	4 jobs 12 operations 5 machines	11.15	11	11
Kacem Instance 2	10 jobs 29 operations 7 machines	13.86	12.13	11

In Table 6.1, the purposed algorithm shows perfect performance on stability and reliability on Instance 1, which owing to the low complexity of the problem.

As for the Kacem Instance 1, the average mean makespan starts floating around 11.15 as shown in Figure 6.4, where the algorithm obtains a consistent solution for about 1/3 times within 30 iterations i.e. all the solutions in those populations obtain the same makespan which is exactly the best makespan. In addition, the value of the average mean makespan is quite close to the BKM which illustrates that most of the solutions in a population are able to reflect the best makespan. So the purposed algorithm is generally reliable for Kacem Instance 1. The algorithm shows outstanding performance on stability as the average best makespan is always equal to BKM of this instance.

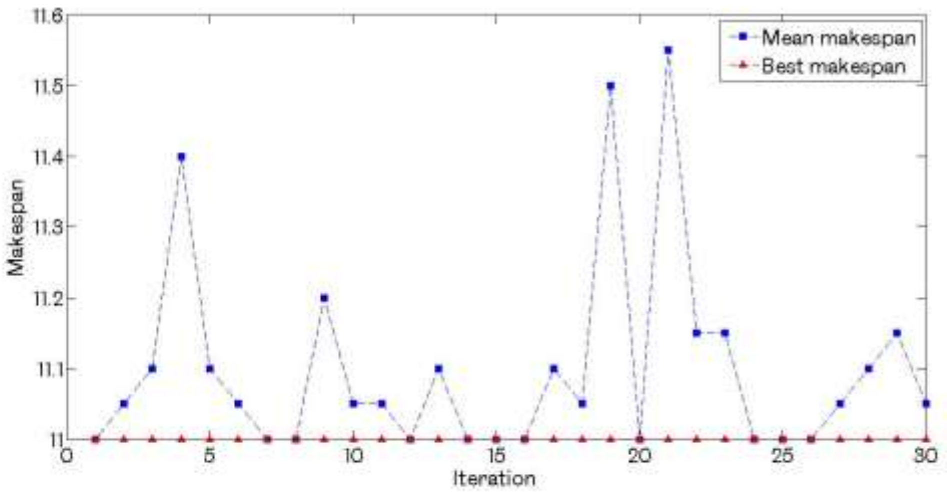


Figure 6.4 Stability & reliability of proposed algorithm on Kacem Instance 1

As a large size instance, the purposed algorithm performs poorer on Kacem Instance 2 than the other two instances as shown in Figure 6.5, due to the high complexity and rarity of the best solution. Compared to small size instances, it is more difficult for the algorithm approaching the optimal region which is now quite narrow. The effectiveness of the local search is also substantially weakened when the solution is close to the optimal region as shown in Figure 5.3, where the minimum makespan is almost not enhanced from 50 generations to about 130 generations. In Figure 6.5, the mean makespan and the best makespan are vibrating around 14 and 12 respectively, which demonstrates poor stability and reliability. However, there are few successful attempts where the best known makespan is approached by the algorithm, which means that there is potential for further improvement.

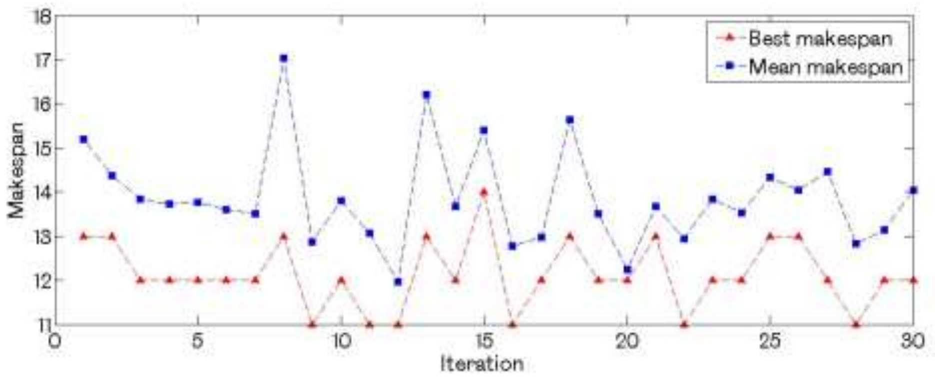


Figure 6.5 Stability & reliability of proposed algorithm on Kacem Instance 2

6.2.2 Effects of parameters

In order to investigate the effect of different parameters on HDE algorithm, sets of experiment were carried out on Instance 1 to determine the optimal parameter setting for a particular instance. Every set of parameter combination was tested for 30 independent times. Except for the tested parameters, all the other parameters remains same as the data shown in Table 5.1.

Table 6.2 Influence of Population size NP and Maximum generation G_{max} on HDE

	G_{max} = 5			G_{max} = 10			G_{max} = 20			G_{max} = 30		
NP	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av
5	9.53	0.33	0.04	9.39	0.07	0.04	9.38	0.07	0.04	9.2	0	0.03
10	9.39	0.53	0.11	9.17	0.22	0.08	9.02	0.06	0.06	9	0	0.05
20	9.31	0.52	0.18	9.05	0.18	0.14	9.01	0.04	0.10	9	0	0.09
30	9.27	0.50	0.25	9.09	0.26	0.20	9.02	0.09	0.22	9	0	0.14
*Best makespan for this instance is 9.												

Generally, with the expansion of population size and number of generations, more solutions in the population are able to determine the lowest possible makespan of the test instance as shown in Table 6.2. When the algorithm runs for limited generations, the rising population size raises the possibility of generating better solutions in the population initially, which keeps ideal solutions stay in the population through the evolution to accelerate the optimization on the whole population. Besides, in the initial phase, the convergence of results highly depends on the performance of local search as shown in previous sections so when the total generations for the algorithm is limited, local search can ensure that the solution will get close to the optimal region at an acceptable convergence rate. As the generation number increasing, it turns out that the DE algorithm plays a more important role than local search in generating a better solution through the evolution when the convergence rate is slowing down and stuck until the convergence point is approached. More optimal solutions appear in the population with the increasing generation number so that the standard derivation drops down i.e. the solution becomes stable. As for the efficiency, large max generations consume less average CPU time but the total computational time increases. Similarly, the algorithm consumes more time with the increasing NP. For Instance 1, solutions normally converge at about 20 generations and at least a population size of 10 is required for an ideal convergence rate.

Table 6.3 Influence of parameters P_l and $iter_{max}$ on HDE

	$iter_{max} = 10$			$iter_{max} = 40$			$iter_{max} = 80$			$iter_{max} = 100$		
P_l	AVG	SD	CPU $_{av}$	AVG	SD	CPU $_{av}$	AVG	SD	CPU $_{av}$	AVG	SD	CPU $_{av}$
0.1	9.22	0.26	0.01	9.16	0.23	0.01	9.13	0.21	0.01	9.13	0.10	0.01
0.3	9.04	0.09	0.01	9.07	0.05	0.02	9.06	0.07	0.01	9.03	0.08	0.02
0.5	9.03	0.06	0.02	9.02	0.04	0.03	9.01	0.03	0.04	9.01	0.02	0.06
0.7	9.01	0.02	0.02	9.01	0.02	0.04	9	0	0.05	9	0	0.09
0.9	9.01	0.02	0.04	9	0	0.05	9	0	0.08	9	0	0.09
*Best makespan for this instance is 9.												

