Fuzzy scheduling of job orders in a two-stage flowshop with batch-processing machines

Alebachew D. Yimer, Kudret Demirli *

Fuzzy Systems Research Laboratory, Department of Mechanical and Industrial Engineering, Concordia University, 1515 St-Catherine W., Montreal, QC, Canada H3G 1M8

Received 10 April 2007; received in revised form 1 August 2007; accepted 25 August 2007
Available online 20 September 2007

Abstract

In this paper, we present a mixed-integer fuzzy programming model and a genetic algorithm (GA) based solution approach to a scheduling problem of customer orders in a mass customizing furniture industry. Independent job orders are grouped into multiple classes based on similarity in style so that the required number of setups is minimized. The family of jobs can be partitioned into batches, where each batch consists of a set of consecutively processed jobs from the same class. If a batch is assigned to one of available parallel machines, a setup is required at the beginning of the first job in that batch. A schedule defines the way how the batches are created from the independent jobs and specifies the processing order of the batches and that of the jobs within the batches. A machine can only process one job at a time, and cannot perform any processing while undergoing a setup. The proposed formulation minimizes the total weighted flowtime while fulfilling due date requirements. The imprecision associated with estimation of setup and processing times are represented by fuzzy sets.

© 2007 Elsevier Inc. All rights reserved.

Keywords: Fuzzy scheduling; Batch production; Built-to-order

1. Introduction

The build-to-order (BTO) manufacturing system is a demand satisfying strategy in supply chains that are involved in assembling of customized products. It combines the characteristics of both make-to-stock (forecast driven) and make-to-order (demand driven) strategies. In a BTO system, standard component parts and non-customizable subassemblies are acquired or build in-house based on short-term forecasts, while schedules for the few customizable parts and the final assembly are executed after detailed product specifications have been derived from booked customer orders. A BTO strategy is used to achieve economies-of-scale and improve customer service by allowing mass customization of products.

* Corresponding author. Tel.: +1 514 848 2424x3160; fax: +1 514 848 3175.
E-mail addresses: a_dessal@encs.concordia.ca (A.D. Yimer), demirli@encs.concordia.ca (K. Demirli).

0888-613X/S - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ijar.2007.08.013
Many furniture-manufacturing firms including Elran Furniture are undergoing a paradigm shift towards a built-to-order (BTO) and lean production system. In a BTO environment, firms assess specific needs of individual customers and manufacture products as per their requirements. Production of the final product is performed only after actual orders are received from customers. Adopting BTO processes allows firms to effectively customize their products in order to meet the specific requirements, resulting in enhanced satisfaction and better relationship with targeted customers. BTO processes also generate tremendous manufacturing cost savings in terms of reduced raw material inventories, reduced finished goods inventories, reduced space requirements, and increased flexibility. For example, Pella Inc., a manufacturer of windows and doors for both business and individual customers, has developed a build-to-order system [24,31].

The production setting of upholstered furniture manufacturing gives a remarkable opportunity to study the significance of batch scheduling in a build-to-order environment. The upholstering process that involves fabric cutting, sewing, stuffing and assembling operations is the critical process that impacts the efficiency of the whole production scheduling. Furniture manufacturers can implement mass customization of their products by taking different combinations of variety of styles with wide range of fabric types [32]. In order to limit the variety of options available to customers, furniture designers usually specify the fabrics that appear best with each style and publish the recommended mixes in product catalogs. A limited variety gives greater responsiveness to the market both in production time and procurement time. Agility and leanness in the shop floor are two important aspects that help manufacturers able to respond relatively quickly to specific customer orders. Flexibility is required in order to accommodate the dynamic workload imbalances inherent in producing different furniture styles. The changeover times between different styles must be very short to minimize WIP at each stage of production. The jigs and fixtures used must also be relatively simple to operate and easily exchangeable. Upholstered furniture products are bulk in size and thus consume large amounts of space; therefore, manufacturers must implement a lean production system in order to keep the products moving smoothly through the plant and to the customer [31].

In many research papers, setup time is considered to be part of the processing time. Though this assumption simplifies the analysis, it adversely affects the solution quality. Many applications such as group technology manufacturing system require an explicit treatment of setup [2,32]. Recently, an important class of scheduling with setup requirements is characterized by a flowshop group-scheduling problem. The jobs are classified into families based on operation similarities, and a single setup is required on a machine if it switches processing of jobs from one family to another, but no setup is required if the jobs are from the same family [11,33,26,30,18]. The problem of scheduling jobs with family setup times on parallel machines is also addressed in [25,14,15,29].

A mixed-integer programming approach for scheduling of batch-processing machines is proposed by [8,23]. Flowshop scheduling problems are NP-hard combinatorial problems, which usually involve extremely large solution space with too many local optima. When a sequence is altered slightly, it is difficult to evaluate the improvement of the objective function with respect to the global optimal solution. The highly unstructured nature of the search space makes the problem much more difficult to solve. A number of constructive heuristics exist that can provide good solutions to the problem in a relatively short processing of time. However, many of these algorithms operate by over simplifying the problem, which may not be acceptable from the practical aspect [5,19].

Introduced by Holland in the 1970s, genetic algorithm (GA) has proved to be a successful method for solving many practical optimization problems where the underlying search space is unstructured. It is a random search method which works based on the principle of survival of the fittest or natural selection. GA can provide better solutions when other methods like the branch and bound technique fail to perform efficiently [10]. It has been implemented to wide range of flowshop optimization problems [5,10,16]. Wang and Uzsoy [27] discussed the problem of minimizing maximum lateness by employing GA on a batch-processing machine in the presence of dynamic job arrivals. Ruiz and Maroto [20] have employed GA for a hybrid flowshop problem with sequence dependent setup times and machine eligibility. Sarker and Newton [22], applied GA for solving economic lot size scheduling problem. Recently, Damodaran et al. [7] addressed a makespan minimization problem on a batch-processing machine with non-identical job sizes using GA.

The majority of the literature on scheduling and sequencing of jobs is concerned with deterministic processing and setup times. In practice, however, as there exists a variation among operators effectiveness, those parameters cannot be determined with certainty [1,3,9,3,15]. In the recent decade there have appeared some
papers dedicated to fuzzy scheduling approaches, which consider fuzzy setup and processing times [6,12,28]. A comprehensive review of the literature for flowshop scheduling problems, and comparative evaluation of heuristics and metaheuristics algorithms are presented in [2,17,20].

In this paper, we present a mixed-integer fuzzy programming (MIFP) model for batch scheduling of job orders on parallel machines in a two-stage flowshop. The rest of the paper is organized as follows. A brief description of the scheduling problem considered is presented in Section 2. A MIFP formulation of the problem under study is set out in Section 3. In Section 4, we put forward an interactive fuzzy satisfying solution procedure to the proposed model. Computational results show that the MIFP model can be solved, in reasonable CPU runtime, for only limited number of jobs. For problems with larger number of jobs, we describe a genetic algorithm based solution approach in Section 5. Finally, concluding remarks are given in Section 6.

2. Problem description

In the upholstered furniture manufacturing process at Elran, the frame building operation is performed as a single card kanban or constant work in process (CONWIP) system [32]. Whence this operation is basically independent of the individual job orders. However, scheduling of the orders directly affects the performance of the fabric processing (stage-1) and the upholstering (stage-2) workcenters. In manufacturing processes with a bottleneck operation, there is a desire to keep changeovers as few as possible in order to reduce the non-production setup time required. When setups are very costly in terms of money or time, jobs with similar characteristics are often grouped and processed together [29]. In such scenarios, grouping jobs together in a batch and allowing a single setup per batch may give sound operational advantage.

