Cost based Rescheduling Approach to Handle Disruptions in Flexible Manufacturing Systems

Ehsanallah Naseri

A Thesis

in

The Department

of

Mechanical and Industrial Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Industrial Engineering) at

Concordia University

Montreal, Quebec, Canada

August 2010

Your file Votre référence
ISBN: 978-0-494-80151-2
Our file Notre référence
ISBN: 978-0-494-80151-2

Abstract

Cost based Rescheduling Approach to Handle Disruptions

in Flexible Manufacturing Systems

Ehsanallah Naseri

Rescheduling is an essential operating task to efficiently tackle uncertainties and unexpected events frequently encountered in today's complex and flexible manufacturing systems. The main purpose of this thesis is to develop a real time reactive scheduling methodology in order to respond to such disturbances and uncertainties in a cost efficient manner. In order to assess the impact of schedule changes, a compound rescheduling cost function is developed based on machine, job, and material related rescheduling activities.

A Total Rescheduling (TR) approach based on the Filtered-Beam-Search-heuristic algorithm (FBS) is proposed to generate a prespecified number of cost efficient suboptimal schedules by using the proposed cost function in case of each disruption. Thereafter, the current schedule is replaced by the alternative schedule which causes the minimum rescheduling cost.

Responding to each single disruption with TR may cause system nervousness and increase the operational cost. Hence, a partial rescheduling approach is developed by a Modified Filtered-Beam-Search-heuristic algorithm (MFBSR) in order to generate a prespecified number of sub optimal cost-efficient schedules with a lower rescheduling cost and fewer deviations than TR.

In order to validate the performance of the proposed methodologies, TR and MFBSR, different case studies and experimental designs have been performed considering various disruption scenarios. The performance of the suggested methods in terms of rescheduling cost, makespan efficiency and stability have been compared with similar rescheduling and repair methods in the literature. The results reveal that the proposed methodologies could be considered as competitive methods in responding to disruptions in flexible manufacturing systems.

"I do not know what I may appear to the world;
But to myself I seem to have been only like a boy
playing on the seashore, and diverting myself in now
and then finding a smoother pebble or a prettier shell
than ordinary, whilst the great ocean of truth lay all
undiscovered before me."

Isaac Newton

AKNOWLEDGEMENTS

First and foremost, I would like to express my deepest appreciation to my supervisor, Professor Onur Kuzgunkaya, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. This thesis would not have been possible unless his guidance and encouragement. His deep understanding of seemingly every aspect of manufacturing system and scheduling has never ceased to amaze me. He led me into the rescheduling field and cost analysis, and taught me how to think of ideas, prove concepts and write a scientific report. I have learnt a lot about the methodology of research and life from him. One simply could not wish for a better or friendlier supervisor.

I would like to thank my family; my father, who taught me to be eager in learning, asking questions and breaking complex problems into simpler parts, and who demonstrates the work ethic to me that I can only hope to approximate. My dear mom, who always pushed me to do the best and be the one. My sister and brother, for their love, enthusiasm and warm support. None of this would have been possible without their encouragement. They spared no sacrifice to support me in pursuing my dreams.

I am also thankful to my friends who helped me and encouraged me a lot during my research, Mr. Alireza Rahimi vahed, Mr. Mohsen Eftekhari hesari, and Mr. Ehsan Rezabeigi for sharing thoughts and advices. I would like to thank all my friends with whom I have been lucky enough to explore the plethora of opportunities for activities outside of the university and I hope our friendship will be long lasting after the school.

Finally, I want to express my gratitude to all the committee members for their time and willingness to read and critique my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AGA	Adaptive Genetic Algorithm
AGV	Automated Guided Vehicle
AI	Artificial Intelligence
AOR	Affected Operation Rescheduling
B&B	Branch and Bound
BS	Beam Search
C_B	Cost Based rescheduling
Dev	Deviation
DR	Dispatching Rule
FBS	Filtered Beam Search
FLX	Flexibility level
FMS	Flexible Manufacturing System
GA	Genetic Algorithm
HFBS	Heuristic based Filtered Beam Search
M_SPT	Modified Shortest Processing Time
MAG	Magnitude of disruptions
mAOR	modified Affected Operation Rescheduling
MFBSR	Modified Filtered Beam Search Rescheduling
MKs	Makespan
MTD	Method of rescheduling
OBJ	Objective function
Q_Trd	Quadratic function of Tardiness
RSR	Right Shift Rescheduling

SA	Simulated Annealing
Stb	Stability
TIM	Timing of disruptions
TOD	Time Of Disruption
TR	Total Rescheduling
TS	Tabu Search
W_Mks	Weighted sum of Makespan
WIP	Work In Process

LIST OF NOTATIONS

$Cost_{Rsch}$	Rescheduling Cost
Cost(Machine_R)	Machine Related Cost
Cost(Job_R)	Job Related Cost
Cost(Mat_R)	Material Related Cost
P_{ijk}	Process time of operation i of job j on machine k
C_{ijk}	Completion time of operation i of job j on machine k
D_j	Due date of job j
S_{ijk}^{new}	Starting time of the operation i of job j on machine k after schedule update
$S_{ijk}^{initial}$	Original starting time of the operation i of job j on machine k
$Y'_{ijkk'}$	Binary variable indicating whether the operation i of job j on machine k is switched to machine k' or not
b	Beamwidth
f	Filterwidth
PS_l	Partial schedule containing l scheduled operations
N	Total number of nodes generated at each level
η_k	Cost coefficient of idle time machine of k
δ_j	Cost coefficient of unit time lateness for job j
θ_j	Cost coefficient of saving on due dates for job j
μ_{ij}	Cost coefficient of expediting cost of material for the operation i of job j

h_{ij}	Cost coefficient of holding cost of material for the operation i of job j
$\omega_{ijkk'}$	Cost coefficient of reallocating the material between machines after rescheduling process
τ	Cost coefficient of extra machining time needed after rescheduling
$Mks_{Initial}$	Makespan of the initial schedules
Mks_{New}	Makespan of the updated schedules
EFF	Efficiency of the updated schedule
DevSt	Normalized deviation of updated schedule
St'_{ij}	Starting times of operation i of job j of the updated schedule
St_{ij}	Starting times of operation i of job j of the original schedule
k	Total number of jobs
p	Total number of operations
$S^{initial}$	List of all scheduled operations in the initial schedule sorted based on their starting times
S_l^{temp}	Set of schedulable operations at level l
S_l^{new}	Subset of S_l^{temp} consisting of operations with minimum starting time according to the initial schedule
s_{ij}^{new}	Earliest possible starting time of operation in the S_l^{new}
γ_{ij}	Minimum processing time of operation i of job j over the capable machines
M_avl_k	Updated available time for machine k
O_{ij}	Operation i of job j

s_{ijk}^{new}	Updated starting time of operation after rescheduling
IS_l	Initial schedule at level l
RL_l	Remaining list at level l

Chapter 1

Introduction

In order to survive and be globally successful in today's competitive manufacturing environment, companies have to respond to changes in the market quickly and satisfy the needs related to mass customization through flexibility and adaptability.

Flexible manufacturing systems (FMS) possess the capability of handling these changes and disruptions thanks to the machine and process planning flexibilities; however, these capabilities should be efficiently exploited through scheduling rules in order to get the full benefit at minimal operational costs. In a dynamic environment, the task of managing and controlling manufacturing systems becomes more difficult as a result of internal disturbances such as machine failures, and external disturbances such as rush orders and supplier problems. Scheduling is an essential task in achieving timely and cost effective production. The omission of the dynamic nature and stochastic events in scheduling literature creates a gap between scheduling theory and practice. Once an initial schedule is disrupted, it should be updated through rescheduling activities to

satisfy the new requirements. Machine failure, new order arrival, job cancellation, due dates changes, job priority changes, rework or quality problems, and operator absenteeism are some of the disruptions that may occur widely during the production schedules.

Rescheduling refers to finding a new schedule when a disruption occurs in the operations of an on-going initial schedule. Two important factors that need to be considered in rescheduling problems are when and how to efficiently react to such disruptions. There are three types of policies in order to find the proper time to respond to disruptions: event driven rescheduling, periodic rescheduling and hybrid rescheduling. As well, three methodologies are used to efficiently react to disruptions: Right Shift Rescheduling (RSR), partial rescheduling (repair) and total rescheduling (TR) [1]. Finding an appropriate method and policy in responding to disruptions could prevent system nervousness and decrease operational cost as a result.

1.1 Flexible Manufacturing Systems

A flexible manufacturing system (FMS) is an integrated system of machine modules (usually CNC machines), equipped by an automated material handling and storage system under computer control for the automatic random processing of palletized parts [2]. An FMS is a form of job shop system in which machines are capable of performing various operations based on their tool assignment. Figure 1.1 shows an FMS implemented at Vought Aerospace Co. Workstations in the system are interconnected by an automated material handling system by using the Automated Guided Vehicle (AGV) [2]; hence, jobs have various routes in the system and can be carried out in any sequence.

The processing time of each operation may vary from one machine to the other based on the toolset and machine's specification.

Flexibility is the term used for the characteristic which allows a manufacturing system to cope with variation of production style without an interruption in the production process for any changeovers between models. Exploiting this attribute into the system provides high machine utilization and throughput rates, and decreases lead-time and work in-process inventory by reducing parts movement and tool changing time [3].

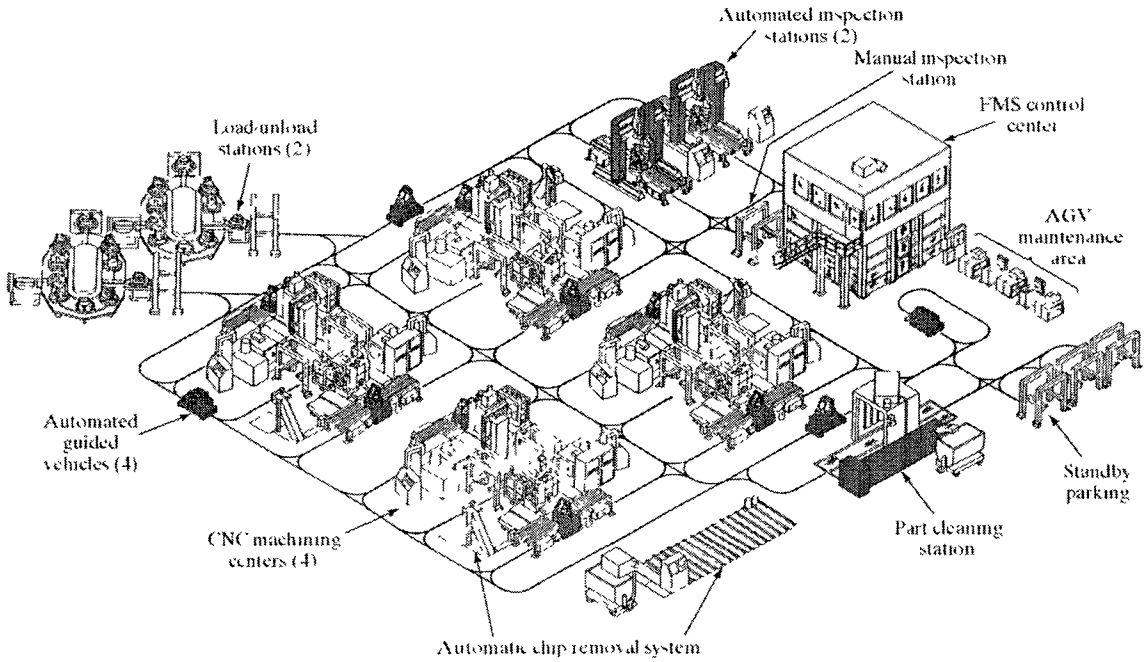


Figure 1.1 : FMS implemented at Vought Aircraft [2]

1.2 Uncertainty and Disruptions in FMS

Manufacturing operations are inevitably faced with wide range of uncertainties and variations in production process. Companies need to handle them in advance or react after their occurrence. Uncertainty and unexpected events may change the system status

and affect the performance. The production schedule employed as a crucial tool in manufacturing systems in order to increase productivity and decrease the operating cost, is subject to be upset widely by disruptions during the execution schedules. If these disruptions cause significant deterioration in performance, the system needs to react and update the existing schedule in order to lessen the impact [1]. Machines may be stopped during their operations for major failures such as breakdown, toolset wearing, or tool reassignments. New jobs may enter to the system or the existing jobs may be cancelled due to changes in customers' orders. All of these events which are called *rescheduling factors* [4], will affect the performance of the existing schedule and require a quick response. 19 types of rescheduling factors found in literature which occur in the FMS can be categorized as follows:

1. Machine breakdown: [5-14]
2. New order arrival: [11,14-17].
3. Rush (urgent) order arrival: [5,9,10,13,14,18].
4. Order cancelation: [10,13,19-21].
5. Maintenance of machine: [5].
6. Operator absenteeism: [11,22].
7. Tool Breakdown: [11].
8. Tool Wearing: [11].
9. Delay in material handling process: [23].
10. Due date changes: [14,23].
11. Process time variation: [11,13,14,24].
12. Performance variation in machines: [11].
13. Set-up time variation: [11].