The local search probability P_l and maximum iteration $iter_{max}$ controls the quality of local search algorithm and the efficiency of the whole HDE algorithm. From Table 6.3, the convergence rate is clearly in a linear relationship with P_l and $iter_{max}$ respectively. With a high probability of local search, a deep search for local improvement on the schedules is proceeded and it can be basically completed within about 10 iterations for small size instances like Instance 1. There is no obvious improvement by adjusting the parameters of local search because as mentioned previously, DE algorithm replaces local search to become the main role of optimization. In addition, there is possibility of acyclic search during the local search as we discussed in section 4.6.3. When the program encounters acyclic situation, the local search will be interrupted, which makes large number of iterations meaningless. Therefore, the acyclic search will possibly have a negative influence when the large number of iterations is truly needed in the large size instances. Lastly, due to the complexity of local search, it becomes a burden for the efficiency of HDE algorithm if the P_l and $iter_{max}$ are excessive, which highly increases the CPU computational time. For Instance 1, P_l

= 0.7 and $iter_{max}$ = 80 is proved to be an appropriate setting.

Table 6.4 Influence of parameters F and Cr on HDE for Instance 1

	Cr = 0.1			*Cr* = 0.3			*Cr* =0.5			*Cr* = 0.7		
F	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av
0.1	9.17	0.33	0.07	9.	0	0.03	9	0	0.03	9.05	0	0.05
0.3	9.23	0.38	0.06	9.06	0.03	0.06	9.05	0	0.06	9.05	0	0.07
0.5	9.21	0.35	0.06	9.03	0.11	0.07	9	0	0.06	9	0	0.06
0.7	9.27	0.44	0.05	9.03	0.07	0.07	9.06	0.02	0.07	9	0	0.07
*Best makespan for this instance is 9.												

As we mentioned in section 3, the crossover ratio and the scaling factor are generally selected between 0.6 and 0.8 in order to obtain an ideal searching area for DE algorithm. It is proved by Table 6.4 that the algorithm performs well when the parameters are set between 0.5 ~ 0.7. However, it is worth noting that the algorithm demonstrates an outstanding performance as well when *F* is set as 0.1 and *Cr* is set within 0.3 to 0.5. In order to distinguish the effect of these two different setting, this experiment is also conducted on Kacem Instance 1 & 2 and the result is recorded in Table 6.5 and Table A7 in appendix.

Table 6.5 Influence of parameters F and Cr on HDE for Kacem Instance 2

	Cr = 0.1			*Cr* = 0.3			*Cr* =0.5			*Cr* = 0.7		
F	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av
0.1	15.18	2.24	0.28	13.28	0.99	0.29	13.09	0.61	0.30	15.17	0.81	0.28
0.3	17.27	2.53	0.28	15.77	1.75	0.30	15.69	1.32	0.32	15.82	0.98	0.32
0.5	18.53	2.40	0.24	18.01	1.87	0.26	18.78	1.65	0.28	20.28	1.43	0.29
0.7	18.45	2.29	0.27	19.82	2.06	0.32	20.78	1.67	0.31	23	1.73	0.31
*Best makespan for this instance is 11.												

According to Table 6.5, compared with the efficient performance in the last instance, the general setting of *F* and *Cr* (0.5~0.7) in the shadowed block leads to performance deterioration. However, the highlighted setting of *F* = 0.1, *Cr* = 0.5 demonstrates a better performance compared to other settings. The possible reason why the setting for FJSS problem is distinct with the general setting can be concluded as the requirement for inheriting the best solutions instead of expanding the searching region. In other words, in large size FJSS instance, by focusing on the local improvement with a small *F* under the assistance of local search, an outstanding convergence rate can be achieved to enhance the performance of HDE algorithm compared to focusing on global improvement with a large *F*.

6.3 Self-adaptive algorithm

As discussed in the previous sections, the performance of the HDE algorithm could be improved by applying self-adaptive algorithm, which adjusts the appropriate parameters and searching strategy to adapt to the specific instance as the suitable parameters change in different instances. In the purposed algorithm in this paper, the self-adaptive algorithm is not involved as the focus of previous work was on implementing the HDE algorithm, where the time consumed was underestimated due to the massive information involved in HDE. Therefore, considerable work was done so that the time for implementing the self-adaptive algorithm and processing computational simulation was not enough. However, the research on self-adaptive algorithm is briefly discussed in this sub-section and further progress could be made in the future to apply this algorithm based on the purposed algorithm.

It was emphasized in many previous DE algorithm related literatures (Rogalsky et al., 1999; Gaemperle et al., 2002) that the searching strategy and the parameter settings are key factors on the performance of DE algorithm, which both consume massive effort to determine the suitable setting for a specific instance. Although the chosen strategy and parameters is suitable for the global improvement, the failure of improvement may occurs in process where most of the chromosomes in the population are not successfully improved by DE and the convergence rate is retarded.

In order to further improve the performance of DE, the real-time adjustment of searching strategy and the parameters was purposed in several literatures (Liu and Lampinen, 2005; Qin and Suganthan, 2005). According to Qin and Suganthan (2005), the theory of their Self-Adaptive DE algorithm (SADE) is 'to probabilistically select one out of several available learning strategies and apply to the current population'. Initially, the probability of selecting one from a set of mutation strategy is equal, which means every strategy have the same chance to be applied on each chromosome in the initial population. After a complete generation, the result of whether the chromosome is improved by corresponding strategy will be recorded through the evaluation and selection phase so that a successful rate for each strategy can be determined after a specified number of generations, which is called ‘learning period’ in their paper. Based on this successful rate, the strategy can be automatically adjusted to adapt to the progress of convergence. As for the parameters setting *F* and *Cr*, their adaptive rule is illustrated in Figure 6.6, which is based on ensuring that the parameters are generated with a normal distribution within a specified range.

Through the purposed rule, the algorithm can ensure its ability on both global search and local search based on a floating *F*, and the performance of the algorithm keeps consistently when *Cr* is changing in a proper range. Associated with the discussion of stability and reliability, the stability is improved by an adaptive *F* which enhances the ability to approach the global optimum, while the adaptive adjustment to *Cr* ensures the consistent performance in population in order to enhance the reliability of the algorithm.

In the case of FJSS, as the scaling factor is set as a small value in large size instances, the ability on searching for global improvement is actually limited throughout the process, while the local improvement ability is emphasized instead. In addition, as the highlighted standard derivation shows in the Table 6.5, the performance of the algorithm is not consistent within a generation, which is dependent on the crossover ratio. The performance is even deteriorated in several specific population with a fixed *Cr* as shown in Figure 6.5. When the mutation strategy and the scaling factor are self-adapted, the convergence speed could be improved without sacrificing the global exploration ability, which is significant to avoid trapping in a local optimum. A self-adaptive *Cr* is also beneficial to control the performance quality. By implementing SADE algorithm, the maximum generations required for convergence could also be reduced, where further simulations is needed to investigate the detailed influence and improvement.

In conclusion, the effects of the DE parameters and mutation strategy are fully utilized by self-adaptive algorithm in order to improve the performance of DE algorithm. Based on different characteristic and complexity of specific problems, the parameters *F* and *Cr* as well as the learning strategy will be self-adapted under the circumstance of being trapped in a local optimum.

Assuming max generation is 50, NP is 5.