The required number of setups, both at the fabric cutting–sewing and the upstream upholstering operations are reduced if the jobs can be grouped together by virtue of similarity in style. The main focus of this paper is thus to partition the family groups into a sequence of batches so as to minimize the total weighted flowtime, while maintaining delivery promise dates. Flowtime measures the length of time a job stays within the system. Minimizing the total flowtime helps to reduce WIP inventory and improve customer service interims of responsiveness. Therefore, the scheduling problem under study requires five distinct, but interdependent decisions to be made:

- **Grouping decision** – classify the set of job orders into families based on their setup similarity,
- **Batching decision** – find out which jobs of the same family are to be included in each batch,
- **Allocating decision** – resolve how batches are assigned to available parallel machines at each stage of operation,
- **Sequencing decision** – determine the order in which the batches and the jobs within each batch are to be processed, and
- **Sorting decision** – regroup the finished jobs based on their due date and customer-ID.

Basically, the first and last decisions require technical and clerical work, while the other three require serious optimization technique. At Elran, furniture orders are first grouped into families on the basis of similarity in style. For each family, batching of the jobs and allocation on machines are done using subjective managerial judgments by considering the order size and sometimes the fabric type. With the exception of few esteemed customers, the batches are mainly processed in a first-come first-served (FCFS) order. However, as evidenced by different computational results, such sequencing is not the optimal schedule that minimizes the total weighted flowtime of the system. In the finished products store, furniture items also wait for a longer time than necessary until all orders from one customer, but could be in different batches, are being processed.

In this paper, a set of independent job orders that are received from a group of customers \( k = 1, \ldots, K \) at release time \( r_k \) and for due at time \( d_k \) are considered. The jobs are first indexed as \( j = 1, \ldots, J \) on the basis of earliest release time rule and next by shortest processing time at the critical stage \( p_{j,i} \). At each stage of operation \( i = 1, 2 \), the jobs are partitioned into distinct families \( g = 1, \ldots, G \) according to setup similarity. A sequence of batches \( b = 1, \ldots, B \) are to be created by assigning jobs from the same family \( j \in \Omega_{g} \). A batch cannot contain jobs from different groups. A sequence independent setup time \( a_{g,i} \) at each stage is required prior to processing a batch, but there are no setups between jobs within a batch. The release time of a batch
is the time its setup may begin such that once complete, all jobs in the batch are processed without idle time. The total number of batches created should be greater or equal to the number of family groups available and less than the total number of independent jobs available.

At each stage of operation, there are \( m \) identical parallel machines available, and each job is processed on one machine at a time. Machines cannot be preempted. The batch-processing time \( P_{bi} \) at each stage is equal to the completion time of the last job \( C_{bi} \), minus the starting time of the first job in that batch \( S_{bi} \). An optimal schedule can be regarded as a sequence of batches, where a batch is a maximal consecutive subsequence of the jobs from the same class. It is shown in the literature that this problem is highly NP-hard, even in case of a single machine [33] and one stage parallel machines [11,25] with no setup consideration.

3. A MIFLP formulation of the problem

In formulating scheduling models, parameters such as job processing, ready and setup times are conventionally treated as deterministic values. However, in real-world situations, these parameters are often associated with uncertainties. The length of time required to process parts on machines cannot be determined precisely because of measurement errors and in involvement of human actions in the manufacturing process. Due to the inconsistency in the performance of operators and machines at the shop floor, repeated measurement of the system’s parameters provides a certain range of values. Therefore, the information that we have about the model parameters is often vague and imprecise [9,15]. For instance, Elran management allocates an incentive mechanism to motivate employers working on parallel processing lines so that they will achieve preset target levels per shift. This will create computation among group of workers and narrow the gap in their performance. The prevalent approach used to represent uncertain parameters is using probabilistic distribution functions. However, this approach is computational expensive, since the probability distributions are basically defined from historical data by applying statistical techniques [3]. In a situation where we lack sufficient information to sharply define the parameters, qualitative terms described by linguistic expressions like ‘too short’ or ‘about 100’ are often used based on imprecise data. Indeed, fuzzy set theory provides the means for handling uncertain model parameters, which are not given as crisp values but rather as interval values representing estimates [1].

In this section, we formulate a mixed-integer fuzzy linear programming (MIFLP) model for the problem described in Section 2. The objective is to determine the set of jobs to be included in each family and sequence the batches so that the total weighted flowtime will be minimized. The uncertain time related parameters (setup or processing) are represented by triangular fuzzy sets.

3.1. Nomenclature

The following notations are used in formulating of the model:

Indices and sets

- \( j \) index of jobs, \( j = 1, \ldots, J \)
- \( k \) index of customers, \( k = 1, \ldots, K \)
- \( i \) index of processing stages, \( i = 1, 2 \)
- \( g \) index of job families or groups, \( g = 1, \ldots, G \)
- \( b \) index of processing batches, \( b = 1, \ldots, B \)
- \( \Omega_k \) set of job indexes from customer \( k \)
- \( \Omega_g \) set of job indexes in group \( g \)
- \( \Omega_b \) set of job indexes in batch \( b \)

Crisp parameters

- \( w_k \) priority rating of customer \( k \) (0 < \( w_k \) < 1)
- \( r_k \) release time for orders made by customer \( k \)
- \( d_k \) due date for orders made by customer \( k \)
- \( n_k \) number of independent jobs ordered by customer \( k \)
mi \quad \text{number of available machines at stage } i

nb \quad \text{number of jobs assigned to batch } b

M^\infty \quad \text{very big positive number}

Fuzzy parameters

\tilde{a}_{g,i} \quad \text{machine setup time at stage } i \text{ for jobs in group } g

\tilde{q}_{g,i} \quad \text{processing time per job in group } g \text{ at stage } i

\tilde{p}_{j,i} \quad \text{processing time of job } j \text{ at stage } i

\tilde{C}_{j,i} \quad \text{completion time of job } j \text{ at stage } i

\tilde{S}_{b,i} \quad \text{processing start time of first job in batch } b \text{ at stage } i

\tilde{P}_{b,i} \quad \text{processing times of all jobs in batch } b \text{ at stage } i

\tilde{C}_{b,i} \quad \text{completion time of last job in batch } b \text{ at stage } i

\tilde{C}_k \quad \text{stage-2 completion time of last job in customer group } k

\tilde{C}_\text{max} \quad \text{imprecise makespan}

\tilde{z}(\tilde{x}) \quad \text{imprecise total weighted flowtime}

A tilde mark on top of the symbols is used to show that those variables represent imprecise values or fuzzy numbers:

Binary integers

W_{k,j} 1 \text{ if job } j \text{ is ordered by customer } k (j \in \Omega_k), \text{ or } 0 \text{ otherwise}

X_{b,j} 1 \text{ if job } j \text{ is assigned to batch } b (j \in \Omega_b), \text{ or } 0 \text{ otherwise}

Y_{g,b} 1 \text{ if all jobs into batch } b \text{ are from group } g (\Omega_b \subseteq \Omega_g), \text{ or } 0 \text{ otherwise}

Z_{g,j} 1 \text{ if job } j \text{ is member of group } g, (j \in \Omega_g), \text{ or } 0 \text{ otherwise}

General variables

\chi_f(\tilde{x}) \quad \text{fuzzy solution space}

\chi_c(\tilde{x}) \quad \text{crisp solution space}

\tilde{x} \quad \text{a feasible solution vector of decision variables } \tilde{x} \in \chi_f(\tilde{x}) \cup \chi_c(\tilde{x})

\lambda \quad \text{fuzzy goal satisfying level } (0 < \lambda < 1)

Note that values for the following parameters: \(m_i, w_k, r_k, d_k, \Omega_k, \Omega_g, \tilde{a}_{g,i}, \tilde{q}_{g,i}\) are predetermined and will be used as input to the model.