14. Change in job priority: [10,11].
15. Rework or quality problem: [11,22].
16. Rejection: [11,14].
17. Unavailability of raw material: [11].
18. Outsourcing: [11].
19. Ready-time changes [14].

1.3 Problem Statement and Motivation

Due to the dynamic nature of flexible manufacturing systems and the wide range of unexpected changes and disruptions, the production schedule as a crucial tool in manufacturing process needs to be monitored and updated. Analyzing how to efficiently handle the disruptions and uncertainties in FMS scheduling is the main purpose of this work. Rescheduling process identifies when and how to react to such disruptions in FMS, mentioned in Section 1.2, in order to make the schedules work efficiently. Towards this end, a comprehensive review of literature is performed in Section 1.4, and the contribution of this study is presented in Section 1.5, in order to find the gap between the methodologies and contributions in the literature, and finally the objective of the thesis is presented in Section 1.6.

1.4 Literature Review

The literature on rescheduling topic can be broadly classified into two main areas: rescheduling methods and strategies in FMS environments, and selection of performance criteria to generate and evaluate the updated schedules. The review of literature on rescheduling methods and policies are presented in Section 1.4.1, and the performance

measures used in rescheduling of relevant works are discussed in Section 1.4.2. Finally, Table 1.1 represents the summary of rescheduling literature and the corresponding features of each work, in terms of rescheduling methods and performance criteria.

1.4.1 Rescheduling

Rescheduling is the process of updating the existing schedule in response to the disruptions and changes that take place on the FMS in order to keep the system running at high performance [1]. During the execution of a schedule, two essential factors need to be considered. First, the timing for a rescheduling decision needs to be made, which is called *when-to-schedule*. Once the decision on timing of updating the schedule is made, the second decision is how to perform the rescheduling action called *how-to-schedule* [1,24-26].

Regarding the timing decision, previous studies implement a periodic, event-driven, or a hybrid rescheduling policy in order to determine an appropriate time to react to disruptions and changes. Periodic scheduling generates the schedules based on a constant or variable length time by time. According to this policy, the system is monitored periodically and the rescheduling action is run at the beginning of each time period in order to recover the system from negative impact of disruptions [24].Sabuncuoglu and Karabuk [27] propose another method called adaptive scheduling, which triggers the rescheduling action after a predetermined amount of deviation from the existing schedule. The authors also show that frequently updating the schedule performs actually worse than myopic dispatching rules. Event-driven scheduling updates the existing schedule at each disruption occurrence point. Subramaniam *et al.* [18] propose a reactive scheduling repair methodology in order to handle multiple disruptions

occurred during scheduling horizon. The authors expressed that rescheduling the system at the time of each disruption is a convenient solution for job shop systems. The hybrid rescheduling policy refers to the method of updating the existing schedules not only at the end of each fixed time, but also in response to each disruption occurrence [22,28].

How-to-schedule term relates to the methods in which the updated schedules are generated. The schedule can be generated online or offline. Sabuncuoglu and Goren [14] define this aspect as the schedule scheme and Vieira *et al* [1] name it as the rescheduling strategy. Offline scheduling or predictive-reactive has two major steps: First, all operations of available jobs are scheduled before executing the schedule for the entire horizon, and in second step, schedules are updated in response to disruptions. In online scheduling, however, there is no scheduling generation and the decision is made once at a time during the schedule execution. The online scheduling requires the knowledge of state of the system at the moment of time. This knowledge is combined with decision rule to determine the next operation to be scheduled. Dispatching rules or control theory are such examples of online scheduling [27,29,30]. Sabuncuoglu and Karabuk [27] show that in static and dynamic environments, offline scheduling outperforms online scheduling as the online scheduling fails to consider the global perspective given by offline scheduling. However, in a dynamic and stochastic environment, further study is needed to compare different performance measures in online and offline scheduling.

Another issue in *how-to-schedule* refers to the methodology to regenerate the schedules. Existing studies use one of the three main rescheduling methods according to a performance criterion: (i) Total Rescheduling (TR), (ii) Right-Shift Rescheduling (RSR), and (iii) Partial rescheduling (Repair). The TR approach regenerates the whole schedule from scratch for the remaining operations, using different types of objective functions,

methods and algorithms [31]. On another extreme, the RSR simply updates the schedule by right shifting all the remaining operations in time to recover the negative effects of change and disruptions [32]. Between these two extremes, the repair is introduced to partially reschedule the remaining operations [5,11,13]. More details about rescheduling methods and literature are explained in Section 1.4.1.1-3.

A variety of performance metrics are utilized in rescheduling studies. These metrics can be categorized into four groups: schedule efficiency, schedule stability, robustness, and cost [7,10,26,33]. Efficiency metrics usually refer to the time related measures such as makespan, tardiness, mean flow time, and lateness [1]. Measure of stability relates to the impact of the new schedule deviations from the original one, while robustness is concerned with the differences in terms of performance measure such as objective function values [26]. Considering cost as a performance measure could reflect the economic performance of a manufacturing system. More details about performance metrics and literature are explained in Section 1.4.2.

1.4.1.1 Total Rescheduling (TR)

Total Rescheduling (TR) which is widely called in literature as rescheduling refers to the task of regenerating the schedules from the scratch for the remaining operations in the schedule, with a predetermined objective function [13]. Rescheduling in FMS has been broadly studied during past decades, and it continues to attract the interest of researchers both in academia and industry. The main purpose of these researches is to find out how to efficiently and quickly generate the new schedule in case of disruptions. Most of research contributions could be categorized in three main aspects: solving the rescheduling

problem by optimization methods for exact solutions, dispatching rules, and artificial intelligence (AI)-based heuristic approach [20].

Mathematical programming is the major part in optimization methods used in FMS rescheduling [35]. Several mathematical programming models have been developed and used for solving scheduling problem in FMS. Han *et al.* [36] propose a nonlinear integer programming model for real-time scheduling problem in FMS. Hutchison *et al.* [37] have developed a mixed-integer programming formulation for a random FMS scheduling problem. Caumond *et al.* [38] propose an MILP model for scheduling problem in FMS. The authors have found the optimal solution for small and medium size scheduling problems. Although these methods ensure achieving the best solution for the small and medium sized problems, they require huge computational efforts due to the complexity of FMS scheduling problem for large scale problems [35]. Wang *et al.* [39] mention that the FMS scheduling problem is NP-hard. Therefore, it is not always possible to find an optimal solution quickly. Hence, the exact optimization methods are usually applied as a tool for analyzing and validating the problems and as a basis in developing the heuristic algorithms.

Dispatching rule (DR), also called scheduling rules or priority rules, is one of the most common approaches for scheduling in dynamic environments [40]. Balckstone *et al.* [41] defined a DR to select the next job to be processed from a job waiting list. The authors compare several DRs in the literature, and conclude that identifying a single DR as the best for all circumstances is impossible. Stecke and Solberg [42] have comprehensively studied the performance of DRs in FMS environments. Ishii and Talavage [43] have presented a real-time scheduling algorithm in FMS, which dynamically collects the DRs for the short period ahead responding the changes in the

system. The authors conclude that changing DRs over a short-term period based on current state of the system could perform better than using single DR for a long time. Kim and Kim [9] propose a scheduling method where DRs are dynamically varied and based on certain criteria the best one is selected at each step. Chan *et al.* [44] show that dynamically changing the DRs at a proper frequency during the dynamic scheduling could improve the performance of the system. Numerous DRs were introduced in literature, and although they can obtain the schedules quickly, their efficiency highly depends on the performance criteria and operating conditions. Accordingly, they can achieve a good result on a given performance criterion, but may cause poor results on another criterion [45].

AI-based meta-heuristics algorithm (GA, SA, TS, BS etc.) has become more popular among researchers in recent years for solving the scheduling/rescheduling problems, as they can generate near optimal solutions in real-time complex system rapidly [46]. Brandimarte *et al.* [47] have proposed a tabu search (TS) algorithm for FMS scheduling in order to minimize the weighted sum of tardiness and makespan. Dauzere-Peres and Paulli [48] have presented an integrated approach in FMS scheduling by applying TS based on a new neighborhood structure for the problem. Honghong and Zhiming [19] have introduced an adaptive genetic algorithm (AGA) to find the new suboptimal schedule of a large and complicated FMS quickly, as a response to disruptions. The proposed AGA is an improved approach of GA which can prevent the premature convergence. Jain and Elmaraghy [10] suggest a steady state GA for solving multiple routing scheduling problems (FMS). The authors use this method in generating the initial and updated schedule. Najib *et al.* [49] have introduced a modified simulated annealing (SA) method for scheduling problem in FMS, and Xia and Wu [50] have

implemented a hybrid optimization approach for FMS scheduling problem using swarm optimization and SA.

Filtered-beam-search heuristic algorithm as one the most competitive AI-based search methods is widely utilized for efficiently searching in decision tree especially with the enormous solution space [51]. Ow and Morton [51] firstly introduced the FBS as an extension and improvement of BS. The high performance and quick searching speed of FBS-based algorithms are two key elements expressed in the literature. De and Lee [52] propose a problem-solving strategy based on FBS algorithm in FMS. The authors use a frame-based knowledge representation scheme to improve the quality of generated schedules. Sabucuoglu and Karabuk [6] have introduced a heuristic based FBS for FMS scheduling problem. The authors consider finite buffer capacity, routing and sequence flexibilities, and generate the schedules for machines and automated guided vehicle (AGV) for a given period. Wang *et al.* [39] have developed a heuristic filtered beam search algorithm (HFBS) to find suboptimal schedules with a reasonable computational time in FMS. The authors have incorporated several DRs and intelligently explored the search space in order to avoid useless paths, to improve the speed and maintain the solution quality. Wang *et al.* [20] have introduced a filtered beam search algorithm to solve the dynamic rescheduling problem in a large and complex FMS environment responding to realistic disruptions. The authors have performed a comparison with the testing results of Honghong and Zhiming [19] and concluded that the results from the proposed FBS based algorithm in FMS rescheduling outperformed the results generated by AGA in both speed and accuracy aspects.

1.4.1.2 Right Shift Rescheduling (RSR)

Right shift Rescheduling (RSR) refers to the action of simply delaying the whole schedule for the duration of disruptions; or in other words, shifting the remaining operations to the right on the Gantt chart [5]. The RSR process, also referred to *Do Nothing* method, is quite simple and easy to be modeled and implemented [14]. The authors have considered 8 types of disruptions and expressed the required response actions for RSR. They conclude that RSR is suitable with Periodic scheduling. Abumaizar and Svestka [5] have compared three rescheduling methods (TR, RSR, and repair), and concluded that the performance of RSR is worse than the two others in terms of stability and efficiency and the updated schedules by RSR will have high deviations from the initial schedule. They suggest that RSR may be applicable just when the disruptions overlap with the scheduled processing time in the initial schedule. Subramaniam and Raheja [11] have compared the performance of RSR with mAOR and showed that for various types of disruptions mAOR outperforms RSR in both stability and efficiency measures. According to their results, the RSR itself cannot not be a competitive method in responding to the disruptions; it needs to be revised and combined with other methods in order to generate reasonable results.