For scaling factor *F*, randomly generate a number in the range (0, 2] with normal distributions of mean 0.5 and standard deviation 0.3 for each individual chromosome. *F* keeps changing in every generation.

$$\begin{tikzcd} F & 0.17 \arrow[d] & 0.51 \arrow[d] & 0.67 & 0.83 & 0.96 \\ & F_1, X_{1,j} & F_2, X_{2,j} & \cdots & & \end{tikzcd}$$

For crossover ratio *Cr*, randomly generate a number in the range (0, 1] with normal distributions of mean Cr_{mean} 0.5 and standard deviation 0.1 for each individual chromosome. This set of *Cr* remains same for 5 generations.

▨ 0.51	▨ 0.35	0.43	0.39	▨ 0.74

$Cr_1, X_{1,j}$ $Cr_2, X_{2,j}$ ...

After 5 generations, regenerate a set of *Cr* under the same condition.

▨ 0.44	0.57	0.48	▨ 0.59	0.42

For each *Cr*, record its value if its corresponding trial vector enters next generation. (Values in shadowed block)

Until the 10th generation, *Cr* has been changed for 2 times. Recalculate Cr_{mean} of all the recorded values. Regenerate *Cr* with the new Cr_{mean} and standard deviation 0.1 for next 5 generations and empty the records as well as Cr_{mean} for the new recording in the 20th generation. Repeat previous steps till the end.

Figure 6.6 Illustration of self-adaptive process for *F* and *Cr*

6.4 Limitations on purposed algorithm

According to the previous computational results and discussion, the purposed HDE algorithm has following limitations to improve in the future. Firstly, although the HDE algorithm can solve the normal size instances like Instance 1 and Kacem Instance 1 smoothly, it performed unsatisfactorily on large size instance like Kacem Instance 2. Despite that there were several successful attempts obtaining the best known makespan, the solutions are unstable in general, which will disturb the decision-making on scheduling and slowing down the manufacturing efficiency. Secondly, cyclic search is not completely avoided, which also causes low efficiency of the algorithm in large size instances. Thirdly, the performance of HDE algorithm could be improved by importing self-adaptive algorithm, which is able to adjust the learning strategy and parameters instead of determining fixed suitable parameters through massive tests.

7 CONCLUSION

This paper purposed a hybrid differential evolution algorithm (HDE) for solving the flexible job shop scheduling (FJSS) problem. By combining local search algorithm with the basic differential evolution algorithm, the searching efficiency is improved compared to pure DE algorithm. For small size of FJSS instances, the performance of HDE is well enough to consistently achieve a reliable best makespan, while for large size of FJSS instances which contains over 10 jobs and 30 operations on 7 machines, the performance is not satisfactory due to several potential defects, including probability of cyclic search and fixed controlling parameters. In addition, the further improvement on HDE could be proceeded based on the research on self-adaptive differential evolution (SADE), where the theory and methodology is discussed.

REFERENCE

1. **Garey, M. R., Johnson, D. S., & Sethi, R.** The complexity of flowshop and job-shop scheduling. Mathematics of Operations Research, 1, 117–129, 1976.
2. **Jain, A. S., & Meeran, S.** Deterministic job-shop scheduling: Past, present and future. *European journal of operational research*, *113*(2), 390-434, 1999.
3. **Applegate, D., Bixby, R.E., Chvátal, V., and Cook, W.** The Traveling Salesman Problem: A Computational Study, Princeton University Press, Princeton, USA, 2006.
4. **Bruker, P., & Schlie, R.** Job-shop scheduling with multi-purpose machines. Computing, 45, 369–375, 1990.
5. **Brandimarte, P**. Routing and scheduling in a flexible job shop by taboo search. Annals of Operations Research, 41, 157–183, 1993.
6. **Hurink, E., Jurisch, B., & Thole, M.** Tabu search for the job shop scheduling problem with multi-purpose machines. Operations Research Spektrum, 15, 205–215, 1994.
7. **Dauzere-Peres, S., & Paulli, J.** An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. Annals of Operations Research, 70, 281–306, 1997.
8. **Mastrolilli, M., & Gambardella, L. M.** Effective neighborhood functions for the flexible job shop problem. Journal of Scheduling, 3(1), 3–20, 2002.
9. **Kacem, I., Hammadi, S., & Borne, P.** Pareto-optimality approach for flexible job-shop scheduling problems: Hybridization of evolutionary algorithms and fuzzy logic. Mathematics and Computers in Simulation, 60(3-5), 245–276, 2002.
10. **Jia, H., Nee, A., Fuh, J., & Zhang, Y.** A modified genetic algorithm for distributed scheduling problems. Journal of Intelligent Manufacturing, 14(3), 351–362, 2003.
11. **Xia, W., & Wu, Z.** An effective hybrid optimization approach for multiobjective flexible job-shop scheduling problems. Computers & Industrial Engineering, 48(2), 409–425, 2005.
12. **Moslehi, G., & Mahnam, M.** A Pareto approach to multi-objective flexible job-

shop scheduling problem using particle swarm optimization and local search. International Journal of Production Economics, 129(1), 14–22, 2011.

13. **Storn, R., & Price, K.** Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization, 11(4), 341–359, 1997.

14. **Storn, R.** Differential evolution design of an IIR-filter. In Proceedings of 1996 IEEE International Conference on Evolutionary Computation (pp. 268–273). IEEE, 1996.

15. **Manson, G, Worden, K.** *Lamb-wave sensor optimization using differential evolution*, Smart Structures and Materials 2001: Modeling, Signal Processing, and Control in Smart Structures, 570, 2001.

16. **Ilonen, J., Kamarainen, JK. & Lampinen, J.** Differential Evolution Training Algorithm for Feed-Forward Neural Networks, Neural Processing Letters, 2003.

17. **Lakshminarasimman, L. Subramanian, S.** Applications of Differential Evolution in Power System Optimization, Advances in Differential Evolution, Springer Berlin Heidelberg, pp. 257-273, 2008.

18. **Gonzalez, C., Blanco, D. & Moreno, L.** Optimum robot manipulator path generation using Differential Evolution, Evolutionary Computation, 2009.

19. **Di Maio, F., Baronchelli, S., & Zio, E.** Hierarchical differential evolution for minimal cut sets identification: Application to nuclear safety systems. European Journal of Operational Research, 238(2), 645–652, 2014.

20. **Kazemipoor, H., Tavakkoli-Moghaddam, R., Shahnazari-Shahrezaei, P. and Azaron, A.** A differential evolution algorithm to solve multi-skilled project portfolio scheduling problems. *The International Journal of Advanced Manufacturing Technology*, pp.1-13, 2013.

21. **Yuan, Y., Hua, X.** Flexible job shop scheduling using hybrid differential evolution algorithms, Computers & Industrial Engineering, Volume 65, Issue 2, pp.246–260, 2013.

22. **Brest, J., & Maučec, M. S.** Control parameters in self-adaptive differential evolution, 2006.

23. **Liu, J. & Lampinen, J.** Soft Comput. 9: 448. doi:10.1007/s00500-004-0363-

x, 2005.

24. **Wang, L., Pan, Q., & Fatih Tasgetiren, M.** Minimizing the total flow time in a flow shop with blocking by using hybrid harmony search algorithms. Expert Systems with Applications, 37(12), 7929–7936, 2010.

25. **Nasiri, M., & Kianfar, F.** A GES/TS algorithm for the job shop scheduling. Computers & Industrial Engineering, 62(4), 946–952, 2012.

26. **Zhang, C., Li, P., Rao, Y., & Guan, Z.** A very fast TS/SA algorithm for the job shop scheduling problem. Computers & Operations Research, 35(1), 282–294, 2008.