3.2. The proposed model

Fuzzy goal function: Flowtime of a job is the length of time the job stays within the system starting from order release to final delivery. Since jobs from a given customer are released and delivered together, they will have the same flowtime. Therefore, the total weighted flowtime of all jobs is the sum of the flowtime of the individual jobs in each customer group multiplied by the priority rating of the customers. The fuzzy objective function (1) gives the imprecise weighted total flowtime of all jobs:

\[ \tilde{z}(\tilde{x}) = \sum_{k=1}^{K} \sum_{j=1}^{J} w_k(\tilde{C}_k - r_k) \cdot W_{k,j} = \sum_{k=1}^{K} w_k(\tilde{C}_k - r_k)n_k \] (1)

Crisp solution space: The constraints related with batching restrictions do not depend on the fuzzy time variables. Therefore, they are considered to be crisp:
\(\chi_c(x) \equiv \sum_{b=1}^{B} X_{b,j} = 1 \quad \forall j\) \hspace{1cm} (2)

\(\sum_{g=1}^{G} Y_{g,b} \leq 1 \quad \forall b\) \hspace{1cm} (3)

\(X_{b,j} \leq \sum_{\Omega}^{G} Z_{g,j} \cdot Y_{g,b}, \quad j \in \Omega_g \quad \forall b\) \hspace{1cm} (4)

\(n_b = \sum_{j=1}^{J} X_{b,j} \quad \forall b\) \hspace{1cm} (5)

\(X_{b,j}, Y_{g,b}, Z_{g,j}, W_{k,j} \in \{0, 1\}\)

\(i = 1, 2, \quad j = 1, \ldots, J, \quad k = 1, \ldots, K\)

\(b = 1, \ldots, B, \quad g = 1, \ldots, G\) \hspace{1cm} (6)

Constraints (2) ensures that a job must be assigned to exactly one batch. Since there is no prior information as to how many batches can be created, we can initially assume that there will be at most \(J\) batches. Some of these batches may have multiple jobs and others could be with no job assigned. Therefore, constraint (3) restricts that all jobs assigned to a batch are derived from the same family (\(Y_{g,b} = 1\)), or else no job will be assigned (\(Y_{g,b} = 0\)). Constraint (4) controls that a job in a given group (\(Z_{g,j} = 1\)) can be assigned to a batch (\(X_{b,j} = 1\)) if and only if the group itself is assigned to the batch (\(Y_{g,b} = 1\)). Constraint (5) determines the number of jobs assigned in each batch \(b\). Constraint (6) restricts the decision variables \(W_{k,j}, X_{b,j}, Y_{g,b}\) and \(Z_{g,j}\) to be binary integers.

**Fuzzy solution space:** All other sequencing constraints, which are dependent on uncertain time parameters, are thus fuzzy constraints:

\(\chi_f(x) \equiv \tilde{p}_{i,j} = \sum_{g=1}^{G} \tilde{a}_{g,i} \cdot Z_{g,j} \quad \forall i,j \in \Omega_g\) \hspace{1cm} (7)

\(\tilde{S}_{b,i} = \tilde{C}_{(b-m,i)} \quad \forall i b > m_i\) \hspace{1cm} (8)

\(\tilde{S}_{b,j} \geq \tilde{c}_{j,1} + M^\infty (X_{b,j} - 1)b \leq m_2 \quad \forall j\) \hspace{1cm} (9)

\(\tilde{P}_{i,b} = \sum_{g=1}^{G} \tilde{a}_{g,i} \cdot Y_{g,b} + \sum_{j=1}^{J} \tilde{p}_{i,j} \cdot X_{b,j} \quad \forall i \quad \forall b\) \hspace{1cm} (10)

\(\tilde{C}_{b,i} = \tilde{S}_{b,i} + \tilde{P}_{i,b} \quad \forall i \quad \forall b\) \hspace{1cm} (11)

\(\tilde{c}_{i,j} \geq \tilde{C}_{b,i} + M^\infty (X_{b,j} - 1) \quad \forall i \quad \forall b \quad \forall j\) \hspace{1cm} (12)

\(\tilde{C}_{b,j} \geq \tilde{C}_{b,j} \cdot W_{k,j} \quad \forall j \in \Omega_k \quad \forall k\) \hspace{1cm} (13)

\(\tilde{C}_{b,k} \leq \tilde{d}_{k} \quad \forall k\) \hspace{1cm} (14)

\(\tilde{C}_{\max} \geq \tilde{C}_{b} \quad \forall k\) \hspace{1cm} (15)

\(\tilde{S}_{b,i}, \tilde{P}_{i,b}, \tilde{C}_{b,i}, \tilde{C}_{k}, \tilde{p}_{i,j}, \tilde{c}_{j} \geq 0\) \hspace{1cm} (16)

\(i = 1, 2, \quad j = 1, \ldots, J, \quad k = 1, \ldots, K\)

\(b = 1, \ldots, B, \quad g = 1, \ldots, G\)

Constraint (7) determines the processing time of each job at each stage of operation. Taking the number of identical parallel machines at each stage \(m_i\) into account, constraints (8) and (9) determine the operation starting time of the first job in the \(b\)th batch. Allowing the necessary sequence independent setup times for each batch, constraint (10) determines the batch-processing time period required at each stage. The processing time of a batch \(b\) is the sum of the processing times of all the jobs within the batch plus a machine setup time. Constraint (11) determines the completion time of the last job in a batch \(b\), while constraint (12) resolves the com-
pletion time of the individual jobs at each stage of operation. Constraint (13) determines the longest completion time for set of jobs coming from the same customer group, and constraint (14) restricts the group completion time to be within the promised due dates. Constraint (15) determines the maximum completion time of all jobs or the makespan. Constraint (16) imposes a nonnegativity restriction on the dependent time parameters.

3.3. Fuzzy goal programming

The uncertain time dependent parameters are represented by fuzzy sets. The degrees of membership functions for the fuzzy numbers parameters are defined based on subjective judgments. A symmetric triangular fuzzy number is considered to be a more simplistic shape function as it can be constructed easily from two basic estimates – most possible value and maximum deviation from it [32]. For example, a symmetric triangular membership function for a fuzzy setup time $\bar{a}_{g,i}$ can be defined by

$$\bar{a}_{g,i} = a_{g,i}^m \pm \alpha \bar{a}_{g,i} = (a_{g,i}^m - \alpha, a_{g,i}^m, a_{g,i}^m + \alpha)$$

(17)

The right and left extreme values have the lowest likelihood of belonging to the set of possible values, and hence have a null degree of membership $|\mu_a(a_{g_i}^m) = \mu_a(a_{g_i}^l) = 0$. The most likelihood value, which lays exactly at the midpoint between the two bound estimates, posses the highest degree of membership $|\mu_a(a_{g_i}^m) = 1|$. Other values within the span of $\bar{a}_{g,i}$ will assume a linearly varying membership degree in between 0 and 1. Other fuzzy parameters can also be represented in the same fashion. Fig. 1 depicts a symmetric triangular membership function for $\bar{a}_{g,i}$.

Likewise, the fuzzy goal function $\tilde{z}(\bar{x})$ can be defined in terms of two crisp functions for weighted flowtime:

$$\tilde{z} = z^m(\bar{x}) \pm z^0(\bar{x})$$

where

$$z^m(\bar{x}) = \sum_{k=1}^{K} \sum_{j=1}^{J} w_k (C^m_k - r_k) \cdot W_{k,j}, \quad \text{and} \quad z^0(\bar{x}) = \sum_{k=1}^{K} \sum_{j=1}^{J} w_k (C^0_k - r_k) \cdot W_{k,j}$$

(18)