1.4.1.3 Partial Rescheduling (repair)

Partial rescheduling, also called schedule repair in the literature, refers to rescheduling the operations that have been directly or indirectly affected by the disruptions [1,5,18,53-56]. This method intends to revise the schedule by keeping the existing schedule stability and avoiding unnecessary changes. Most of the heuristics methods developed in repair

approach are based on affected operations rescheduling (AOR) [5,10,11,54]. Match-up scheduling introduced by Bean *et al.* [57] is another type of scheduling repair methods.

Bean *et al.* [57] have proposed a repair methodology for production schedules responding to disruptions based on match-up procedure. In this process, the part of the original schedule is rescheduled in order to accommodate the disruptions to fit with the original schedule at some time in future. The authors have applied a heuristic of ordering rules in order to resequence all operations before a match-up point. Increasing lateness cost results in increasing the match-up point. If the match-up point becomes too large, the proposed method solves integer programming or DRs to reallocate operation to different machines. They conclude that match-up scheduling brings optimal results for the low frequent disruptions, which can allow the system to return to the original schedule before the next disruption occurrence. Akturk and Gorgulu [58] have proposed another repair algorithm based on the match-up method which can partially reschedule a modified flow-shop system responding to machine breakdown. Sabuncuoglu and Goren [14] suggest an extensive repair methodology for 8 types of disruptions which can update the disrupted schedule by minor modifications. They use match-up scheduling in order to response to machine breakdown.

Li *et al.* [54] have developed a heuristic algorithm based on binary tree and net change concept to update the schedule by rescheduling only the operations that needed to be revised. Abumaizar and Svestka [5] have applied the binary branching algorithm to present an algorithm for job shop rescheduling called affected operations rescheduling (AOR) in order to minimize the increase in makespan and deviations from initial schedule and overcome the deficiency of RSR. The authors have implemented AOR for machine breakdown and compared the performance of it with TR and RSR for various

disruption scenarios. They conclude that, compared to TR, AOR reduces deviations and computational time significantly, but in terms of makespan, TR performs slightly better. AOR is modified by Mason *et al.* [59] for fixed sequence rescheduling in order to consider batch-processing machines in FMS. The authors have compared the performance of schedules generated by RSR, modified AOR (fixed sequence rescheduling), and TR for machine breakdown in an FMS environment. Subramaniam and Raheja [11] have employed AOR and extended it for 17 different types of disruptions. The performance of mAOR is compared with RSR for four types of disruptions; machine breakdown, rush order arrival, process time variations and urgency of the existing jobs. They show that mAOR outperforms RSR in terms of efficiency and stability. Subramaniam *et al.* [18] have introduced a reactive repair methodology based on mAOR [11] for handling multiple disruptions that occur during the schedule horizon. The authors consider five types of disruptions: Machine breakdown, absenteeism, process time variations, unexpected order arrival, and job cancellation. They have performed an experiment based on different levels of magnitude, density, and dispersion of the mentioned disruptions for efficiency and stability in job shop environment, and compared the performance metrics by RSR. Their results also show that mAOR outperforms RSR in case of multiple disruptions, but the efficiency of mAOR deteriorates after multiple repair actions, and applying the repair approach in response to disruptions is not always recommended. The authors advise to find a proper point for TR some time during the schedule horizon to offset the poor efficiency of mAOR.

1.4.2 Performance Measures

In Section 1.1.3, four types of performance measures utilized in the rescheduling literature were introduced: measures of schedule efficiency, stability, robustness, and cost. These measures could be used as a tool or a function applied in generating the schedules or can be used as a metric for performance comparison. In this Section these measures are explored and the relevant works from literature are presented.

1.4.2.1 Measure of schedule efficiency

Efficiency measures are often used for generating a production schedule. They are generally time based measures [7]. The majority of rescheduling models optimize a time related objective function, the same one used for generating initial schedule, such as tardiness or makespan [14,20,39]. The main reason for the popularity of the makespan [11,13,23,27,28,33,39,60] or mean tardiness functions [9,10,19,20,27] is due to the fact that their primary objective is to satisfy customer needs, but none of them reflects the negative effects of changes to the manufacturing environment.

Hoitomt *et al.* [61] have proposed a weighted quadratic function of Tardiness as a metric to generate schedules. This objective function comprises the importance of due dates, values of each job, and the fact that a job becomes more critical after passing its due date. By considering a weighted quadratic function rather than weighted sum, the function reflects the incremental penalty of increasing the lateness. Honghong and Zhiming [19] and Wang *et al.* [20] have applied the weighted quadratic function into generating the initial and updated schedules. They express that this objective function is quite more useful than makespan in an actual manufacturing environment.

Xia and Wu [50], and Wang *et al.* [39] propose a weighted sum function of makespan, total workload and critical machine workload in order to generate the schedule in FMS. The authors express that since investment and installation of modern fabrication tools in FMS are highly capital intensive, there is a great need to take workload and utilization factors into account and include them in generating the schedules.

Abumaizar and Svestka [5], Subramaniam and Rehja [11], Subramaniam *et al.* [18], and Fahmy *et al.* [13] employ the percentage changes in makespan of updated schedule in defining efficiency metric for evaluating their updated schedules. As a result, the more efficient rescheduling process is the one which has lower increase in makespan.

1.4.2.2 Measure of schedule stability

A schedule which deviates minimally from the original schedule is called stable. Stability is measured to demonstrate the impact of schedule changes and can be defined in two ways: the starting time deviations between the updated and the original schedule, and as a measure of sequence difference between the two schedules [1].

Abumaizar and Svestka [5] propose the measure called starting time deviation (DevSt) in order to evaluate the stability of the updated schedules. Subramaniam and Rehja [11], Subramaniam *et al.* [18], and Fahmy *et al.* [13] define a metrics based on normalized deviations of starting times of operations from the original schedule. Sabuncuoglu and Goren [26] present six different stability based measures in order to evaluate different aspects of schedule changes impacts. The authors incorporate completion time of operations into the presented measures.

Sotskov *et al.* [62] copes with stability aspect from another perspective. The authors use a posteriori analysis to handle disruptions in a job shop environment. They

try to determine the maximum variation in the process time of operations to keep the existing schedule optimal in case of disruptions. They call this maximum variation "*the stability radius*" which can be obtained by sensitivity analysis.

1.4.2.3 Robustness

Robustness can be described as the insensitivity of scheduling performance to the disruptions. It can be defined as the difference in terms of objective function values between the updated schedule and the original one [26]. Robust schedule is the term used for the schedules whose performance does not significantly deteriorate in face of disruptions.

Leon *et al.* [63] use average system slack as a surrogate measure to estimate the expected performance degradation. They show that the average system slack as a robustness measure performs well under processing time variation. Daniels and Kouvelis [64] have generated a robust schedule for a single machine system in face of process time variation in order to minimize the performance measure under the worst possible scenario. They show that the schedule generated by robustness metrics performs better than former DRs. Goren and Sabuncuoglu [65] have developed two new surrogate measures for robustness and stability to generate the robust and stable schedules in single machine environments by considering three different measures: makespan, total tardiness or total flow time. Gan and Wirth [66] use an empirical approach and an entropy measure in order to justify the time that is needed to switch between the deterministic, robust and online scheduling.

Sabuncuoglu and Goren [14] categorize robustness measures into two groups. The measures in the first group are based on the actual performance of an updated schedule

and the other group is based on regrets. The authors propose seven measures for the first category and four measures for the second one in order to minimize the deviations between the new schedules and the initial schedule in terms of performance values.

1.4.2.4 Cost

For managers, issues such as job profitability and total cost minimization are often more important than any time based or stability based measures [7,8]. The former measures fail to reflect the economic impact of disruptions on the manufacturing systems; hence, several studies have been done to find a cost based performance measure to offset this deficiency. The authors propose the total cost function in terms of job due dates, completion time, number of jobs, number of operations, processing time, raw material cost, processing cost of operations, job revenue, processing start time job release time, job tardiness, holding cost, and lateness cost. This cost function has been used in evaluating the different scheduling rules in job shop environments.

Vieira *et al.* [1] categorize rescheduling cost into three groups: computational costs, setup costs, and transportation costs. Computational cost may refer to the cost of loading and running the scheduling system on the computers [22,27], the cost of investment in information systems (hardware and software), and the cost of administration, maintenance and upgrading of the system. Setup cost refers to changing and reallocating the toolset and pallets according to the changes in the existing schedules [56]. Transportation cost, also called material handling cost, refers to handling the materials earlier than the required time, or additional material handling work required due to changes in the existing schedules [56]. The mentioned cost measures in the literature are not quite applicable in FMS environments. In the FMS configuration mentioned in

Section 1.1.1, the system consists of flexible machines like CNC machines, which are capable of simply changing toolsets during the production, hence the setup cost is not useful any more. Furthermore, as FMS is equipped by automated material handling system (such as AGV), transportation cost also would not be applicable. In addition, An FMS has its own computer based infrastructure which controls and manages the whole system and does not need any more devices and tools for its scheduling; thus, the computational cost would also be useless. In view of these reasons, there is a great need in finding a proper cost measure in rescheduling applicable to an FMS environment.

Kapanos *et al.* [67] have introduced a rescheduling cost function and applied it into the optimization process of generating schedules to make the updated schedules more stable in real chemical industry scenarios. Their proposed rescheduling cost function consists of three main sources: total starting time deviation, unit reallocation cost, and order resequencing cost. The authors conclude that considering the rescheduling cost in generating the schedule results in smoothing the gap between theory and practice in scheduling problems.

Table 1.1: Summary of the literature review

	Litrature	[5]	[7]	[8]	[10]	[11]	[13]	[14]	[18]	[19]	[20]	[24]	[27]	[28]	[32]	[57]	[58]	[59]	[67]	[68]	[69]
Methodology	TR	√	√	√	√	√	√	√	√	√	√	√	√	√	√				√	√	√
	Repair	√				√	√	√	√			√				√	√	√			
	RSR	√				√	√	√	√						√			√			√
Performance measure	Makespan					√	√		√				√	√	√		√			√	√
	Tardiness				√					√	√		√			√		√	√	√	√
	Stability	√	√	√		√	√	√	√								√			√	√
	Robustness		√	√				√							√						
	Cost																		√	√	√
	Other				utilization flow time							flow time				utilization					

1.5 Contribution of this Study

In this thesis, a cost-based rescheduling methodology using an FBS algorithm in an FMS environment is studied.

The offline and event-driven rescheduling as an appropriate strategy in FMS environments [18,27] is considered in this study. At the time of disruptions (TOD), the existing schedule needs to be updated quickly in order to cope with the disruption. Two rescheduling methodologies are studied in this work: Total rescheduling (TR) and scheduling repair methodologies. Responding to each disruption in FMS by TR causes high operational cost and creates system nervousness. Repairing the schedules for every disruption in every circumstance also causes low efficiency [18]; therefore, finding a point for total rescheduling the system after performing some repairs action could be an effective way in order to minimize the total cost of rescheduling actions and increase the efficiency of the updated schedule.

Even though the existing repair algorithms in the literature can generate solutions with lower deviations at a faster speed than TR, the efficiency of the repair methods is usually lower than TR since the repair methods try to adhere to the original schedule. Hence, developing a repair method that can generate a schedule based on the performance metrics would cover this deficiency. Filtered beam search heuristic algorithm (FBS) is applied in this study in order to repair the disrupted schedules at TOD. This algorithm is also used in generating the initial schedule and the updated schedule by TR. The high quality of solutions and performance, high searching speed, and ability to exploit flexibilities in FMS are some valuable benefits of FBS mentioned in the literature [20,39], which make it a reasonable tool for rescheduling in FMS environments.

Timed based measures and stability are competing objectives: minimizing the makespan in order to satisfy the customers' needs can create the high deviations in the updated schedule and cause high operational costs as a result. On the other hand, any deviation from the original schedule increases system nervousness. Wu *et al.* [33] show that, since efficiency and stability are conflicting objectives, the choice of objective depends on the circumstances. A good way of overcoming the competing effects of the measures is to define a cost based measure that can inherently identify the trade-off between schedule efficiency and stability.

1.6 Objectives and Approach

The purpose of this thesis is:

To show that the cost based rescheduling methodology using FBS is an appropriate way of handling disruptions in FMS environments, which can result in generating the cost efficient updated schedules allowing the trade-off between the time based and stability based criteria.