27. T. **Rogalsky, R.W. Derksen, and S. Kocabiyik.** "Differential Evolution in Aerodynamic Optimization," In: Proc. of 46 Annual Conf of Canadian Aeronautics and Space Institute. 29-36, 1999.

28. **R. Gaemperle, S. D. Mueller and P. Koumoutsakos,** "A Parameter Study for Differential Evolution", A. Grmela, N. E. Mastorakis, editors, Advances in Intelligent Systems, Fuzzy Systems, Evolutionary Computation, WSEAS Press, pp. 293-298, 2002.

29. **Qin, A. K., & Suganthan, P. N.** Self-adaptive differential evolution algorithm for numerical optimization. In Evolutionary Computation, 2005. The 2005 IEEE Congress on (Vol. 2, pp. 1785-1791). IEEE, 2005.

30. **Wang, L., Pan, Q., & Fatih Tasgetiren, M.** Minimizing the total flow time in a flow shop with blocking by using hybrid harmony search algorithms. Expert Systems with Applications, 37(12), 7929–7936, 2010.

APPENDIX

Table A1 Operation time table for an improved solution to Instance 1

Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	0	2
6 ($O_{3,1}$)	2	0	5
3 ($O_{2,1}$)	3	0	3
4 ($O_{2,2}$)	3	3	7
2 ($O_{1,2}$)	1	2	6
7 ($O_{3,2}$)	1	6	9
5 ($O_{2,3}$)	3	7	9

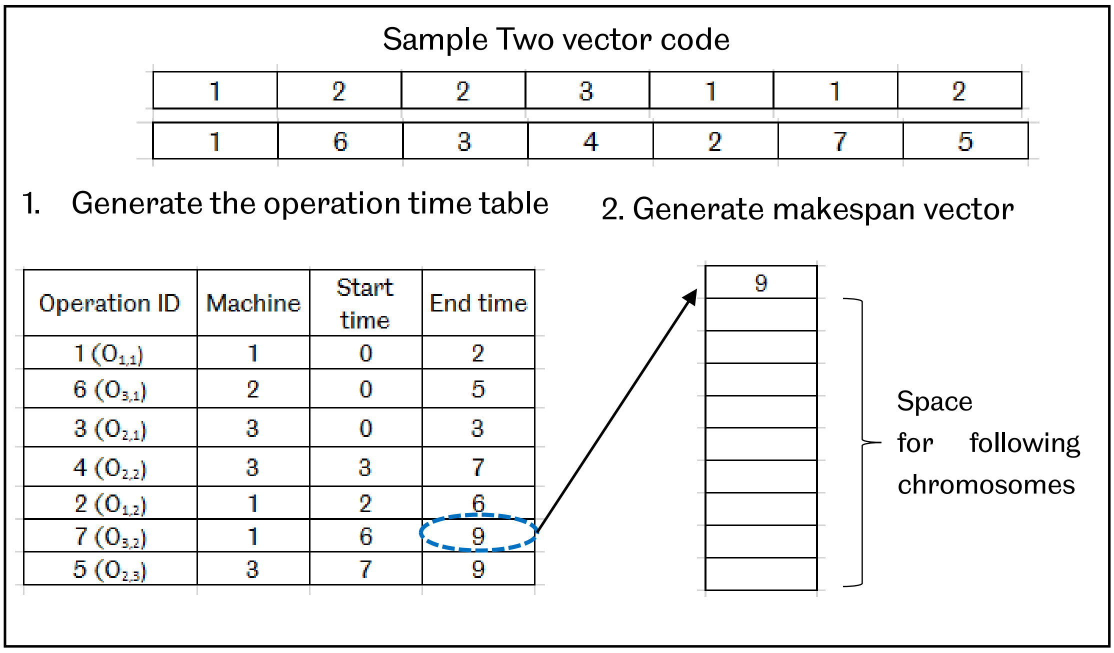
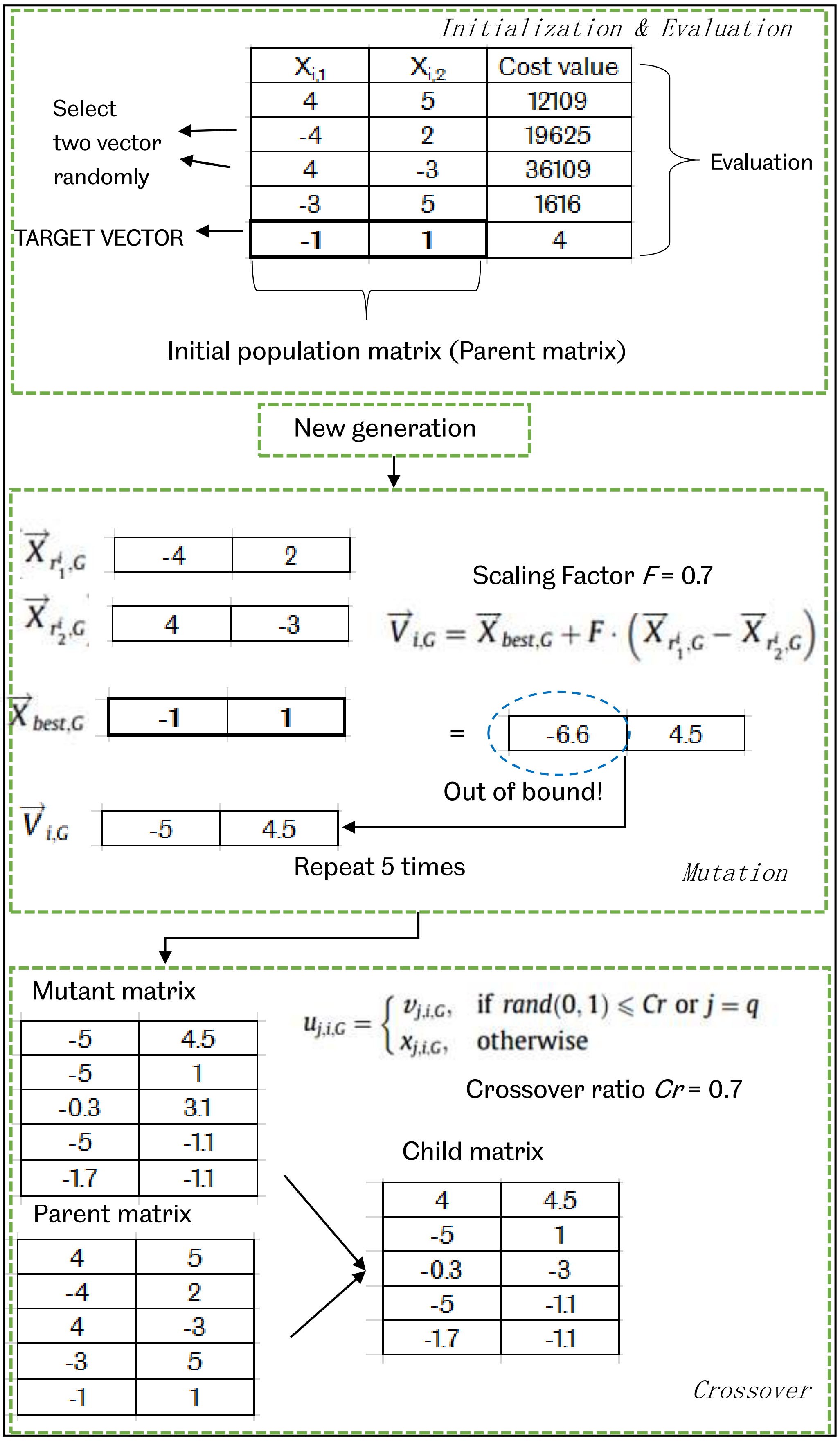