Similarly, the fuzzy solution space $\chi_f(\bar{x})$ shown by Eqs. (7)–(16) can also be defined as a combination of two sets of crisp constraints as described next:

$$\chi_f(\bar{x}) \equiv \chi_m(\bar{x}) \pm \chi_0(\bar{x})$$

(19)

where

$$\chi_m(\bar{x}) = P_{ij}^m = \sum_{g=1}^{G} q_{g,ij}^m \cdot Z_{g,ij} \quad \forall ij \in \Omega_g$$

$$S_{b,i}^m = C_{b-m,i} \quad \forall i \ b > m$$

$$S_{b,k}^m \geq e_{m,j+1} + M \cdot (X_{b,j} - 1) \ b \leq m \quad \forall j$$

$$P_{b,i}^m = \sum_{g=1}^{G} a_{g,ij}^m \cdot Y_{g,ib} + \sum_{j=1}^{J} p_{ij}^m \cdot X_{b,j} \quad \forall i \ \forall b$$

$$C_{b,i}^m = C_{b,i}^m + P_{b,i}^m \quad \forall i \ \forall b$$

$$C_{b,j}^m \geq C_{b,j}^m + M \cdot (X_{b,j} - 1) \quad \forall i \ \forall b \ \forall j$$

$$C_{k}^m \geq C_{k}^m \cdot W_{k,j} \quad \forall j \ \forall k$$

$$C_{k}^m \leq d_k \quad \forall k$$

$$C_{k}^m \geq C_{k}^m \quad \forall k$$

$$S_{b,i}^m, P_{b,i}^m, C_{b,i}^m, C_{b,j}^m, C_{k}^m, C_{k}^m, C_{j}^m \geq 0$$

and
When information associated with the objective and the set of constraints is vague, the problem can be formulated as a fuzzy goal programming problem of type (22): A fuzzy decision is obtained by taking the intersection of the fuzzy objective and the total solution space \[ \mathcal{Z} \]

\[ \mathcal{Z}(\bar{x}) \cong p^i_{\bar{x}} = \sum_{g=1}^{G} q^i_{g,j} \cdot Z_{g,j} \quad \forall i \in \Omega_g \]

\[ S_{b,i} = C_{b,m,i} \quad \forall i \quad b > m_i \]

\[ S_{b,2} \geq c^i_{j,1} + M^\infty (X_{b,j} - 1) \quad b \leq m_2 \quad \forall j \]

\[ P_{b,1} = \sum_{j=1}^{b} p^i_j \cdot X_{b,j} \quad \forall i \quad \forall b \]

\[ C^i_{b,j} = S^i_{b,j} + P^i_{b,j} \quad \forall i \quad \forall b \]

\[ c^i_{j,1} \geq C^i_{b,j} + M^\infty (X_{b,j} - 1) \quad \forall i \quad \forall b \quad \forall j \]

\[ C_k \geq c^i_{j,2} \cdot W_{k,j} \quad j \in \Omega_k \quad \forall k \]

\[ C_k \leq d_k \quad \forall k \]

\[ C_{\max,k} \geq C_k \quad \forall k \]

\[ S_{b,1}, P_{b,1}, C_{b,1}, C_{k,1}, P_{b,1}, c^i_{j,1} \geq 0 \]

A fuzzy decision is obtained by taking the intersection of the fuzzy objective and the total solution space \[ \mathcal{Z} \]. When information associated with the objective and the set of constraints is vague, the problem can be formulated as a fuzzy goal programming problem of type (22):

\[ \text{Find : } \bar{x} \]

\[ \text{To satisfy : } \tilde{z}(\bar{x}) \cong z^m(\bar{x}) \quad \text{and} \quad \bar{x} \in \chi_c(\bar{x}) \cup \chi_f(\bar{x}) \]

where \( \bar{x} \) is a solution vector of decision variables within a feasible solution space \( \chi_c(\bar{x}) \cup \chi_f(\bar{x}) \), and \( z^m(\bar{x}) \) refers to the target value of the fuzzy goal. The symbol \( \tilde{z} \) in the goal constraint represents the linguistic term ‘about’ and it means that the resulting total weighted flowtime \( \tilde{z}(\bar{x}) \) should be around the vicinity of the aspiration value \( z^m(\bar{x}) \), with some symmetric deviation \( z^i(\bar{x}) \) on both sides. As outlined in Section 4, the target value \( z^m(\bar{x}) \) is evaluated by taking the most possible values for the individual time dependent fuzzy parameters.

### 4. Solution approach

For the fuzzy integer programming problem (FILP) presented in Section 3.2, the imprecise objective function will have a triangular symmetric possibility distribution. Its shape function can be defined in terms of the three vertices: \( \tilde{z}(\bar{x}) = (\tilde{z}_c(\bar{x}), \tilde{z}_c(\bar{x}), \tilde{z}_c(\bar{x})) \). Minimization of \( \tilde{z}(\bar{x}) \) is achieved by pushing those three vertices towards the origin. To this end, the mixed-integer fuzzy programming problem is transformed into an auxiliary multi objective linear programming (MOLP) problem by converting \( \tilde{z}(\bar{x}) \) into three interdependent crisp objectives [32]. The simultaneous optimization of the three objectives involves minimizing the most possible value \( z_1(\bar{x}) \), maximizing the possibility of obtaining lower objective \( z_2(\bar{x}) \), and minimizing the risk of obtaining higher objective value \( z_3(\bar{x}) \), as shown in (23):
Min \quad z_1(\vec{x}) = z^m(\vec{x}) \\
Max \quad z_2(\vec{x}) = z^{m-1}(\vec{x}) = z^d(\vec{x}) \\
Min \quad z_3(\vec{x}) = z^{m-2}(\vec{x}) = z^\mu(\vec{x}) \\
Subject to : \quad \vec{x} \in \chi_c(\vec{x}) \cup \chi_d(\vec{x})

(23)

where \(z^d(\vec{x})\) is the symmetric deviation of the triangular fuzzy number \(\vec{z}\) with respect to the aspiration value \(z^m\).

Employing the fuzzy decision making of Bellman and Zadeh [4] and Zimmermann [34] fuzzy programming method, the auxiliary MOLP problem can be converted into an equivalent single goal linear programming problem. By solving (23) for each objective \(z_i(\vec{x})\) separately, we can determine the initial values for the positive and the negative ideal solutions. Therefore

\[
\begin{align*}
\mu^\text{PIS}_1 &= \min z^m(\vec{x}) \\
\mu^\text{NIS}_1 &= \max z^m(\vec{x}) \\
\mu^\text{PIS}_2 &= \mu^\text{NIS}_3 = \max z^\mu(\vec{x}) \\
\mu^\text{NIS}_2 &= \mu^\text{PIS}_3 = \min z^\mu(\vec{x})
\end{align*}
\]

(24)

The decision maker can later adjust those parameters interactively within the range of values obtained from (24). The three objective functions are then translated into fuzzy goals using the linear membership functions shown in Fig. 2. Equivalently, the three membership functions can be expressed algebraically as in (25):

\[
\begin{align*}
\mu_1(z_1) &= \frac{z^\text{NIS}_1 - z_1}{z^\text{PIS}_1 - z^\text{NIS}_1} \\
\mu_2(z_2) &= \frac{z^\text{NIS}_2 - z_2}{z^\text{PIS}_2 - z^\text{NIS}_2} \\
\mu_3(z_3) &= \frac{z^\text{NIS}_3 - z_3}{z^\text{PIS}_3 - z^\text{NIS}_3}
\end{align*}
\]

(25)

Using such linear membership functions and following the fuzzy decision of Bellman and Zadeh [4], the original MOLP problem can be interpreted as

Maximize : \( \min \{ \mu_1(z_1), \mu_2(z_2), \mu_3(z_3) \} \) \\
Subject to : \( \vec{x} \in \chi_c(\vec{x}) \cup \chi_d(\vec{x}) \) \\

(26)

By introducing an auxiliary fuzzy goals satisfying level \(\lambda\) (0 \(\leq\) \(\lambda\) \(\leq\) 1), the MOLP problem can be reduced to Zimmermann’s [34] equivalent single objective conventional LP problem:

Maximize : \( \lambda \) \\
Subject to : \( \lambda \leq \mu_i(z_i) \) for \(i = 1, 2, 3\) \\
\( \vec{x} \in \chi_c(\vec{x}) \cup \chi_d(\vec{x}) \) \\

(27)

Higher value of \(\lambda\) (close to 1) indicates that the three objective functions are optimized to a high degree of satisfaction level.