In order to prove this purpose, first, a rescheduling cost function is proposed in order to assess the negative impact of changes due to rescheduling in FMS. The rescheduling cost function is used as an objective function in order to generate schedules, and can also be employed as a performance measure to compare the impact of various methodologies.

Second, an FBS-based heuristic algorithm is applied in order to totally reschedule the system at the time of disruption and generate the cost efficient updated schedules by the proposed rescheduling cost function.

Third, a schedule repair methodology is developed based on FBS algorithm, in order to generate the cost efficient updated schedules at TOD.

A case study is presented for each case to demonstrate the benefit of using the proposed methodologies. For three types of disruptions, i.e., machine breakdown, new order arrival, and job cancellation, various scenarios are identified and the results are compared with the similar rescheduling methods in the literature in terms of cost based, time based and stability based measures.

1.7 Thesis Outline

This thesis composed of four chapters as follows:

- Chapter one includes introductions, review of relevant literature, contribution of the study, objectives and approach.

- Chapter two discusses about the cost based total rescheduling (TR) approach in FMS. Towards this end, a compound cost function is developed as a measure to be employed as an objective function in FBS in generating the updated schedules. The performance of the cost based schedules is compared with the performance of schedules generated by time based and stability based measures in terms of rescheduling cost, stability and efficiency.

- Chapter three presents the proposed repair methodology in FMS scheduling. For this purpose, a two-phased methodology is suggested. The rescheduling cost function is inserted in a modified version of filtered beam search (MFBSR) in order to generate several repaired schedules. The performance of the updated schedules by this method is compared with mAOR and TR in terms of rescheduling cost, efficiency and stability for various disruption scenarios and different flexibility levels of the FMS environment.

- Chapter four gives the conclusions of the thesis work, and suggests the future directions for the research.

- Appendices include sample TR and MFBSR models in Java Eclipse (www.eclipse.org).

Chapter 2

Cost based rescheduling in FMS using FBS

Changes and unexpected events may occur inevitably during the production process as a result of dynamic and uncertain nature of manufacturing environments. The system needs to be updated and adapted to changes rapidly in a cost efficient way in order to be alive and competitive in today's market. Rescheduling is an essential operational task required to be carried out in order to respond to disruptions and unexpected events in manufacturing systems.

In this Chapter, a methodology for cost based rescheduling is proposed in FMS environment, in order to generate the new schedules having minimum rescheduling cost while obtaining acceptable efficiency and stability levels. A rescheduling cost measure is defined in a form of a compound function to assess the negative impact of changes in the schedule due to disruptions. This function consists of three main rescheduling cost sources; job related, machine related, and material related cost. The method selected for generating the schedules, both the initial and the updated ones, is based on the filtered

beam search heuristic algorithm (HFBS) [39] because of its speed and quality of solutions. Three types of disruptions are considered in this study: machine breakdown, new order arrival, and job cancellation. Finally, In order to validate the performance of the proposed rescheduling methodology, in terms of solution quality, rescheduling cost, efficiency, and stability, various test problems are simulated by different methods in the literature for different disruptions scenarios.

2.1 Problem Statement

A flexible manufacturing system (FMS) with partial flexibility is considered in this work. There are a certain number of jobs to be scheduled each having a different number of operations with alternative machines capable of performing the same operation albeit with different processing times. The initial schedule is generated by a time based objective function (weighted sum of makespan, maximum machine workload, and total workload) [39]. During the execution of the schedule, the following types of disruptions are considered: machine breakdown, job cancellation, and new order arrival. At the time of each disruption (TOD), a new schedule will be generated based on the availability of the machines and remaining available operations while minimizing the rescheduling costs of switching from the existing schedule. Following are the assumptions considered in this study:

- The jobs are non-preemptive.
- An operation cannot be performed on more than one machine at the same time.
- Each machine cannot perform more than one operation at the same time.
- The machines are independent of each other and all are available at $t=0$.

- Machines' set-up times and material handling time are not considered.
- The jobs are independent of each other and can be done at any time separately.
- Processing time is deterministic and fixed during the horizon based on the process plan.

The initial and the updated schedule are generated by using the filtered beam search heuristic method (HFBS) according to an objective function. In the next Sections, the proposed rescheduling cost function is described as a performance metric which is used in generating the updated schedules. The suggested rescheduling methodology is demonstrated with an illustrative example, where the schedules generated by the proposed cost measure are compared with the schedules updated by using time based and stability based measures under different disruption scenarios.

2.2 Rescheduling Cost

A compound cost function is introduced in order to assess the negative impacts of schedule updates while considering both aspects of timing and deviations [68].

Three main rescheduling cost sources can be categorized as machine related, job related, and material related cost:

$$Cost_{Rsch} = Cost(Mach\_R) + Cost(Job\_R) + Cost(Mat\_R) \qquad (2\text{-}1)$$

2.2.1 Machine related Cost

Due to a schedule update, an extra machining cost can result from switching an operation to an alternative machine which has longer processing time. Similarly a change in the sequence or reallocation of jobs due to a disruption can cause an increased idle time over

the updated makespan. Then, the machine related rescheduling cost is the total cost of increased idle time of all machines (C_1) and total cost of extra processing time (C_2) after rescheduling, which are given by the following equations:

$$C_1 = \sum_k Max\left[\left(\underset{ij}{Max}(C_{ijk}^{New}) - \sum_i \sum_j P_{ijk}^{New}\right) - \left(\underset{ij}{Max}(C_{ijk}^{Initial}) - \sum_i \sum_j P_{ijk}^{Initial}\right), 0\right] \times \eta_k \quad (2\text{-}2)$$

$$C_2 = \sum_k Max\left[\left(\sum_j \sum_i P_{ijk}^{New} - \sum_j \sum_i P_{ijk}^{Initial}\right), 0\right] \times \tau \quad (2\text{-}3)$$

Where k is the machine index and the coefficient η_k corresponds to the rate of idle time machine k. P_{ijk} represents the processing time of operation i of job j assigned to machine k. C_{ijk} represents the completion time of operation i of job j on machine k . The coefficient τ represents the penalty cost of extra machining time needed after rescheduling. The first term in Equation (2-2) corresponds to total idle time of machines in the updated schedule and second term corresponds to total idle time of machines in the initial schedule. In Equation (2-3), terms identify total machining time in the updated and the initial schedule respectively.

2.2.2 Job related Cost

Jobs could be shifted as a result of rescheduling and finished earlier or later than the initial schedule. C_3 is defined as the cost of the added lateness of jobs after rescheduling which can be expressed by the following equation:

$$C_3 = \sum_j Max\left[\left(\left(\underset{i}{Max}(C_{ij}^{New}) - D_j\right) - \left(\underset{i}{Max}(C_{ij}^{Initial}) - D_j\right)\right), 0\right] \times \delta_j \quad (2\text{-}4)$$

The coefficient δ_j represents the penalty cost of unit time lateness for each job j and D_j is the job due date. The first term in Equation (2-4), is the tardiness of each job in the updated schedule while the second term corresponds to the tardiness of jobs in initial schedule.

Jobs can also be finished earlier than their completion time in the initial schedule after rescheduling, hence, this creates saving on due dates. This can be considered as benefit (C_4) or negative cost in job related cost function which is expressed by the following equation:

$$C_4 = \sum_j Max\left[\left((Max_i(C_{ij}^{Initial}) - D_j) - (Max_i(C_{ij}^{New}) - D_j)\right), 0\right] \times \theta_j \qquad (2\text{-}5)$$

Where the coefficient θ_j represents saving on due dates of unit time for each job which considered to be negative in the cost function.

2.2.3 Material related Cost

The material related cost consists of three types of cost sources such as:

- Holding cost of WIP and raw material.
- Cost of expediting the material to an earlier time.
- Cost of reallocating the material to another machine.

The starting time of operations is subject to change in the updated schedule. It is assumed that the required raw material for each operation is to be supplied just before the starting time of operation, according to the initial schedule. Thus, changing the starting time or machine assignment of an operation may incur a cost. If the repaired operation

starts later, holding cost (C_5) occurs as a result. Similarly, if the operation has to start earlier than the original schedule, expediting cost (C_6) occurs. Operations may also be assigned to different machines after rescheduling, and this change causes the cost of reallocation (C_7) such as changing toolsets, and extra material handling. The corresponding cost functions are given as follows:

$$\forall k; \quad S_{ijk}^{new} > S_{ijk}^{initial} \qquad C_5 = \sum_i \sum_j \left(S_{ijk}^{new} - S_{ijk}^{initial}\right) \times h_{ij} \tag{2-6}$$

$$\forall k; \quad S_{ijk}^{initial} > S_{ijk}^{new} \qquad C_6 = \sum_i \sum_j \left(S_{ijk}^{initial} - S_{ijk}^{new}\right) \times \mu_{ij} \tag{2-7}$$

$$\forall i, j, k \qquad C_7 = \sum_i \sum_j Y_{ijkk'} \, \omega_{ijkk'} \tag{2-8}$$

The coefficient h_{ij} and μ_{ij} represent the holding and expediting cost of material for the operation i of job j in unit of time. S_{ijk}^{new} is the starting time of the operation i of job j on machine k after schedule update, and $S_{ijk}^{initial}$ is the original starting time of that operation. $\omega_{ijkk'}$ represents the penalty cost of reallocating material between machines after rescheduling process. $Y'_{ijkk'}$ is the binary variable indicating whether the operation i of job j on machine k is switched to machine k' or not.

2.3 Cost based Rescheduling using FBS

It is not always possible to find an optimal solution as an initial or updated schedule quickly in FMS environments, due to NP-hardness of scheduling/rescheduling problems. Therefore, AI-based meta-heuristics approaches are widely utilized recently in order to generate near-optimal solutions in a reasonable time. Through this study, filtered beam search heuristic (FBS) is employed in generating the initial and updated schedules. The proposed cost function, explained in Section 2.2, is used to generate cost efficient updated schedules in FBS. In the next Section, firstly the FBS method is shortly explained, and after that, the complete FBS-based rescheduling algorithm is presented.

2.3.1 Filtered Beam Search

Filtered Beam Search (FBS) is an extension of Beam search (BS) which is the adaptation algorithm of branch and bound (B&B) used in solving optimization problems. This algorithm uses heuristics to estimate certain number of the best paths and eliminate permanently the rest. FBS works much faster than B&B as the large parts of search tree are pruned accumulatively. The BS-based algorithms are like breadth-first algorithms as they progress level by level without backtracking [27]. However unlike the breadth-first search, BS doesn't search through all possible nodes and only moves down from the best promising nodes at each level. An evaluation function used to identify the promising nodes in each level, which introduces the problem of finding proper trade-off between quick but poor, and computationally demanding but better solutions [70]. Filtered beam search is introduced in order to find a good tradeoff between speed and accuracy [51]. By two phase evaluations which are called as local and global evaluation, filtering phase and

beam selection (known as rough and accurate), nodes are pruned in each level and the best node is identified. Two key parameters in FBS algorithm are called as *filterwidth* and *beamwidth* which identify the number of the filtered nodes (*f*), and the number of final solutions (*b*) respectively. *b* numbers of nodes are selected by a global evaluation procedure at the first level. For each of the following level, *f* numbers of nodes are filtered by a local evaluation procedure firstly, and in next step, by performing a global evaluation procedure on the remaining nodes (*f*), the best promising node for each *b* node is selected. The selected nodes in each level added to the partial schedules, and the *b* numbers of schedule will be generated at end.

As shown in Figure 2.1, after determining the beam nodes in the first level by a global evaluation, the filtered beam search is employed independently to generate a partial schedule from each of them (in this example, as *b* is set to 2, two independent tree is generated would result in two schedules). Once the best node in each level for each beam is identified, nodes are generated for the next level by using the branching method. The generated nodes first locally evaluated and *f* numbers of nodes remain for the global evaluation. The procedure continues until all the machine-job pairs are allocated and *b* numbers of schedules are generated.