Figure A1 Illustration of evaluation process

Table A2 Sample population matrix and its cost value of Rosenbrock function

$X_{i,1}$	$X_{i,2}$	Cost value
4	5	12109
-4	2	19625
4	-3	36109
-3	5	1616
-1	1	4



$X_{i,1}$	$X_{i,2}$	Cost value
4	5	12109
-4	2	19625
4	-3	36109
-3	5	1616
-1	**1**	4

$\vec{X}_{r_1^i,G}$	-4	2
$\vec{X}_{r_2^i,G}$	4	-3
$\vec{X}_{best,G}$	**-1**	**1**
$\vec{V}_{i,G}$	-5	4.5

$$\vec{V}_{i,G} = \vec{X}_{best,G} + F \cdot \left(\vec{X}_{r_1^i,G} - \vec{X}_{r_2^i,G}\right)$$

= -6.6 | 4.5

$$u_{j,i,G} = \begin{cases} v_{j,i,G}, & \text{if } rand(0,1) \leqslant Cr \text{ or } j = q \\ x_{j,i,G}, & \text{otherwise} \end{cases}$$

Mutant matrix

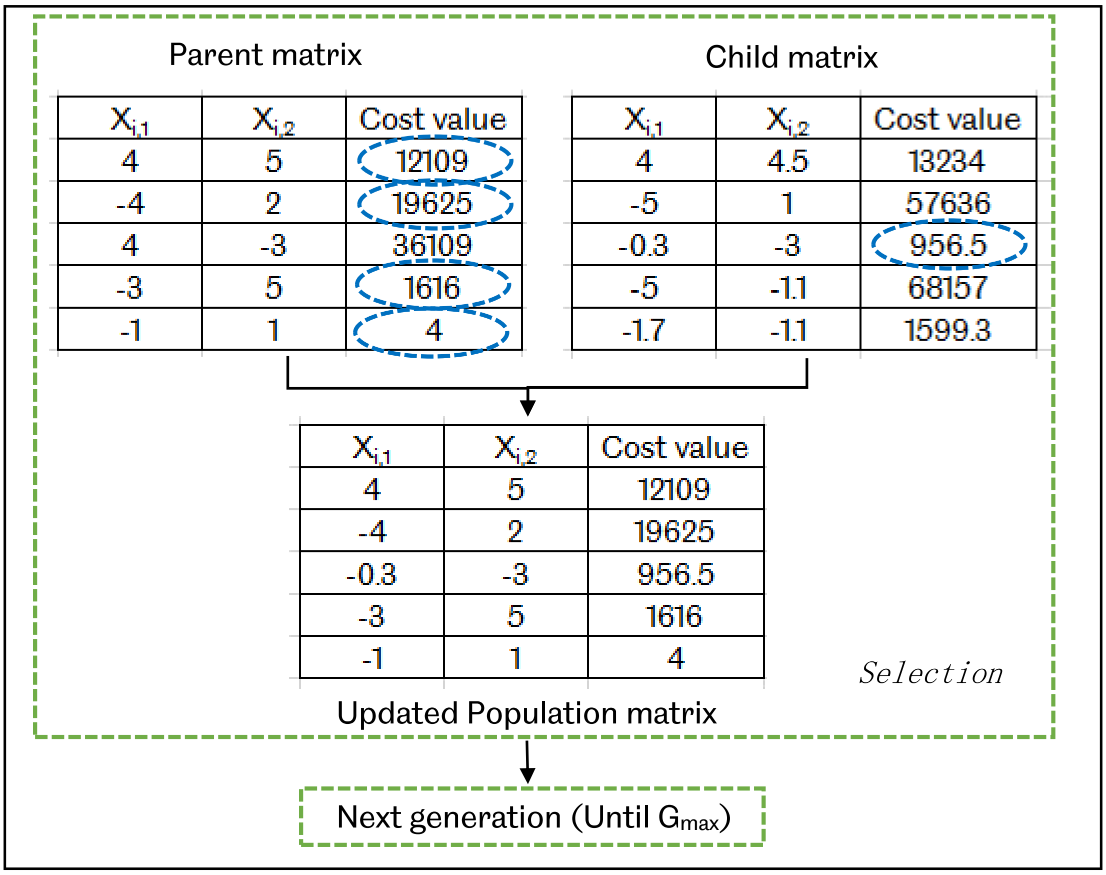
-5	4.5
-5	1
-0.3	3.1
-5	-1.1
-1.7	-1.1

Parent matrix

4	5
-4	2
4	-3
-3	5
-1	1

Child matrix

4	4.5
-5	1
-0.3	-3
-5	-1.1
-1.7	-1.1



Parent matrix

$X_{i,1}$	$X_{i,2}$	Cost value
4	5	12109
-4	2	19625
4	-3	36109
-3	5	1616
-1	1	4

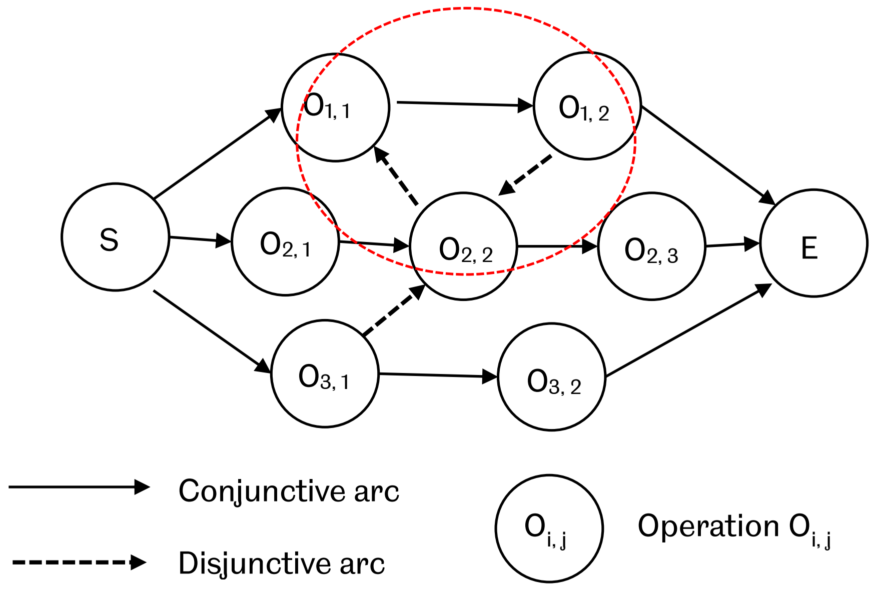
Child matrix

$X_{i,1}$	$X_{i,2}$	Cost value
4	4.5	13234
-5	1	57636
-0.3	-3	956.5
-5	-1.1	68157
-1.7	-1.1	1599.3

$X_{i,1}$	$X_{i,2}$	Cost value
4	5	12109
-4	2	19625
-0.3	-3	956.5
-3	5	1616
-1	1	4

Updated Population matrix

Figure A2 Flow chart of basic DE algorithm



* Weight of nodes is omitted.

* $O_{1,1}$ $O_{1,2}$ $O_{2,2}$ forms a cyclic path.