**Numerical example-1:** A small sized problem consisting of 14 independent job orders from three customers is considered to demonstrate the approach. Assume, the set of independent jobs can be grouped into four
families based on their style. Two parallel machines are available at each stage of operation. The set of job indices in each group, and the corresponding fuzzy setup and unit processing times are tabulated in Table 1.

Employing Eq. (25), the positive and negative ideal solutions for the three objective functions are calculated as follows:

\[ z_{1}^{\text{PIS}} = \min z_m(\tilde{x}) = 18840 \]
\[ z_{1}^{\text{NIS}} = \max z_m(\tilde{x}) = 25000 \]
\[ z_{2}^{\text{PIS}} = z_{2}^{\text{NIS}} = \max z_d(\tilde{x}) = 4500 \]
\[ z_{2}^{\text{NIS}} = z_{2}^{\text{PIS}} = \min z^*(\tilde{x}) = 3248 \]

For the equivalent single objective LP model shown in (27), the final results are given in Table 2. The auxiliary model is solved using a commercial software LINGO 8.0 installed on a pentium-4 PC, and takes a CPU run-time of 1.35 h. The three auxiliary objective functions are optimized simultaneously with a degree of satisfaction level \( k/C_3 = 0.675 \), bearing the values: \( z_1^{\text{PIS}} = 22087 \) and \( z_3^{\text{NIS}} = 3473 \). As illustrated in Fig. 3, the triangular expectation fuzzy number for the imprecise objective \( \tilde{z} \) is quantified from the three values.

\[ \mu(z(x)) \]

![Fig. 3. Fuzzy output of imprecise total flowtime, \( \tilde{z}(\tilde{x}) \).](image)
(17714, 22087, 26460). Since the shape function for the fuzzy output $\tilde{z}$ is continuous and symmetric, the most likely value $z^m = 22087$ can be considered as the defuzzified value of the imprecise total flowtime.

5. GA based solution procedure

GA is a population based heuristic search algorithm that mimics the process of evolution and heredity in nature. It follows the principles of ‘survival of the fittest’ in natural selection to search for best fit individuals within the solution space. In the algorithm, the solution of the problem is coded as a string structure called chromosome. In order to arrive at a near-optimal solution, GA begins searching from a set of randomly generated chromosomes called initial population and evolves to better sets of solutions over a sequence of iterations. Each chromosome in the population is evaluated and assigned a fitness value. Fitness value is a measuring criterion for the aptness of the objective function value. Whence, they are directly proportional to each other for the case of a maximization problem, and inversely proportional for the case of a minimization problem. The higher the fitness value, the better the individual chromosome would be. The search for good chromosomes is guided by the value of the objective function (or fitness measure) for each chromosome in the population [7,16,20].

In GA, one complete-iteration is often referred as a generation. In a given generation, individual chromosomes within the population will undergo a series of genetic operations and strive for survival to the next generation. Consequently, new chromosomes called offspring with better fitness evolve from the genetic processes. A parent selection mechanism is applied to choose individuals from the current population to a mating poll. Selected individuals in the mating poll reproduce by exchanging genetic materials in a process called crossover operation. Some other chromosomes are also selected to undergo a genetic process called mutation in which only certain parts of their genes are altered to bear a new chromosome. Mutation operators maintain population diversity by slight perturbations of selected solutions [10,21].

Finally, set of individuals which will pass to the next generation are chosen by employing a survivor selection mechanism. The fittest chromosomes should have a greater chance of being selected in the process. Once an initial population has been created, parent selection, genetic operations, and survivor selection are performed sequentially in each generation until it converges to the optimal solution or a stopping criterion is met. This procedure is demonstrated by the pseudo-code shown in Fig. 4. For further understanding of the algorithm, we recommend to Ref. [13]. The solution of decision variables in the original problem is then obtained by decoding the best individual chromosome of the final generation. The effectiveness of the GA greatly affected by the proper choice of the chromosomal encoding scheme, parent and survivor selection mechanisms and by the parametric values of crossover and mutation operators [10,22].

5.1. Solution representation (encoding scheme)

In a family setup batch scheduling problem, jobs are first sorted into families manually according to their setup similarity. Next, successive decisions of batching and sequencing are made in two phases. Whence, two

```
BEGIN
  t←0 ;
  Initial population (t = 0) ;
  Fitness evaluation (t = 0); 
  DO
    { Parent selection (t);
      Genetic operations (t);
      Offspring population (t);
      Fitness evaluation (t);
      Survivors selection (t) ;
      t←t+1;
    } WHILE (termination not satisfied);
END
```

Fig. 4. Steps in implementing GA.
groups of decision variables represent a solution structure – variables for batching decisions, and variables for a sequencing decision. The batching decision variables are used to identify the jobs that are included in each batch from the same family group, while the sequencing decision variables determine the order by which the batches are processed. Therefore, the chromosomal encoding scheme consists of structured information for these two parts: an outer string for the ordering of the batches and an inner string for the creation of each batch.

A path representation using an array of batch indices is the most efficient and widely implemented encoding scheme for the sequencing decision [5,10,13,16]. As shown in Fig. 5, the vector of batch indices \( \{b_1, b_2, \ldots, b_B \} \) in the sequencing-string imply that the whole set of jobs available are grouped into \( B \) batches, and each batch is ordered according to its index value. \( F \) corresponds to the objective function (i.e., the total weighted flowtime) value for a given solution. It is computed by taking the weighted sum of the elapsed times between receiving and fulfilling of all orders accommodated within the planning horizon. Embedded within each batch, we will find a detail information on how the batch is created. Associated with a given batch \( b_k \), a batching string tells the number of jobs \( (n_b \leq n_g) \) and set of their indices included \( (\Omega_b \subseteq \Omega_g) \), index key of their family group \( (g_b) \), and the batch-processing times on both machines \( (P_{1,b} \text{ and } P_{2,b}) \). Through the family index key \( (g_b) \), the batching string retrieve specific data pertaining to that family from the input database. The family string data consists of group setup time on the two machines \( (a_{1,g} \text{ and } a_{2,g}) \), unit processing time of a single job on both machines \( (q_{1,g} \text{ and } q_{2,g}) \), total number of jobs \( (n_g) \) and corresponding set of job indices \( (\Omega_g) \) within that family. The batch-processing time on both machines can be calculated by the following equation:

\[
P_{i,b} = a_{i,b} + \sum_{j \in \Omega_b} w_j \cdot q_{i,b} \quad \text{for } i = 1, 2
\]

where \( w_j \) refers to the weighting factor of job \( j \).

5.2. Fitness evaluation

Before a fitness value is assigned, the total weighted flowtime corresponding to each chromosome in the current population must be evaluated. For a given solution representation, the batches will be processed at stage-I according to the order of the batching indices in its sequencing-string. As shown in Fig. 6, batch \( b_1 \) will be assigned to available first machine, \( b_2 \) to second machine and so on. Once all parallel machines are fully loaded, the rest of the batches, which are not yet assigned, will keep waiting in Line-I until one or more of the machines are freed once again. The batches which are processed on one of the machines at stage-I, will join a second waiting line according to their earliest finish time. At stage-II, the batches are processed according to their lineup sequence in waiting Line-II. Since the setup and processing time of each batch is variable, the resulting sequence in Line-II could probably be different from that of Line-I. This will avoid unnecessary restriction by a permutation sequencing rule which might be imposed only for the sake of simplifying the problem. After the batches complete their processing at stage-II, the jobs within each batch will be re-sorted into their customer groups: \( k_1, \ldots, k_m \). After obtaining the completion time of all batches at stage-II, from the simulation process, the total weighted flowtime is computed by the following equation:

\[
F = \sum_{k=1}^{K} n_k (c_k - r_k) w_k \quad \text{where } c_k = \{ \max(c_j) : \forall j \in \Omega_k \}
\]
After the objective function (i.e., total flowtime) value is computed for each member of the population, an equivalent fitness value $f_i$, which reflects the relative importance of the individual solution in its domain, is assigned. Since our objective is to minimize the total flowtime, a member with the lowest objective value should correspond to the highest fitness value and vice versa. Therefore, for each chromosome $i$, its fitness value $f_i$ can be evaluated by taking a factor $K$ times the reciprocal of the objective function value $F_i$:

$$f_i = \frac{K}{F_i} \quad \text{for } i = 1, \ldots, N \text{ and } K = \text{constant}$$

(30)

5.3. Initial population

The initial population consists of $N$ randomly generated chromosomes, where $N$ stands for the population size. As illustrated in Fig. 7, the chromosomes are created by using either of the two methods – a random batching or a random sequencing heuristic. Primarily, a total of $2 \times N$ individuals are generated, and their fitness values be evaluated. Then, the best $N$ chromosomes out of the total will be chosen to form the initial population. Each member of the initial population are then treated with unary mutation operators joining the evolution cycle.