2.3.2 FBS based Methodology

The procedure of generating schedules by FBS method consists of two phases: Generating the search space called branching methodology, and evaluating the nodes by using the global and local evaluation functions called bounding methodology. In order to generate a search tree in FBS, two procedures called Active and Nondelay for job shop scheduling are discussed [71]. In this study the modified form of Nondelay called

M_NONDELAY [39] is used as a branching method. After a level is formed by M_NONDELAY algorithm, it is ready to be bounded and is followed by a search method. The key point in utilizing the FBS is choosing the proper evaluation function. Searching process among the available nodes is performed by the evaluation functions, and the partial schedules are generated at each level of the search tree. In this study, the Modified Shortest Processing Time (M_SPT) [39] is employed as a dispatching rule in local and global evaluation procedure.

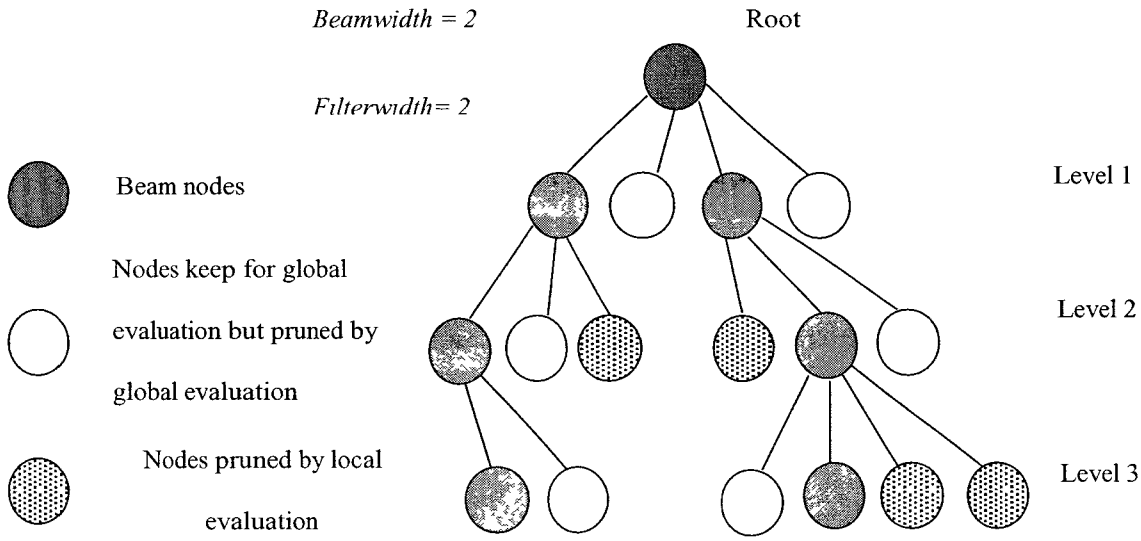


Figure 2.1: Filtered beam search tree representation [39].

Finding an appropriate value for *filterwidth* and *beamwidth* is a tradeoff between accuracy and speed through the algorithm. The number of *beamwidth* determines the number of schedule to be generated for each problem, whereas the *filterwidth* determines the number of operation-machine allocation choices at each level. Their values are problem specific and can be determined by analyzing the tradeoff between the

performance level of the generated schedules and their computational times. In this study, for each specific problem, a range of values for f and b are tested. The smaller pair of f and b which results in the best objective values is selected as *filterwidth* and *beamwidth* to be used in FBS procedure.

2.3.3 The Proposed Rescheduling Algorithm

In case of a disruption at TOD, the schedule can be updated by rescheduling all the remaining operations in the system. In order to generate the updated schedule, the remaining operations and availability of each machine need to be determined. We define the remaining operations the ones that have not begun by TOD. This implies that the operations which are in-progress at TOD should be first completed in order to identify the earliest availability of each machine. The following steps define how the total rescheduling method (TR) is performed in case of each type of disruption:

Step 1: ***Initialization***

Input the initial schedule generated by FBS (operations, assignment, starting time), input the value of *filterwidth (f)* and *beamwidth (b).*

Step 2: ***Disruption Occurrence - (Re-Initialization)***

At TOD, the system needs to be rescheduled in order to respond to disruptions. Depending on the type of disruptions, different scenarios may stand out;

Case 1: Machine Breakdown

In this study, a predetermined repair time is considered for each machine. It is also assumed that the failed machines would be available after the repair time.

In case of a machine breakdown at TOD, the operations which are in-progress on the un-failed machines should be completed first to identify the machines' availabilities. If the failed machine has an operation in-progress, the remaining part of that operation should be served by that machine after the repair time. Starting time of operations and machine availabilities are updated and the system becomes ready to be scheduled for the remaining operations.

Case 2: Job Cancellation

When a job is cancelled at TOD, all its remaining operations should be cancelled even if one is in-progress. Then, the initial partial schedule is formed by all the operations that have been finished by TOD, and are in-progress in TOD, and not-cancelled. The starting times and machine availabilities are updated and the partial schedule is ready to be completed by allocating the remaining operations.

Case 3: New Order arrival

If a new order arrives to the system at TOD, the in-progress operations should be completed first. The new order is scheduled as a regular order among the remaining operations. Hence, the updated list of the remaining operations, start times and machine availabilities will be used to generate a new schedule by TR procedure.

Step 3: ***Node generation***

(i) M_NONDELAY scheme is used to generate nodes from the updated existing partial schedule resulted from Step 2. Check the total number of nodes generated, N; update the level, and update the partial schedule PS_l by generating nodes.

(ii) If N< b, then go down to the next level, generate new nodes by M_NONDELAY and PS_l, update the level and PS_l by generating nodes. If N< b, then go to Step 3. (ii); else go to Step3.(iii).

(iii) Find the global evaluation function values for all the nodes and select the best b numbers of nodes (defining the initial beam nodes). Determine the candidate sets of each beam $PS_l(1)$, $PS_l(2)$,..., $PS_l(b)$

Step 4: ***Determining the beam nodes***

Check the level and the number of remaining operations. If any operation remains, go to step 4.(i); else go to Step 5.

(i) Generate N new nodes from each beam node according to M_NONDELAY with PS_l as the partial schedule represented by the beam node. If N<b go to Step 4.(i); else go to Step 4.(ii).

(ii) *Filtering process*- Choose the best f number of nodes according to the local evaluation function procedure.

(iii) *Global evaluation process*- Computing the objective function values for each filtered nodes.

(iv) *Beam nodes selection*- Select the nodes with the best objective function, add it to the partial schedule, and update the partial schedule and level.

Step 5: ***Select the solution schedule***

Among the b numbers of generated schedules, select the schedule set or schedule sets with the best objective function values.

2.3.4 The Complexity Analysis

The time complexity of the proposed algorithm in worst case is $O(n^3)$, where n is the total number of operations to be scheduled in the FMS problem. In worst case, the algorithm in step 3 generates at most n possible nodes, and then b numbers of best nodes are selected from n nodes by a global evaluation function (bn). For finding the value of the global evaluation function, the algorithm should generate a complete search tree of depth n since there are n operations. Then the total complexity in step 3 is $O(bn^2+n)$. In step 4, two loop exists, the outer loop which needs computational time roughly to bn as it forms b complete beams, each has depth n, and the inner loop which are firstly selected f nodes by a local evaluation among the remaining ones (at most n) and expand further to n level, which makes the total computational time roughly to fn^2.Thus, the total complexity of step 4 would be $bn \times fn^2$. So the time complexity of step 4 is $O(bfn^3)$. Therefore the overall complexity of the proposed algorithm is $O(Max\ (bn^2+n,\ bfn3))$. As b and f are small in comparison to n, the total complexity of the algorithm is $O(n^3)$.

2.4 Illustrative Examples

In order to demonstrate the efficiency and performance of the proposed rescheduling algorithm in FMS environments, a numerical study is developed, tested and evaluated. The algorithm is run on a personal computer with an Intel Core 2 Duo CPU, 2 GB RAM on Microsoft Windows XP Professional. The codes are written in the Java, Eclipse (Galileo 3.5.1 platform). The corresponding Java codes are presented in Appendix A.

The initial test problem considered in this study is a small FMS system with 4 partially flexible machines, with 4 jobs each having 3 operations. Table 2.1 represents the

processing times of operations on each alternative machine. The processing times are represented as random numbers uniformly distributed between 2 and 5.

Table 2.1: Process time table

Job	Operation	Process time			
		Machine 1	Machine 2	Machine 3	Machine 4
	1			3	4
1	2	2	5		5
	3		3	2	5
	1	4			
2	2			3	4
	3		2	2	3
	1	3	5	2	4
3	2	5	5		
	3	2		5	
	1	4		3	
4	2	3	5		2
	3	2			3

2.4.1 Initial Schedule Generation

The initial schedule is generated by the traditional FBS method [39]. M_SPT dispatching rule [39] is used as local and global evaluation functions and the objective function is the weighted sum of makespan, maximum machine workload, and total processing time.

Finding the proper value of *filterwidth* and *beamwidth* is a crucial task in FBS in order to balance the computational time and solution quality. In this case study the appropriate values of these parameters are obtained via experimental trials. b and f are set to be between 2 and 8. The total value of objective function (F-value), the weighted sum of makespan, machine workload and total processing time, is represented by changing the f and b value in Table 2.2. According to the table, setting f=3 and b=4 could be the best choice which could results in the lowest value of F=18.

Table 2.2: F value for different f and b

	f=1	f=2	f=3	f=4	f=5	f=6	f=7	f=8
b=2	20.8	20.4	20.1	20.1	20.1	20.1	20.1	20.1
b=3	20.8	20.4	20.1	20.1	20.1	20.1	20.1	20.1
b=4	18.3	18.1	**18**	18	18	18	18	18
b=5	18.3	18.1	18	18	18	18	18	18
b=6	18.3	18.1	18	18	18	18	18	18
b=7	18.3	18	18	18	18	18	18	18
b=8	18.3	18	18	18	18	18	18	18

As a result, the initial schedule generated by this method and the mentioned parameters is obtained as depicted in Figure 2.2.

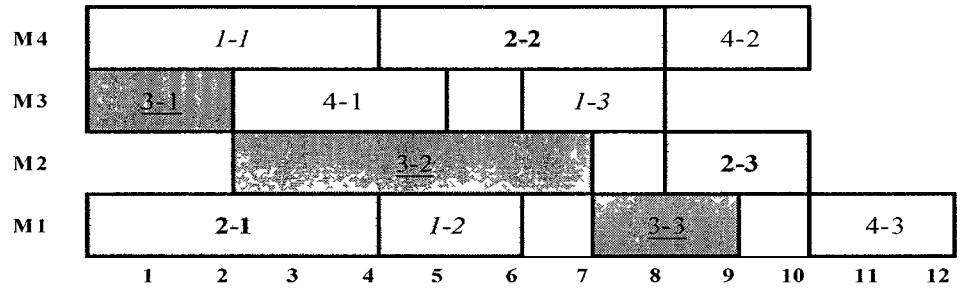


Figure 2.2: The static schedule result for the FMS problem with b=4, f=3.

The generated schedule which is shown in Figure 2.2 has a makespan of 12 as *F1*, maximum machine load of 10 as *F2* and total processing time of 34 as *F3*. The weights of these functions are considered 0.4, 0.3 and 0.3 respectively. Thus, the F value would can be calculated by F= 0.4 ×12 + 0.3×10 + 0.3×34= 18.

2.4.2 Rescheduling Examples

Three types of disruptions i.e., machine breakdown, new order arrival and job cancellation, are considered in this study. The cost based rescheduling (TR) approach is performed at TOD, and the new schedule with the least rescheduling cost is generated

and replaced with the initial one. The cost coefficients shown in Table 2.3 are used to calculate the corresponding rescheduling cost for each updated schedule.

Table 2.3: Cost coefficients

Cost function coefficient		value
Coefficient of Lateness	δ_j	12
Coefficient of Due date saving	θ_j	-12
Coefficient of Expediting	μ_{ij}	10
Coefficient of Holding	h_{ij}	1
Coefficient of Reallocation	$\omega_{ijkk'}$	4
Coefficient of Extra idle time	η_k	6
Coefficient of Extra processing time	τ	6

2.4.2.1 Machine Breakdowns

The machines which are failed to perform their operations are selected randomly through the program. In the Figure 2.3, machine 4 is failed at time t=4 for ΔT=3 as a repair time. Rescheduling cost of this schedule is derived by the following function;

Rescheduling cost= cost of extra lateness +cost of expediting + cost of holding +cost of reallocation +cost of extra machine idle time+ cost of extra machining- saving on due dates = 12×2 + 10×2 +1×2 + 4×3 +6×0 +6×0- 12×1= 46 .