Figure A3 Illustration of cyclic path in the disjunctive graph

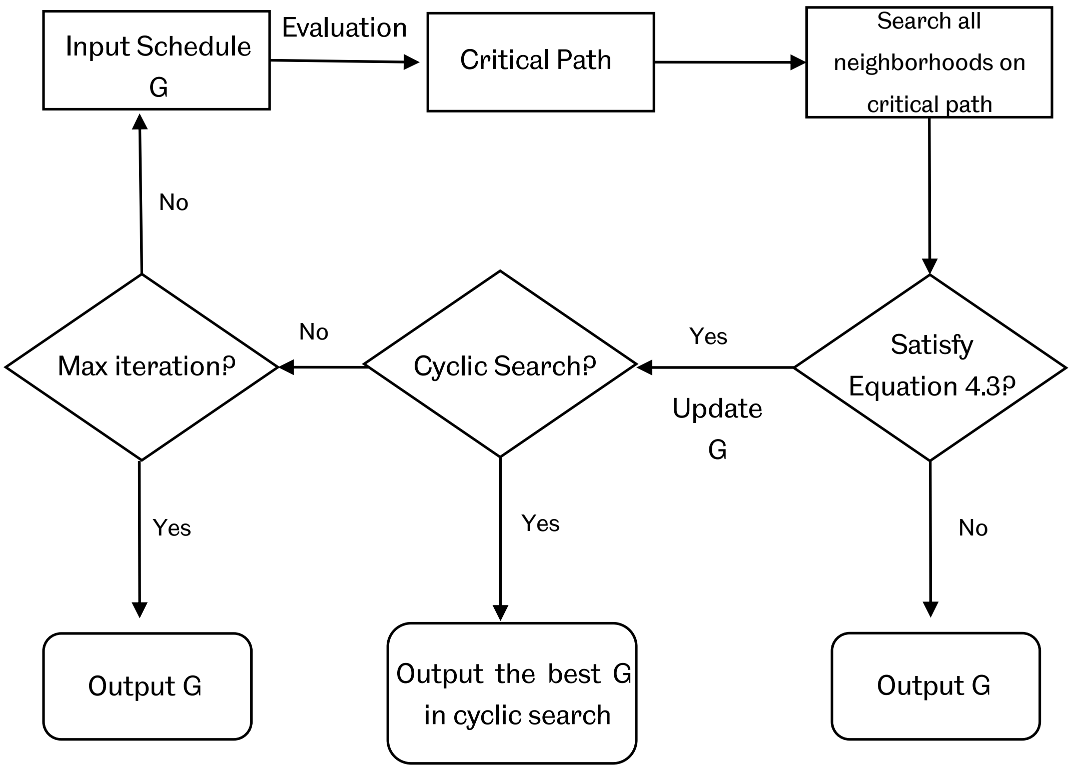


Figure A4 Flow chart of local search

Operation time table

Operation ID	Machine	Start time	End time
3 ($O_{2,1}$)	3	0	3
1 ($O_{1,1}$)	1	0	2
6 ($O_{3,1}$)	1	2	6
7 ($O_{3,2}$)	1	6	9
4 ($O_{2,2}$)	2	3	8
5 ($O_{2,3}$)	1	9	12
2 ($O_{1,2}$)	3	3	6

Latest operation time table

Operation ID	Machine	Start time	End time
3 ($O_{2,1}$)	3	1	4
1 ($O_{1,1}$)	1	0	2
6 ($O_{3,1}$)	1	2	6
7 ($O_{3,2}$)	1	6	9
4 ($O_{2,2}$)	2	4	9
5 ($O_{2,3}$)	1	9	12
2 ($O_{1,2}$)	3	9	12

'Required' makespan

1. Find the 'required' makespan from operation time table.
2. Fill in the new schedule from the last operation of each job by setting the end time as the 'required' makespan.
3. Remember operations on the same machine need to be performed one by one!
4. Complete the schedule by the given operation order and processing time.

Figure A5 Illustration of derivation of latest operation time table

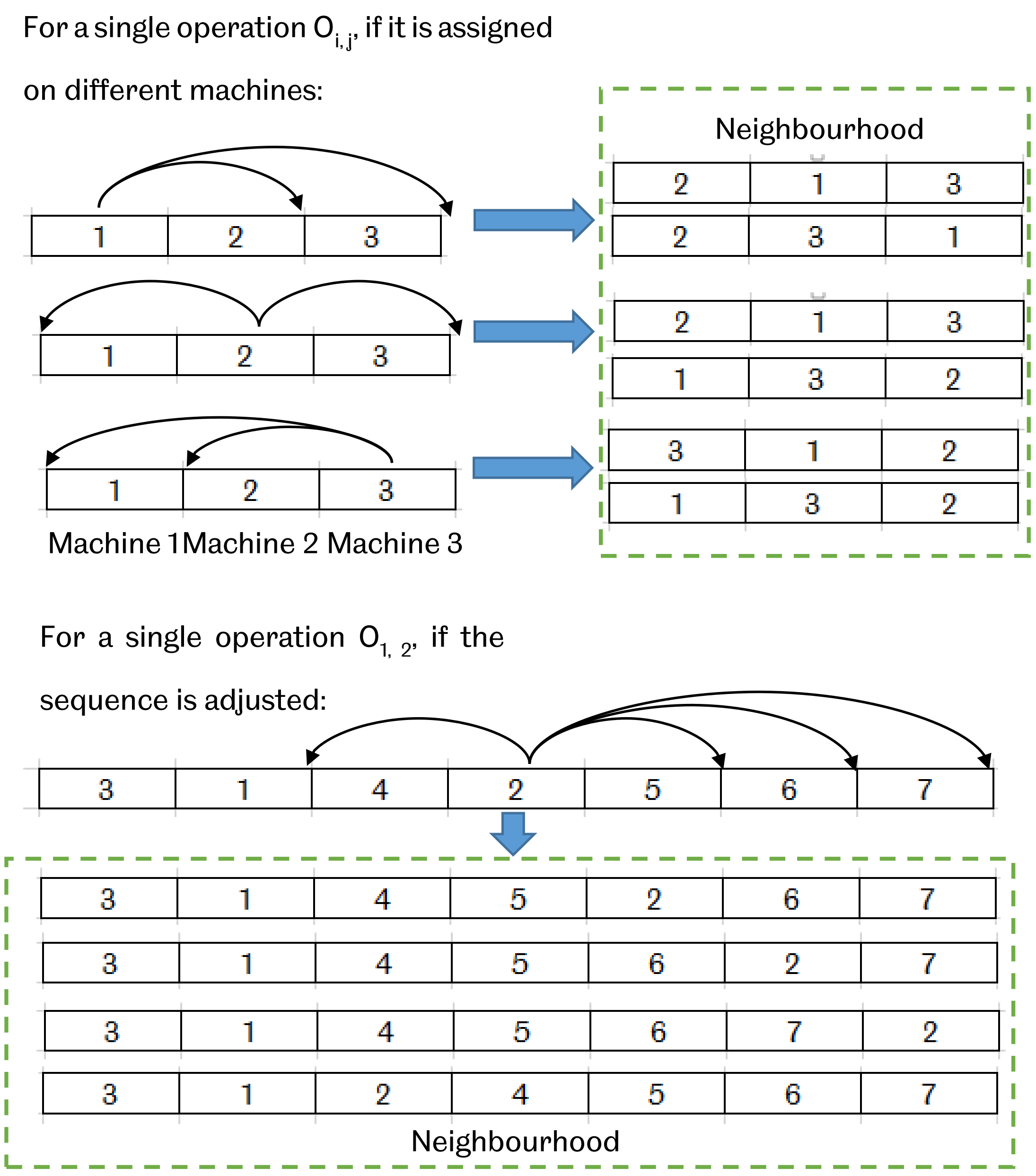

For a single operation $O_{i,j}$, if it is assigned on different machines:
Neighbourhood
Machine 1
Machine 2
Machine 3
For a single operation $O_{1,2}$, if the sequence is adjusted:
Neighbourhood

Figure A6 Illustration of neighborhood structures

For a possible schedule A shown as following, the critical path is $O_{3,1} \rightarrow O_{2,2} \rightarrow O_{2,3}$

Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	0	2
3 ($O_{2,1}$)	3	0	3
6 ($O_{3,1}$)	2	0	5
7 ($O_{3,2}$)	3	5	7
4 ($O_{2,2}$)	2	5	7
2 ($O_{1,2}$)	2	7	8
5 ($O_{2,3}$)	1	7	10

In schedule A: Max (0, 0) = 0

$P_{i,j,k} = 4$ Min (5, 8) = 5

$0 + 4 < 5$

Equation 4.3 is satisfied when $O_{3,1}$ is deleted and assigned to machine 1 (4 unit time) at a new position. The improved schedule B is shown as following.