### Random batching method

0. Let $\Omega_g$ be the set of job indices in family $g$, for $g = 1, 2, \ldots, G$.

1. Assign all jobs in a family to the same batch: $b_k = g_k$, for $k = 1, 2, \ldots, G$.

2. Fix number of batches randomly: $B^* = \text{RND}(G, B_{\text{max}})$.

3. Create new batches by splitting existing ones: $B \leftarrow G$;

   DO
   
   Select a random batch: $b^* = \text{RND}(1, B)$;
   
   Split $b^*$ into two halves: $b^* \equiv \{c^1, d^1\}$;
   
   Replace $b^*$ by $c^*$;
   
   Add a new batch:
   
   $B \leftarrow B + 1$;
   
   Copy $d^1$ onto $b^*$;
   
   WHILE ($B < B^*$);

4. Randomize the sequence of batches:

   Update batch indices accordingly;

---

### Random sequencing method:

0. Let $\Omega_i = \{j_1, j_2, \ldots, j_n\}$ be the set of job indices;

1. Randomize the jobs sequence:

   Let $\pi = \{j_{i1}, j_{i2}, \ldots, j_{in}\}$ be their ordered sequence;

2. Create first batch:

   $k \leftarrow 1$;
   
   Assign $j_{i1}$ to $b_{i,k}$;

3. Batch the jobs successively:

   FOR $i = 2$ to $i = n$ DO
   
   IF ($j_{i0}$ and $j_{i-1}$ are of d/t family)
   
   Create new batch:
   
   $k \leftarrow k + 1$;
   
   Assign $j_{i0}$ to $b_{i,k}$;

   ELSE
   
   Retain old batch:
   
   Assign $j_{i0}$ to $b_{i,k}$;

   ENDIF
   
ENDFOR

---

Fig. 6. Queuing simulation of the production flow.

Fig. 7. Two heuristics used for random generation of chromosomes.
5.4. Recombination (crossover operation)

Crossover is a binary operation which combines genetic information from two randomly selected parent chromosomes with the purpose of breeding better offspring chromosomes. It exchanges the genetic material of the two parents in order to produce one or two new members for the next generation. The exchange of genes in the process is intended to search for better individuals that improve the good properties of their parents [10]. To perform crossover, two parents are selected from the mating pool at random. In order to allow better individuals to become parents of the next generation, up to 5% of the best members are included in the mating pool automatically while the remaining 95% of the population compete with equal selection probability of \( p_c \). Once a crossover operation has been performed on two mating parents, the new offspring chromosome may be accepted if its fitness is not inferior to that of the worst member in the current population (i.e., \( F_i < F_{\text{max}} \)). Since different number of batches can be created in each parent chromosome, the crossover operators we have considered treat the genes at the detail job string level. At each stage of operation, the processing time of every job in a batch is somehow equal. Therefore, jobs within a batch are arranged based on the earliest due date (EDD) sequencing rule.

In our implementation of GA, we have utilized three different problem specific chromosome operations as shown next.

*Job based uniform crossover (JUXO)*: The operator works as follows:

1. Select two parent chromosomes from the mating poll.
2. For each parent, extract the jobs included in each batch and build a job based string.
3. Choose two randomly selected positions \( P_1 \) and \( P_2 \), and block the segment of jobs in between.
4. Copy the blocked jobs from parent-1 to offspring-1 in their respective positions.
5. From the string of jobs in parent-2, select a job in order, which is not a member of the blocked segment in parent-1 and copy to the lowest vacant position of offspring-1.
6. For the resulting string in offspring-1, batch adjacent jobs together if they belong to the same family, or separately otherwise.
7. Update all the genetic information in offspring-1 accordingly.
8. Reversing the roles of parent-1 and parent-2, repeat from step 4 to create offspring-2.

**Example.** Suppose the string of jobs in the two mating parents be \( \{J_3,J_2,J_7,J_1,J_5,J_4,J_6\} \) and \( \{J_5,J_6,J_4,J_7,J_2,J_1,J_3\} \), respectively, as shown in Fig. 8. If \( P_1 \) and \( P_2 \) are randomly chosen to be the second and fifth positions, we shade all jobs from \( P_1 \) to \( P_2 \) inclusive. Next, the shaded jobs from each parent are copied to the same location of boxes in the two offsprings. To fill out the empty boxes of offspring-1, we look into the sequence of jobs in the second string and vice versa. The first potential candidate from the second string to fill out the left most empty box of offspring-1 is \( J_5 \). Since \( J_5 \) is a member of the shaded segment in offspring-1, it fails to satisfy the selection criteria. Moving forward one more step along the same string, we find \( J_6 \). Since \( J_6 \) is not included in the shaded part of offspring-1, it will be eligible to be placed at the first position of offspring-1. Next we find \( J_4 \), which satisfies the criteria to fill out the second vacant box. Continuing the process in the
same way, we will get $J_2$ and $J_3$ that can meet the selection condition to fill out the two remaining vacant boxes, respectively. Likewise, the empty boxes in offspring-2 are filled by copying unscheduled job indices from the first string, which are not members of the shaded block in parent-2.

**Job based cyclic crossover (JCXO):** This operator works as follows:

1. Repeat steps 1 and 2 of JUXO.
2. Constrict the cycle as follows:
   (a) Start with a randomly selected job position $P_1$ from parent-1.
   (b) Include $P_1$ in the cycle and record the corresponding job.
   (c) Find a new position $P_i$ indicating the recorded job in parent-2.
   (d) Add $P_i$ to the cycle and record the corresponding job in parent-1.
   (e) Repeat steps (c) and (d) if $P_i \neq P_1$.
3. For the corresponding positions in the cycle, copy the indicated jobs from parent-1 to offspring-1 in their respective places.
4. Select the unscheduled jobs from the string in parent-2, and copy them into the vacant positions of offspring-1 sequentially.
5. Batch adjacent jobs in the string of offspring-1 together if they belong to the same family, or separately otherwise.
6. Update all the genetic information in offspring-1 accordingly.
7. For the case of offspring-2, repeat the steps from 3 to 6 by swapping the tasks of the two parents.

**Example.** As illustrated in Fig. 9, suppose the string of jobs in the two mating parents be the same as the previous example. Let the second position be our first point $P_1$ to start the cycle. In the first string, $P_1$ indicates to the position of $J_2$. Then, we look for the position of $J_2$ in the second string (i.e., to fifth position) and fix point $P_2$. Since $P_2$ points to $J_5$ in the first string, we search for $J_5$ in parent-2 and fix point $P_3$. In the same way, $P_4$ is shown by looking to the position of $J_3$ in the second string. The job represented by $P_4$ in parent-1 is $J_6$. Continuing the search for $J_6$ in parent-2, we will rest on the second position, which corresponds to our initial point $P_1$. At this point, we terminate our search for new positions as the cycle will repeat itself. Next, we shade all the jobs inside the boxes pointed out by points $P_1, P_2, P_3$ and $P_4$. The shaded jobs from each parent are then copied to the two offsprings in their respective locations. The remaining task is to fill out the empty boxes in each offspring by pasting unscheduled jobs sequentially from the string of jobs in the other parent, as we did in the previous example.