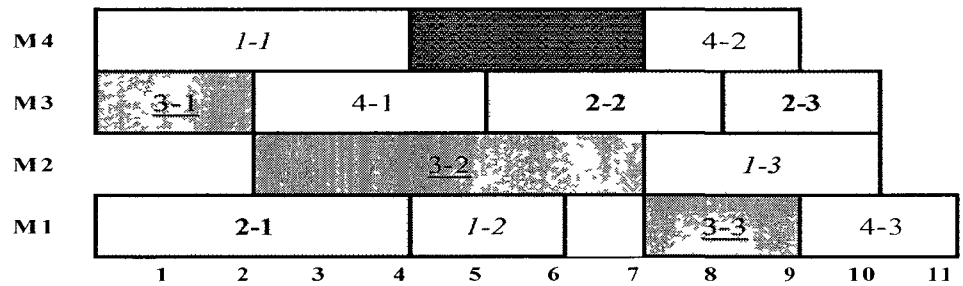


Figure 2.3: Gantt chart obtained after rescheduling, in case of machine 4 failure at t=4.

2.4.2.2 Job Cancellation

The job to be cancelled is selected randomly. In Figure 2.4, job 1 is cancelled at the time t=5. The rescheduling cost of this schedule can be calculated by observing the differences of the initial and the updated schedules represented in Figure 2.4 and Figure 2.2 as follows;

Rescheduling cost= cost of extra lateness +cost of expediting + cost of holding +cost of reallocation +cost of extra machine idle time+ cost of extra machining- saving on due dates = $12\times0 + 10\times0 + 1\times0 + 4\times0 + 6\times3 + 5\times0 - 12\times0 = 18$.

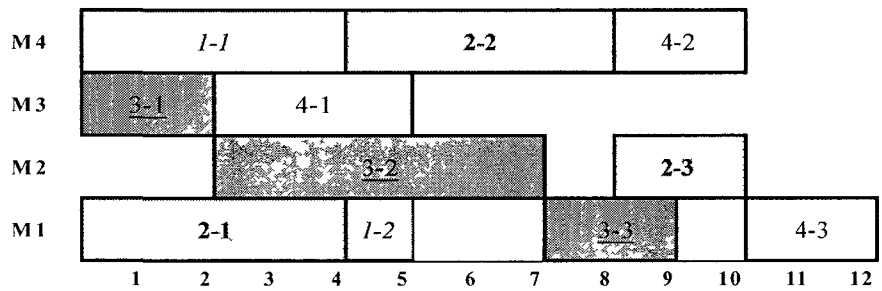


Figure 2.4: Gantt chart obtained after rescheduling, in case of job 3 failure at t=6.

2.4.2.3 New Order Arrival

The new order is selected randomly among the existing jobs to enter to the system. It is assumed that the new order is treated as a remaining job in the system and needs to be scheduled as a regular job. In Figure 2.5 the job 2 is arrived to the system at t=5. The rescheduling cost of the updated schedules in Figure 2.5 can be calculated as follows;

Rescheduling cost= cost of extra lateness +cost of expediting + cost of holding +cost of reallocation +cost of extra machine idle time+ cost of extra machining= $12\times14 + 10\times0 + 1\times3 + 4\times1 + 6\times2 + 6\times10 - 12\times0 = 247$.

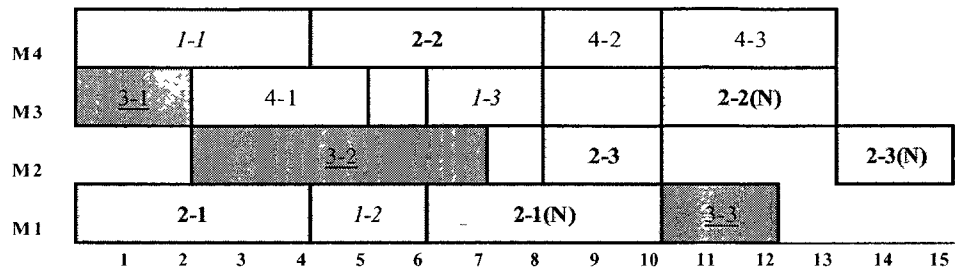


Figure 2.5 : Gantt chart obtained after rescheduling, in case of a job 2 arrives at t=5.

2.5 Experimental Design

A factorial experiment is performed in order to evaluate performance of the proposed cost based rescheduling algorithm, with time based and stability based performance measures. The effects of various disruption scenarios are studied on different rescheduling methods.

2.5.1 Experimental Factors

Three experimental factors are considered: objective function, magnitude of disruption, and timing of disruptions.

- Objective Function (OBJ): the effects of different objective functions are studied. These methods can be categorized in time based methods i.e., weighted sum of makespan, machine load and total processing time [39], Equation(2-9); weighted quadratic function of tardiness [20] , Equation(2-10); and stability based methods [14], Equation(2-11); which generate the schedules based on the corresponding objective functions.

$$OBJ(Mks) = w_m F_1 + w_{tw} F_2 + w_{mw} F_3 \tag{2-9}$$

$$OBJ(Trd) = \sum_i w_i T_i^2 \tag{2-10}$$

$$OBJ(Stb) = \sum_j \sum_i \left| St_{ij} - St_{ij}' \right| \tag{2-11}$$

- Magnitude of the disruption (MAG): The magnitude of disruptions is considered as the number of disruptions that occurs at the time of disruption.
- Time of the disruption (TIM): This factor refers to the time of occurrence of the disruptions relative to the makespan. This timing can be early or late during the horizon.

2.5.2 Dimensions of Experiments

OBJ has four levels reflecting the proposed cost based methodology (*C_B*), *W_Mks* represents the weighted sum function makespan, *Q_Trd* corresponds to the weighted quadratic function of tardiness of jobs, and *Stb* which represents the deviation from the initial schedule. Two levels of treatment, low and high, are considered for magnitude and timing of disruptions. As a result, four combinations of disruption scenarios are represented in Table 2.4 for each type of disruption and method. The designed experiments shown in Table 2.5 are performed in ten replications.

The values of the initial schedule parameters are shown in Table 2.6. The selected coefficients of the proposed rescheduling cost functions are given in Table 2.3.

Table 2.4: Disruption scenario

Expt. No.	MAG	TIM
1	High	Early
2	High	Late
3	Low	Early
4	Low	Late

Table 2.5: Experiments levels

Experimental factor		Level 1 (Low)	Level 2 (High)
Time of disruption (TIM)		60-90% of makespan Late	5-40% of makespan Early
Magnitude of disruption (MAG)			
	Machine breakdown	1 machine breakdown	3 machine breakdown
		Repair Time: Uniformly Distributed [5,10]	
	New order arrival	1 order arrival	3 order arrival
	Order cancellation	1 job cancellation	3 job cancellation

Table 2.6: Initial schedule parameters

Parameter	Values
Number of jobs	20
Number of operations per each job	5
Number of machines	7
Process time of operation	Uniformly distributed [3,8]

2.5.3 Performance Measures

Three performance metrics are considered in this study: the efficiency and stability of the updated schedules, and the proposed rescheduling cost which are explained as follows;

2.5.3.1 Efficiency

Efficiency of the updated schedule is defined as percentage of changes in its makespan compared to the initial schedule makespan, which is as follows:

$$\mathrm{EFF} = \left\{1 - \frac{Mks_{New} - Mks_{Initial}}{Mks_{Initial}}\right\} \times 100\% \ , \tag{2-12}$$

Where Mks_{New} and $Mks_{Initial}$ are the makespan of the updated and initial schedule respectively. Since in some cases especially in job cancellation, the makespan of the updated schedule becomes lower than the makespan of the initial schedule, the efficiency metric might get a value higher than 100%.

2.5.3.2 Stability

Stability is defined as a measure of deviation in starting time of operations in the updated schedules compared to the initial schedule. In this study, the normalized sum of these deviations is applied as a performance metric [11].

$$\mathrm{DevSt} = \frac{\sum_{i=1}^{k}\sum_{j=1}^{p}\left|(St'_{ij} - St_{ij})\right|}{k \times p} \ , \tag{2-13}$$

Where $DevSt$ is the normalized deviation, St'_{ij} and St_{ij} are the starting times of operations in the updated and initial schedules respectively. k is the total number of jobs and p is the total number of operations. The lower value of deviation leads to the more stable schedule.

2.5.3.3 Cost of Rescheduling

The rescheduling cost corresponding of each OBJ is calculated and compared to others in order to assess the negative impact of changes due to rescheduling. In order to make better comparison among rescheduling costs, the rescheduling costs of *C_B* is considered as a base level. Then, the corresponding rescheduling cost resulting from *W_Mks*, *Q_Trd*, and *Stb* are expressed in percentage relative to this base level.

2.6 Experimental Results

The results of the designed experiment in Table 2.4 are discussed through this section. The average changes in performance measures resulting by different disruption scenarios are presented for each OBJ.

2.6.1 Machine Breakdown

The results of machine breakdown show that the rescheduling cost of the updated schedules generated by *C_B* is dominant to the ones generated by other OBJs as shown in Figure 2.6(a). The differences are significant for early disruptions (Table 2.7). Figure 2.6(b) shows that the efficiency of the updated schedules generated by the time based measures, *W_Mks* and *Q_Trd* is slightly higher than *C_B* for early disruptions (Table 2.8), while for the late disruption scenarios they all perform quite similar. The time based measures generate schedule in order to minimize the makespan and lateness criteria, regardless of the initial schedule structure and corresponding changes due to rescheduling, hence, the schedules generated by these measures have higher cost of rescheduling than *C_B*. Stability based measure generates the schedule based on minimizing the changes in starting time from the initial schedule, as a result the updated

schedules have less deviation from the initial schedule, but it causes degradation in efficiency and rescheduling cost (Table 2.9).

Table 2.7 : Rescheduling cost of disruption scenarios

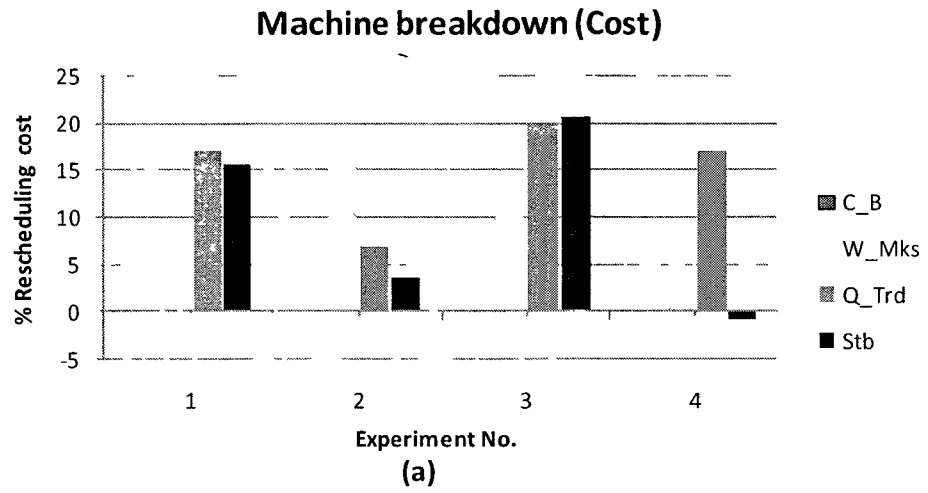
	Disruption											
	Machine breakdown				Job cancellation				Order arrival			
Experiment	*C_B*	*W_Mks*	*Q_Trd*	*Stb*	*C_B*	*W_Mks*	*Q_Trd*	*Stb*	*C_B*	*W_Mks*	*Q_Trd*	*Stb*
1	3702 3	4033 1	4331	4276 3	1268 5	2724	2598 7	2486 2	5568 9	5521 1	5958 9	5864
2	693 9	818 1	741 5	719 6	100 1	190 4	196 7	232 1	1521 5	1518 6	1509	2167 5
3	3341 2	3756 6	4009 6	4029 7	1377 8	3161 5	3167 5	3055 2	2965 5	3907 3	4227 6	4162
4	355	361 1	415 6	352 3	112 3	197 8	266 5	207 8	740 6	770 1	737 2	1039 9