Operation ID	Machine	Start time	End time
6 ($O_{3,1}$)	1	0	4
1 ($O_{1,1}$)	1	4	6
3 ($O_{2,1}$)	3	0	3
7 ($O_{3,2}$)	3	4	6
4 ($O_{2,2}$)	2	3	5
2 ($O_{1,2}$)	2	6	7
5 ($O_{2,3}$)	1	6	9

However, when performing local search on schedule B, equation 4.3 is satisfied again when $O_{3,1}$ is assigned to machine 2 at its original position in schedule A, which means the 'improved' schedule becomes schedule A with a larger makespan. Then the local search is trapped within this cyclic loop.

ES & EC for schedule A

Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	0	2
3 ($O_{2,1}$)	3	0	3
7 ($O_{3,2}$)	3	3	5
4 ($O_{2,2}$)	2	3	5
2 ($O_{1,2}$)	2	5	6
5 ($O_{2,3}$)	1	5	8

LS & LC for schedule A

Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	5	7
3 ($O_{2,1}$)	3	2	5
7 ($O_{3,2}$)	3	8	10
4 ($O_{2,2}$)	2	5	7
2 ($O_{1,2}$)	2	9	10
5 ($O_{2,3}$)	1	7	10

ES & EC for schedule B

Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	0	2
3 ($O_{2,1}$)	3	0	3
7 ($O_{3,2}$)	3	3	5
4 ($O_{2,2}$)	2	3	5
2 ($O_{1,2}$)	2	5	6
5 ($O_{2,3}$)	1	5	8

LS & LC for schedule B

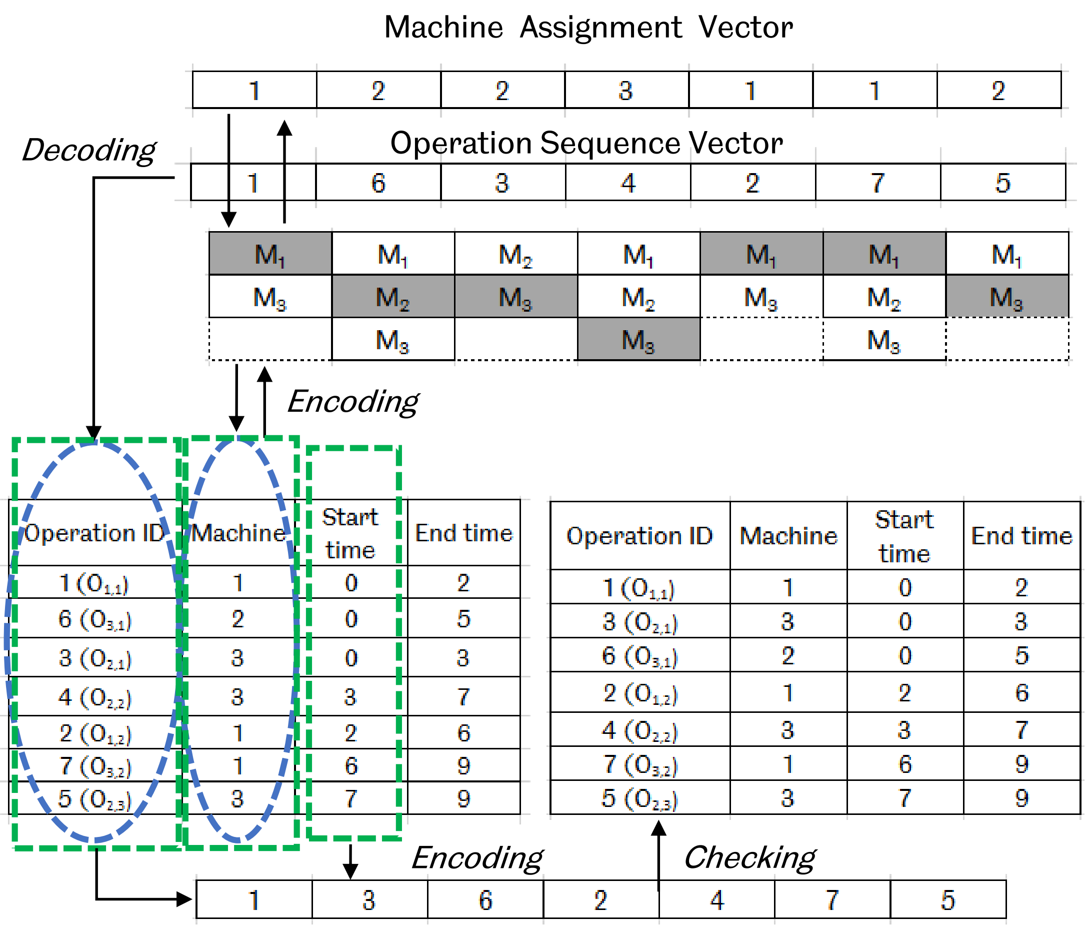
Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	4	6
3 ($O_{2,1}$)	3	1	4
7 ($O_{3,2}$)	3	7	9
4 ($O_{2,2}$)	2	4	6
2 ($O_{1,2}$)	2	8	9
5 ($O_{2,3}$)	1	6	9

In schedule B: Max (0, 0) = 0

$P_{i,j,k} = 5$ Min (7, 7) = 7

$0 + 5 < 7$

Figure A7 Illustration of a possible circumstance of cyclic search



Machine Assignment Vector

1	2	2	3	1	1	2

Operation Sequence Vector

1	6	3	4	2	7	5

M_1	M_1	M_2	M_1	M_1	M_1	M_1
M_3	M_2	M_3	M_2	M_3	M_2	M_3
	M_3		M_3		M_3	

Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	0	2
6 ($O_{3,1}$)	2	0	5
3 ($O_{2,1}$)	3	0	3
4 ($O_{2,2}$)	3	3	7
2 ($O_{1,2}$)	1	2	6
7 ($O_{3,2}$)	1	6	9
5 ($O_{2,3}$)	3	7	9

Operation ID	Machine	Start time	End time
1 ($O_{1,1}$)	1	0	2
3 ($O_{2,1}$)	3	0	3
6 ($O_{3,1}$)	2	0	5
2 ($O_{1,2}$)	1	2	6
4 ($O_{2,2}$)	3	3	7
7 ($O_{3,2}$)	1	6	9
5 ($O_{2,3}$)	3	7	9

1	3	6	2	4	7	5

*Blue frames refer to decoding process; Green frames refer to encoding process.