**Batch based cyclic crossover (BCXO):** This operator works as follows:

1. Repeat steps 1 and 2 of JUXO.
(2) Constrict a batch based cycle as follows:
   (a) Choose a starting position \( P_1 \) randomly from parent-1, add it to the cycle and record the indicated job.
   (b) Mark those jobs around \( P_1 \) which belong to the same batch.
   (c) Find a new position \( P_i \) in parent-2 containing the recorded job.
   (d) Mark those jobs around \( P_i \) which belong to the same batch.
   (e) Add \( P_i \) to the cycle and record the corresponding job in parent-1.
   (f) Repeat steps (c) to (e) until \( P_i \) rests on any marked job in parent-1.

(3) Copy all marked jobs from parent-1 and paste to their own places in offspring-1.
(4) Select unscheduled jobs sequentially from the string in parent-2, and copy the batch of jobs from parent-1 into the lowest vacant position(s) of offspring-1.
(5) In the job sequence for offspring-1, batch adjacent jobs together if they belong to the same family, or separately otherwise.
(6) Update all the genetic information in offspring-1 accordingly.
(7) To create offspring-2, repeat from step 3 by exchanging the roles of the parents.

**Example.** Suppose the two parents contain 10 jobs which are grouped into five batches. In the sequencing-string, the jobs belonging to the same batch are bounded by the smaller boxes as shown in Fig. 10. Let our starting point \( P_1 \) be at the position of \( J_5 \) in parent-1. In order to locate \( P_2 \), we search for the location of \( J_5 \) in the second string and shade the entire batch containing \( J_5 \) in both parents. The job indicated by \( P_2 \) in the first string is \( J_{10} \). Therefore, we look into the position of \( J_{10} \) in parent-2 to fix point \( P_3 \) and shade the corresponding batches. Under \( P_3 \) is \( J_6 \) which will direct us to the starting point \( P_1 \) in the second string. After shading the batches containing \( J_6 \) in both parents, we terminate the cycle. All shaded batches from each parent are then copied to the offsprings as they are. To complete offspring-1, we look for the unscheduled jobs in the second string. Since \( J_3 \) is not scheduled in offspring-1, we copy the batch containing \( J_3 \) from parent-1 to the left vacant positions. The two remaining empty boxes in offspring-1 are then filled by copying the last batch containing \( J_4 \) from parent-1. Likewise, the empty boxes in offspring-2 are filled by copying batches from parent-2, based on the sequence of unscheduled jobs in the first string.

5.5. **Mutation**

Mutation operators are mainly used to prevent premature convergence. Mutation is also used to make local searching around a given solution, or to reintroduce lost genetic material and variability in the population. This operator can also be seen as a simple form of local search. Mutation is a way of enlarging the search space. It acts to prevent the selection and crossover from focusing on a narrow area of the search space or from the GA getting stuck in a local optimum. By mutating an individual, we slightly alter the chromosome,
thus allowing a new but similar permutation. After undergoing mutation, if the fitness value improves, then the mutated individual will automatically replace its parent. However, if its fitness is less than that of the parent but still higher than the worst member of the population, then it will be allowed to join the current population as a new offspring chromosome. On the contrary, if the fitness of the mutated individual is less than the worst fitness member, it will be rejected automatically at its time of birth. Mainly three different mutation operators are proposed in the literature for permutation encodings: Swap, Position, Shift [21,10]. We have implemented a number of mutation operators in our application of GA:

**FLIP mutation:** Here, the sequencing-string of a member is partitioned into two parts at a randomly chosen location. The block of batches in the second portion are then cut from the string and placed in front of the first portion. Next, all genetic information in the modified individual is updated accordingly.

**SWAP mutation:** Two batch positions are chosen randomly from the sequence-string of a member. The genetic material within the two batches is then exchanged giving a variation to the original chromosome.

**SHIFT mutation:** In this case, a batch at a randomly chosen position is picked from a sequence-string and relocated adjacent to another randomly picked position, causing the string of batches in between to be displaced by one step in the reverse direction. The genes are then updated accordingly.

**SHUFFLE mutation:** This works by choosing two positions from the sequencing-string of a member at random. The orders of batches in between the two positions are then randomized, and their genes modified accordingly.

**INVERT mutation:** This also works by randomly picking two positions in the entire string of a member and reversing the order of the batches in between the two positions.

**MERGE mutation:** In this case, families of jobs, which are divided into two or more batches, are first identified for a given member. Then, two batches containing jobs from a randomly chosen family are selected. Giving all of its genes to the first batch, the second batch destroys itself in the process and pulls the block of batches after it by one step. The genetic information of the variation member is then updated.

**SPLIT mutation:** Here, a randomly picked batch consisting of two or more jobs is selected from a sequence-string of a member. Its genes are then divided into two parts with random sizes. The first part replaces the original batch while the second part introduces a new batch positioned at the end of the string.

**WSPT-mutation:** Here the problem is first conceived as a single machine flowtime problem and the batches in the entire string are arranged by a weighted shortest processing time (WSPT) rule. This sequencing rule minimizes optimally the weighted flowtime of independent jobs for the case of one-machine problem without a setup. In order to combine the processing time of each batch at the two stages, the operator introduces a parameter. The operator works as follows:

1. For a given member, let $\Omega = \{b_1, b_2, \ldots, b_B\}$ be the set of batches in the sequence-string.
2. Select a random value of $0 \leq \alpha \leq 1$.
3. For each batch, calculate the combined processing time by: $P_b = \alpha \cdot P_{1,b} + (1 - \alpha)P_{2,b}$ for $\forall b \in \Omega$.
4. Arrange the batches in increasing order of $P_b$.
5. Modify the genes in the variation chromosome accordingly.

**JOHNSON mutation:** This operator follows a modified Johnson's heuristic to alter the sequencing-string of a member. The classical Johnson's algorithm optimally minimizes the makespan ($C_{\text{max}}$) of independent jobs for the case of two-machine flowshop problem without a setup. The operator works as follows:

1. Let $\Omega = \{b_1, b_2, \ldots, b_B\}$ be the set of batches in the sequence-string.
2. Partition $\Omega$ in two subsets ($U$ and $V$) based on the dominant processing time: $U = \{b \in \Omega : P_{1,b} \geq P_{2,b}\}$ and $V = \{b \in \Omega : P_{1,b} < P_{2,b}\}$ for $\forall b \in \Omega$.
3. Arrange the batches in $U$ in increasing order of $P_{1,b}$ and the batches in $V$ in decreasing order of $P_{2,b}$.
(4) Append the block of batches in $V$ next to those in $U$ to get a complete string.
(5) Update the genes in the member accordingly.

**ADOPT mutation**: This operator simply introduces a randomly created chromosome to the current population. Its operation is conditioned to the gap between the objectives of the best and worst chromosomes in the current population. When the gap gets smaller and smaller at higher generations, the operator introduces a new chromosome to replace the worst member. It basically helps to improve population diversity in the coming generations by allowing more chromosomes to survive.

Repeated execution of the GA simulation reveals that, the crossover operators considered are more efficient at lower generations while the mutation operations are more efficient at higher generations. However, the relative importance of each operator varies from one problem instant to another.

### 5.6. Survivor selection mechanism

The population size from generation to generation is supposed to be constant. Thus, survivor selection mechanism is used to choose the set of individuals from the current parent and offspring population, which are allowed to exist in the next generation. We basically implemented a combination of two well known strategies in order to decide whether a member has to live or die. Among the survivor population, $P_E\%$ are selected using a biased (Elitist) mechanism in which only the best fit chromosomes are chosen. The remaining $1 - P_E\%$ of the survivors are then selected by a fitness proportional (Roulette wheel) mechanism. The parameter $P_E\%$, which controls the boundary between the two strategies, can be referred as ‘degree of elitism’.