Table 2.8 : Efficiency of disruption scenarios

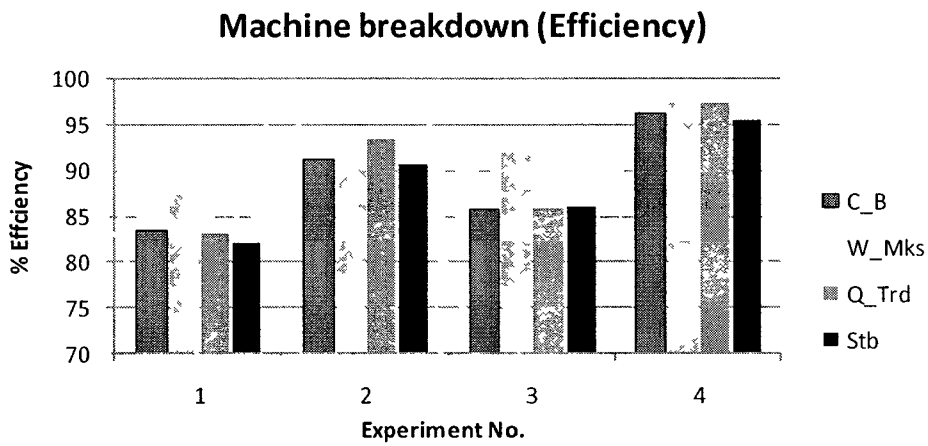
	Disruption											
	Machine breakdown				Job cancellation				Order arrival			
Experiment	*C_B*	*W_Mks*	*Q_Trd*	*Stb*	*C_B*	*W_Mks*	*Q_Trd*	*Stb*	*C_B*	*W_Mks*	*Q_Trd*	*Stb*
1	83 36	87 35	83 19	82 03	103 33	107 15	104 33	98	79 37	81 03	75 04	68 05
2	91 18	92 01	93 34	90 68	103 66	104 33	105 49	99 83	77 54	78 37	77 37	47 09
3	85 69	91 85	85 86	86 02	99	101	98 84	95 17	90 85	92 35	87 35	83 53
4	96 17	97 34	97 5	95 51	100 5	101 33	101 16	97 67	89 35	87 52	85 86	73 54

Table 2.9 : Normalized deviations of disruption scenarios

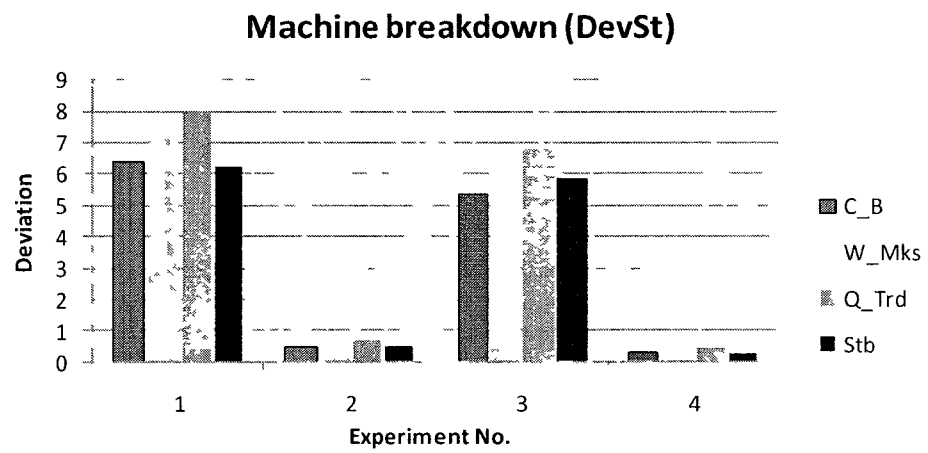
	Disruption											
	Machine breakdown				Job cancellation				Order arrival			
Experiment	*C_B*	*W_Mks*	*Q_Trd*	*Stb*	*C_B*	*W_Mks*	*Q_Trd*	*Stb*	*C_B*	*W_Mks*	*Q_Trd*	*Stb*
1	6 36	7 32	8 03	6 23	1 53	4 35	4 41	3 04	7 85	7 91	8 72	6 91
2	0 5	0 73	0 7	0 51	0 17	0 34	0 37	0 11	0 61	0 65	0 54	0 33
3	5 37	6 37	6 77	5 85	1 98	5 32	5 39	4 34	4 27	6 13	6 51	5 16
4	0 3	0 37	0 47	0 25	0 19	0 3	0 47	0 18	0 21	0 42	0 34	0 15



(a)



(b)



(c)

Figure 2.6: Machine breakdown: (a) rescheduling cost; (b) efficiency; (c) deviation.

2.6.2 Job Cancellation

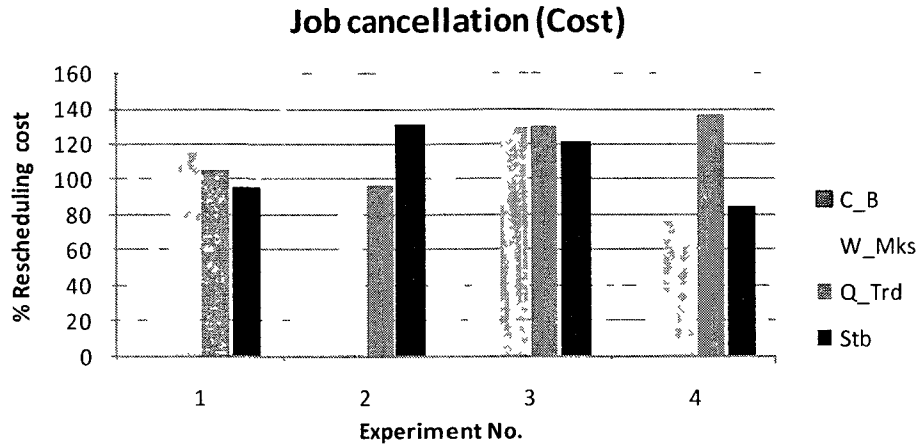
The results of job cancellation in Figure 2.7 reveal that *C_B* significantly outperforms other OBJs in terms of cost of rescheduling. Table 2.7 also reports the huge differences in the rescheduling cost between *C_B* and other OBJs, which are higher for early disruption scenarios. According to Figure 2.7(b), the efficiency of the updated schedules by the time based measures is slightly better than *C_B* for early disruptions while for the late ones, the three OBJs, *C_B*, *W_Mks*, and *Q_Trd* perform quite similar. It is shown in Figure 2.7(c) that the updated schedules by *C_B* have lower deviations and as a result better stability than others, especially for early disruption scenarios. The high amount of deviation in the time based measures results in the high value of rescheduling cost.

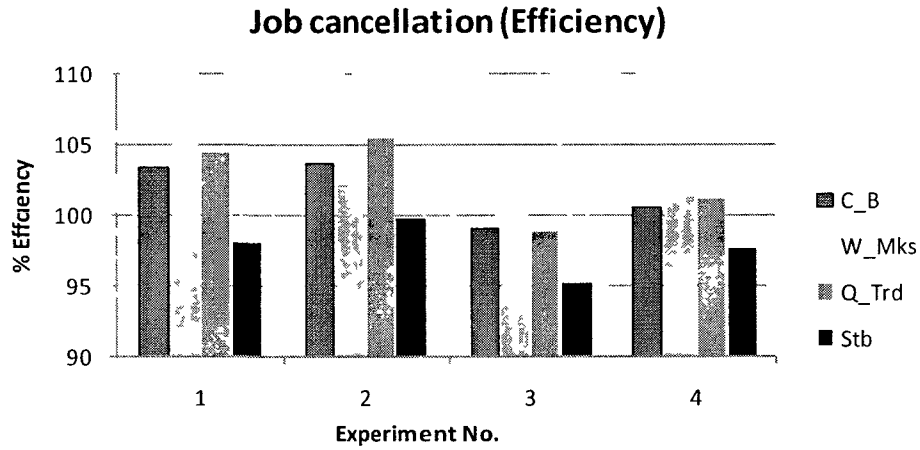
2.6.3 New Order Arrival

The results of new order arrival are represented in Figure 2.8. It is shown in Table 2.7 that C_B performs better than other OBJs in terms of rescheduling cost especially in early disruptions. In late disruptions time based and cost based measures perform quite similar.

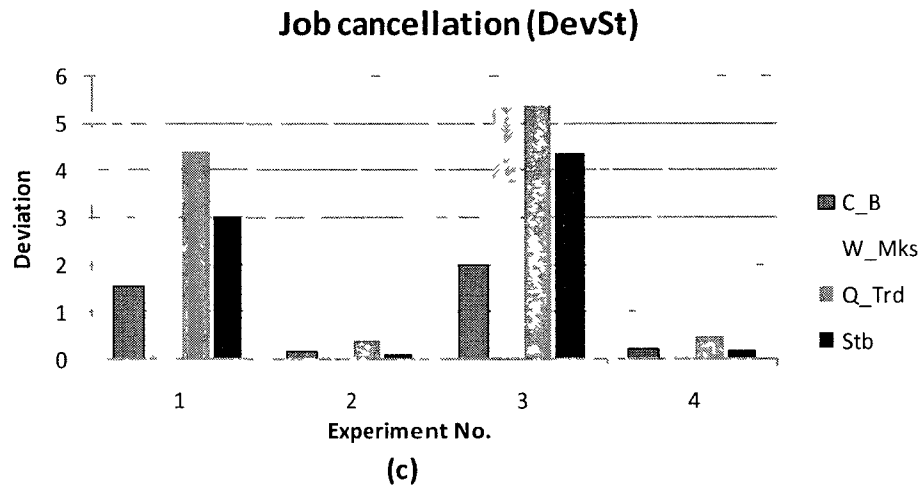
In case of late disruptions, as fewer numbers of operations remained to be scheduled, adding new orders might not result in high deviation between the updated schedules generated by time based and *C_B,* while the schedules generated based on *Stb* has much higher cost and less efficiency. This is because *Stb* tends to keep the schedule the same in order to minimize the changes, hence by adding the new orders towards the end of schedules, it minimizes the deviation, but the makespan and rescheduling cost are degraded.

In case of early and high magnitude disruptions, Experiment 1, as the number of operations added to the schedules is high; all OBJs generate the updated schedules in a high cost and with similar levels (Table 2.7) while still *C_B* performs slightly better as it generates higher efficiency and lower deviations than others. For the low magnitude disruptions, Experiment 2, *C_B* shows its advantage and generates schedule with significantly lower rescheduling cost while having higher efficiency and lower deviations than other OBJs.





(b)



(c)

Figure 2.7: Job cancellation: (a) rescheduling cost; (b) efficiency; (c) deviation.