Figure A8 Illustration of Encoding and Decoding process

Table A3 Definition of several variables with abbreviation in local search algorithm

Name	Definition
$ES(O_{i,j})$	Earliest start time of $O_{i,j}$
$EC(O_{i,j})$	Earliest completion time of $O_{i,j}$
$LS(O_{i,j})$	Latest start time of $O_{i,j}$
$LC(O_{i,j})$	Latest completion time of $O_{i,j}$
$p_{i,j,k}$	Processing time of $O_{i,j}$ on M_k
$C_{max}(G)$	Makespan of schedule G
G	Original schedule
G’	Schedule after local search
$PM(O_{i,j})$	Operation processed on the same machine just precedes $O_{i,j}$
$SM(O_{i,j})$	Operation processed on the same machine just succeeds $O_{i,j}$
$PJ(O_{i,j})$	Operation in the same job right before $O_{i,j}$
$SJ(O_{i,j})$	Operation in the same job right after $O_{i,j}$

```
1: Get the critical path S-$co_1$-$co_2$...-$co_w$-E in G
2: for x = 1 to w do
3:     Delete the operation $co_x$ from G to get $G^-$
4:     for k = 1 to m do
5:         Get the set of available positions to assign $co_x$ on the machine $M_k$
6:         for each position j do
7:             if j satisfies Equation 4.3 then
8:                 Insert cox before the position j to get the improved schedule G'
9:                 return G'
10:            end if
11:        end for
12:    end for
13: end for
14: return G' = a null schedule    % signal for the completion of search for whole critical path
```

Figure A9 Pseudocode for the speed-up method in local search

```
1: i = 0; get $iter_{max}$ & $P_l$; l = rand(0,1)
2. while G' is not a null schedule and i < itermax and l < $P_l$
3.     G = G'
4.     i= i + 1
5. end while
6. return G
```

Figure A10 Pseudocode for the local search based on the speed-up method

Table A4 Different solutions of I_1

First solution (C_{max} = 9)				Second solution (C_{max} = 9)				Third solution (C_{max} = 9)			
$O_{i,j}$	M_m	S	E	$O_{i,j}$	M_m	S	E	$O_{i,j}$	M_m	S	E
$O_{1,1}$	1	0	2	$O_{3,1}$	3	0	2	$O_{1,1}$	1	0	2
$O_{3,1}$	2	0	5	$O_{1,1}$	1	0	2	$O_{2,1}$	2	0	3
$O_{2,1}$	3	0	3	$O_{2,1}$	2	0	3	$O_{3,1}$	3	0	2
$O_{2,2}$	3	3	7	$O_{2,2}$	3	3	7	$O_{1,2}$	2	3	4
$O_{2,3}$	3	7	9	$O_{3,2}$	1	2	5	$O_{2,2}$	3	3	7
$O_{1,2}$	1	2	6	$O_{1,2}$	1	5	9	$O_{3,1}$	1	2	5
$O_{3,2}$	1	6	9	$O_{2,3}$	3	7	9	$O_{2,3}$	3	7	9

Table A5 Different solutions of I_2

First solution (C_{max} = 11)				Second solution (C_{max} = 11)				Third solution (C_{max} = 11)			
$O_{i,j}$	M_m	S	E	$O_{i,j}$	M_m	S	E	$O_{i,j}$	M_m	S	E
$O_{2,1}$	1	0	2	$O_{2,1}$	1	0	2	$O_{1,1}$	4	0	1
$O_{4,1}$	1	2	3	$O_{1,1}$	4	0	1	$O_{3,1}$	3	0	6
$O_{3,1}$	4	0	7	$O_{4,1}$	2	0	5	$O_{2,1}$	1	0	2
$O_{1,1}$	5	0	2	$O_{2,2}$	5	2	7	$O_{1,2}$	2	1	5
$O_{1,2}$	2	2	6	$O_{3,1}$	3	0	6	$O_{3,2}$	2	6	7
$O_{2,2}$	5	2	7	$O_{1,2}$	1	2	7	$O_{2,2}$	5	2	7
$O_{1,3}$	1	6	10	$O_{3,2}$	2	6	7	$O_{1,3}$	3	6	11
$O_{3,2}$	2	7	8	$O_{4,2}$	4	5	6	$O_{4,1}$	4	1	5
$O_{4,2}$	2	8	9	$O_{1,3}$	1	7	11	$O_{3,3}$	4	7	9
$O_{3,3}$	4	8	10	$O_{2,3}$	3	7	11	$O_{4,2}$	4	9	10
$O_{2,3}$	3	7	11	$O_{3,3}$	4	7	9	$O_{2,3}$	1	7	11
$O_{3,4}$	4	10	11	$O_{3,4}$	4	9	10	$O_{3,4}$	4	10	11

Table A6 Different solutions of I_3

$O_{i,j}$	First solution (C_{max} = 11)				Second solution (C_{max} = 12)			
	Operation ID	M_m	S	E	Operation ID	M_m	S	E
$O_{1,1}$	4	7	0	3	27	6	0	6
$O_{1,2}$	18	7	3	5	15	3	0	4
$O_{1,3}$	1	1	0	1	12	2	0	1
$O_{2,1}$	21	4	0	1	28	6	6	7
$O_{2,2}$	15	3	0	4	1	1	0	1
$O_{3,1}$	16	3	4	5	13	1	1	3
$O_{3,2}$	12	2	0	1	18	7	0	2
$O_{3,3}$	6	7	5	6	21	4	0	1
$O_{4,1}$	24	5	0	4	6	2	1	5
$O_{4,2}$	9	6	0	4	9	1	3	5
$O_{4,3}$	2	6	4	5	2	5	1	2
$O_{5,1}$	17	3	5	7	16	3	4	5
$O_{5,2}$	13	1	1	3	29	1	7	8
$O_{5,3}$	27	2	1	5	4	7	2	5
$O_{6,1}$	19	7	6	9	22	4	1	9
$O_{6,2}$	28	6	5	6	24	5	2	6
$O_{6,3}$	22	4	1	9	5	1	8	10
$O_{7,1}$	10	2	5	8	3	6	7	11
$O_{7,2}$	14	1	3	8	23	2	9	11
$O_{7,3}$	23	2	9	11	17	7	5	12
$O_{8,1}$	5	1	8	10	7	3	5	6

$O_{8,2}$	29	1	10	11	25	5	6	10
$O_{8,3}$	7	6	6	7	10	4	9	10
$O_{9,1}$	20	3	9	11	26	4	10	12
$O_{9,2}$	25	5	4	8	14	2	11	12
$O_{9,3}$	11	5	8	9	11	1	10	11
$O_{10,1}$	8	5	9	11	19	3	6	10
$O_{10,2}$	3	6	7	11	20	3	10	12
$O_{10,3}$	26	7	9	11	8	5	10	12

Table A7 Influence of parameters F and Cr on HDE for Kacem Instance 1

	Cr = 0.1			*Cr* = 0.3			*Cr* =0.5			*Cr* = 0.7		
F	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av	AVG	SD	CPU av
0.1	11.33	0.58	0.10	11	0	0.11	11.05	0.02	0.11	11.3	0	0.10
0.3	11.35	0.54	0.11	11.14	0.31	0.12	11.01	0.02	0.11	11.07	0.03	0.12
0.5	11.42	0.55	0.11	11.18	0.38	0.12	11.12	0.22	0.16	11.02	0.06	0.13
0.7	11.37	0.54	0.12	11.12	0.29	0.13	11.14	0.25	0.14	11.15	0.23	0.14
*Best makespan for this instance is 11.												