### 5.7. Termination criterion and parameters tuning

The control parameters which affect the efficiency of GA are: the population size $N$, the parent selection probability $p_c$ in the mating pool to undergo a crossover operation, and the probability of mutation $p_m$. In our implementation of GA, if the best fitness level cannot improve after 200 additional generations, we will terminate the algorithm. For a given problem input data, we execute the algorithm with assigned parametric values and record the best solution obtained. The CPU runtime and maximum generation. In order to ensure that the GA is not trapped by local optima, we repeat the numerical simulation with different set of values of the control parameters (i.e., population size, operators’ probability, degree of elitism, etc.) and record the best solution obtained from the experiment. From our design of experiment (DOE), we observe that the parameters are insensitive for a wide interval of values proving the GA to be robust.

**Numerical example-2**: We considered a medium-sized problem with 60 jobs and 10 family groupings. The job orders came from six equally valued customers. Based on their customer-ID, the jobs are indexed as $J_1, \ldots, J_{60}$. The set of job indices in each family group and along with their fuzzy setup and unit processing times are given in Table 3.

### Table 3

<table>
<thead>
<tr>
<th>Index $g$</th>
<th>Setup time $\hat{a}_{1,g}$</th>
<th>$\hat{a}_{2,g}$</th>
<th>Unit processing time $\hat{q}_{1,g}$</th>
<th>$\hat{q}_{2,g}$</th>
<th>Set of jobs per family $g : \Omega_g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>461 ± 24</td>
<td>766 ± 34</td>
<td>337 ± 23</td>
<td>538 ± 38</td>
<td>${J_{12}}$</td>
</tr>
<tr>
<td>[3]</td>
<td>272 ± 28</td>
<td>642 ± 32</td>
<td>234 ± 24</td>
<td>547 ± 35</td>
<td>${J_{57}}$</td>
</tr>
<tr>
<td>[8]</td>
<td>478 ± 28</td>
<td>792 ± 34</td>
<td>379 ± 28</td>
<td>468 ± 44</td>
<td>${J_{22}}$</td>
</tr>
</tbody>
</table>
processing times is given in Table 3. The production setup consists of two and three identical machines at stage-1 and stage-2, respectively.

Using the most possible values for the imprecise setup and processing times, we tested the GA for about 20 times with different set of control parameters. Accordingly, we select the following combination of parameters in solving the problem: population size $N_p = 100$, crossover probability $p_c = 0.15$, mutation probability: $p_m = 0.1$, and percentage of elitist selection $P_E = 50\%$. The best incumbent solution is obtained after 500 cycles of generations. The resulting sequence batches and the job indices included in each batch are tabulated in Table 4. To account for the fuzziness of the objective function, next we determine its deviation from the mean by re-evaluating the incumbent solution using the maximum deviation values of the setup and processing times. Finally, we found the imprecise total flowtime to be $F \approx 675706 \pm 58270$. The GA is executed on a Pentium(R)4 PC with 3 GHz speed and 1 GB RAM, and took a CPU runtime of 4:27 min to converge.

The proposed GA has been tested for different problem instances and the results obtained are compared with the LINGO solution. For all cases, the mean values of the setup and unit processing time vary uniformly over the intervals indicated in Table 5. Considering the most likelihood value $z^m$ to compare the two approaches, Table 6 shows summary of results obtained for 10 different instances of problems. The

\begin{table}
\centering
\caption{Example-2: Batches created according to their sequence of operations}
\begin{tabular}{ll}
\hline
Set of jobs per batch $b$: $B[0] = \Omega_0$ & $B_{[1]} = \{J_{10},J_{13}\}$ \\
& $B_{[2]} = \{J_{2},J_{5},J_{38},J_{56}\}$ \\
& $B_{[3]} = \{J_{11},J_{21},J_{23}\}$ \\
& $B_{[4]} = \{J_{1},J_{3},J_{6},J_{8},J_{12}\}$ \\
& $B_{[5]} = \{J_{7},J_{9},J_{17},J_{20},J_{29},J_{30}\}$ \\
& $B_{[6]} = \{J_{4},J_{16},J_{19},J_{33},J_{42}\}$ \\
& $B_{[7]} = \{J_{15},J_{26}\}$ \\
& $B_{[8]} = \{J_{18},J_{25},J_{28},J_{31},J_{33},J_{36},J_{39},J_{48},J_{50}\}$ \\
& $B_{[9]} = \{J_{14},J_{34},J_{37}\}$ \\
\hline
\end{tabular}
\end{table}

\begin{table}
\centering
\caption{Range of values for the input parameters}
\begin{tabular}{ll}
\hline
Parameter & Range of values \\
\hline
Setup time at stage-I, $a_1$ & (200, 500) \\
Setup time at stage-II, $a_2$ & (800, 1200) \\
Unit processing time at stage-I, $q_1$ & (100, 400) \\
Unit processing time at stage-II, $q_2$ & (500, 1000) \\
\hline
\end{tabular}
\end{table}

\begin{table}
\centering
\caption{Comparison of results for different problem instances}
\begin{tabular}{llllllll}
\hline
PR. code & Tools problem size & & Best solution ($\times10^6$) found andCPU runtime (s) & \\
Jobs $J$ & Groups $G$ & Orders $K$ & Machines $m_1$ & $m_2$ & GA & LINGO & \\
\hline
$T_1$ & 5 & 2 & 2 & 1 & 1 & 2 & 0.0209 & 6.86 & 0.020901 & 0.020901 & 7 s \\
$T_2$ & 10 & 3 & 2 & 2 & 2 & 0.0588 & 13.75 & 0.060655 & 0.030925 & 2.5 h \\
$T_3$ & 25 & 5 & 4 & 2 & 3 & 0.2029 & 34.06 & 0.224751 & 0.015642 & 1 h \\
$T_4$ & 50 & 10 & 6 & 2 & 4 & 0.5875 & 95.09 & 0.727642 & 0.029242 & 12 h \\
$T_5$ & 75 & 15 & 7 & 3 & 4 & 1.2415 & 240.05 & $\times$ & $\times$ & $\times$ \\
$T_6$ & 100 & 18 & 10 & 3 & 5 & 1.7665 & 145.75 & $\times$ & $\times$ & $\times$ \\
$T_7$ & 150 & 25 & 12 & 4 & 6 & 2.9734 & 331.69 & $\times$ & $\times$ & $\times$ \\
$T_8$ & 200 & 30 & 15 & 5 & 8 & 4.3141 & 1508.83 & $\times$ & $\times$ & $\times$ \\
$T_9$ & 250 & 32 & 20 & 6 & 10 & 4.8936 & 1364.39 & $\times$ & $\times$ & $\times$ \\
$T_{10}$ & 300 & 35 & 25 & 10 & 15 & 4.9653 & 5494.22 & $\times$ & $\times$ & $\times$ \\
\hline
$x$ = problem oversized to be solved by LINGO. \\
\hline
\end{tabular}
\end{table}
computational experiment confirms that the proposed GA gives better quality of solution with medium computational effort especially for larger problem sizes.

6. Conclusion

In this paper, we have demonstrated a mixed-integer fuzzy programming (MIFP) approach for batch scheduling of jobs on parallel machines in a two-stage flowshop. The goal of the fuzzy model is to improve customer responsiveness by reducing the total weighted flowtime for all jobs in the system. The fuzzy mathematical programming approach incorporates the uncertainties associated with estimation of time dependent parameters directly into the optimization model. Such representations of fuzzy parameters with membership functions avoid the need to perform sensitivity analysis after an optimal solution is obtained. The uncertainty associated with batch-processing and setup times are represented by triangular fuzzy sets. Numerical results demonstrate that the proposed model can efficiently schedule small size of jobs. While for higher number of jobs, a robust GA solution method is suggested. The proposed model is vital as a sound decision support tool in the practical application of furniture scheduling, especially when the processing and setup times cannot be determined precisely.

References

[34] H.J. Zimmermann, Fuzzy programming and linear programming with several objective functions, Fuzzy Sets Syst. 1 (1978) 45–56.