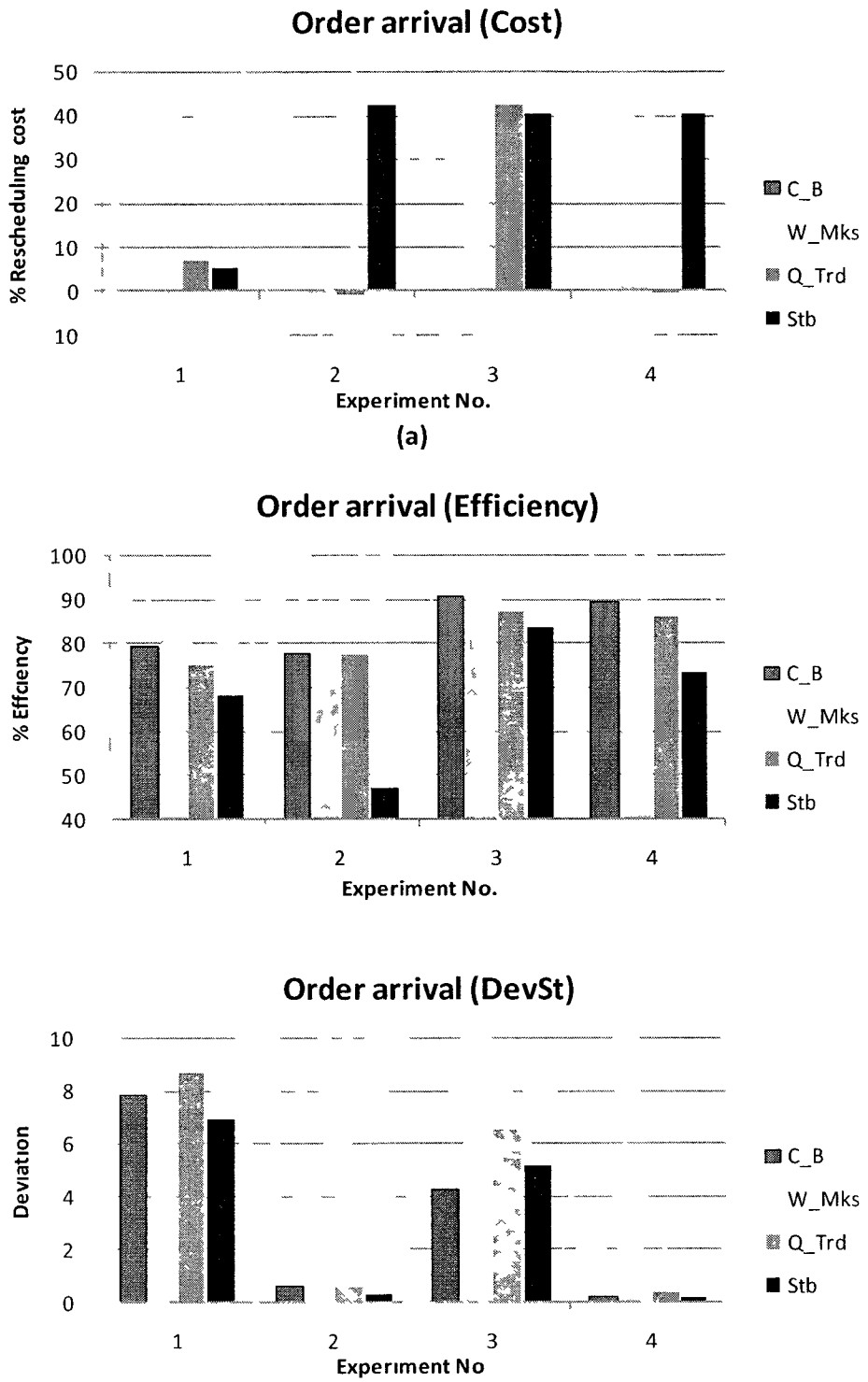


Figure 2 8 Order arrival (a) rescheduling cost, (b) efficiency, (c) deviation

2.7 Sensitivity Analysis

The coefficient values used in the cost function can change depending on the companies' priorities. For example, in this study as shown in Table 2.3, in order to satisfy the customer demands as a main objective, lateness penalty is considered to be the highest value among all the coefficients. In order to investigate the effect of each coefficient on the optimal solution, it is necessary to perform the sensitivity analysis [72]. The main purpose of sensitivity analysis in this study is to identify sensitive parameters, that changing in their values could result in significant changes in the optimal solution.

Through this study, five levels are considered for each coefficient: 0.1, 1, 10, 50, and 100. The remaining parameters are fixed at 1. The initial schedule is generated using data set defined in Table 2.6. The makespan (Mks), deviation (Dev) from the initial schedule, the unit of changes in decision variable (U/C), and the cost of rescheduling are calculated for all the levels of each disruption type. For each change in the cost coefficient, the total rescheduling cost is normalized by the corresponding decision variable value of the cost component (Cost/UC).

2.7.1 Machine Breakdown

The results of sensitivity analysis for cost function coefficients (Section 2.2) in case of machine breakdown are shown in Table 2.8.

By increasing the cost of expediting the material, μ, operations tend to start later than their initial schedule position, hence, the Mks increases and the time unit number of expedited operations (U/C) decreases. Figure 2.9 reveals that that expediting cost parameter is robust between 0.1 and 1, and then becomes dominant for greater values.

The due date saving parameter, θ, could also be considered as a sensitive parameter as the changing in its value results in the following changes in the optimal solution criteria: By increasing this value, jobs tend to finish earlier than previous schedule. Since all jobs cannot be finished earlier in case of machine breakdown, some of them may be finished earlier and exploit the saving and the rest may be postponed which causes lateness, and as a result the makespan of schedule increases. Consequently, the number of time unit of saving on due dates for jobs (U/C) also decreases. As the coefficient value increases, the total cost of rescheduling decreases significantly due to negative impact of θ. Figure 2.9 shows that the changes in Cost/UC values due to the changes in θ are more significant than the corresponding changes for ω, δ, and h.

Figure 2.9 reveals that, the proposed rescheduling algorithm performs similar reactions in changing the values of coefficients of extra lateness, δ, reallocating, ω, and holding, h. These parameters are robust between 0.1 and 10 and have lower normalized costs (cost/UC) compared to μ and θ.

Table 2.8 shows that, by changing the value of extra idle time coefficient, η, in case of machine breakdown, the algorithm may not create good enough results, as the normalized cost highly increases by increasing the η. The corresponding MKs firstly decreases between 0.1 and 50, but it increases significantly between 50 and 100.

Changing in the the coefficient of extra processing time, τ, may not have any significant effect in the optimal solution in case of machine breakdown. As shown in Table 2.8, the corresponding variable of extra processing time has either the value of 1 or 0, and the changes in the total cost are not consistent with the changes in the parameter value.

Table 2.10: Sensitivity analysis of coefficient for machine breakdown

Machine breakdown													
		Mks	Dev	U/C	Cost	Cost / UC			Mks	Dev	U/C	Cost	Cost / UC
δ	0.1	61	324	54	362.4	6.71	w	0.1	60	335	30	369	12.3
	1	60	335	53	396	7.47		1	60	335	30	396	13.2
	10	61	325	59	949	16.08		10	61	375	21	607	28.9
	50	65	398	43	3460	80 47		50	63	312	17	1246	73.29
	100	63	432	47	5175	110.11		100	65	552	16	2232	139.5
m	0.1	58	323	128	260 5	2 04	η	0 1	62	363	18	408 8	22.71
	1	60	335	123	396	3.22		1	60	335	11	396	36
	10	67	602	20	1103	55.15		10	60	371	1	471	471
	50	76	740	14	1845	131.79		50	59	403	1	536	536
	100	74	852	11	2383	216.64		100	67	779	1	1113	1113
h	0.1	63	379	323	236	0 73	t	0.1	60	335	1	395 1	395.1
	1	60	335	212	396	1 87		1	60	335	1	396	396
	10	60	317	210	2282	10 87		10	60	288	1	358	358
	50	59	344	224	11380	50.8		50	58	330	0	377	∞
	100	59	344	224	22580	100.8		100	58	292	0	321	∞
θ	0 1	62	320	3	436	145.33							
	1	64	335	4	396	99							
	10	65	886	11	274	24 91							
	50	66	2517	16	-436	-27.25							
	100	66	4621	21	-1374	-65.43							

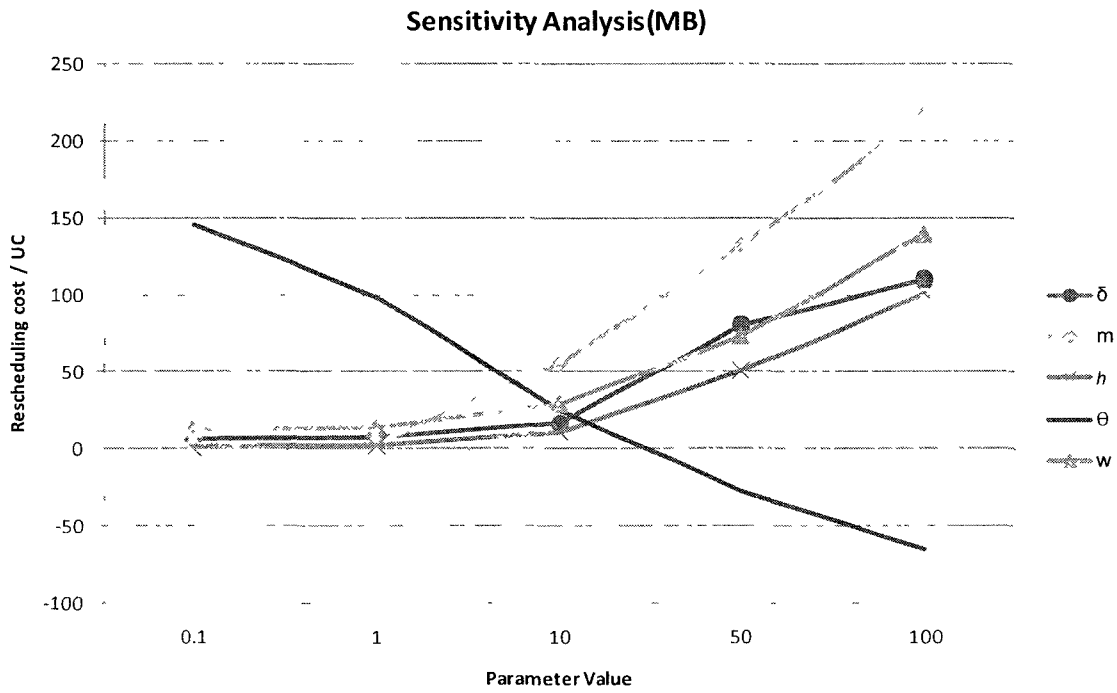


Figure 2.9: Cost of rescheduling changes due to changing the coefficients in case of machine breakdown.

2.7.2 Job Cancellation

The results of sensitivity analysis for job cancellation are represented in Table 2.9 and Figure 2.10. It is shown that μ, as the coefficient of expediting and θ, the coefficient of saving on due dates can be considered as sensitive parameters in case of job cancellation.

Increasing the cost of expediting the material causes degradation of makespan in the updated schedules according to Equation 2.7, as shown in Table 2.9. The results also reveal that the rescheduling cost corresponding to changes in value of μ is consistent with the changes in U/C and Mks. Figure 2.10 shows the significant impact of changing μ in Cost/UC compared to the other parameters.

The makespan of the updated schedules increases by increasing the saving on due dates of jobs. Table 2.9 reveals that the time unit number of saving on job's due dates increases as a result of increasing the θ. Consequently, the rescheduling cost resulting from these changes is consistent with the corresponding changes in U/C and Mks. Figure 2.10 shows that the impact of θ compared with other parameters such as ω, δ, and h is higher and can be considered as an important parameter in case of job cancellation.

The Figure 2.10 shows that the three coefficients of extra lateness, δ, holding, h, and reallocating, ω have quite similar trend in terms of Cost/UC. This reflects the similar performance of the algorithm in responding to these changes. These three parameters are quite robust between 0.1 and 10, and cause lower level in Cost/UC compared to μ and θ.

The coefficient of extra idle time, η, may not have a significant impact in optimal solution as changing in its value results in different trends in the corresponding variable and the total cost, as shown in Table 2.9. In case of job cancellation, as there is no extra

processing time due to rescheduling process, hence, the coefficient of extra processing time, τ, have not any effect in the optimal solution.

Table 2.11 : Sensitivity analysis of coefficients for job cancellation

Job cancellation													
		Mks	Dev	U/C	Cost	Cost / UC			Mks	Dev	U/C	Cost	Cost / UC
	0.1	59	144	10	163	16.3		0.1	57	170	30	178	5.93
	1	58	158	8	178	22.25		1	58	158	31	178	5.74
δ	10	59	161	9	279	31	w	10	58	156	24	410	17.08
	50	59	183	8	594	74.25		50	63	236	17	1183	69.59
	100	59	183	8	994	124.25		100	67	368	15	2023	134.87
	0.1	56	178	113	92 3	0 82		0 1	57	151	7	157 7	22 53
	1	58	158	73	178	2.44		1	58	158	10	178	17.8
m	10	61	219	10	412	41.2	η	10	60	250	1	355	355
	50	62	297	3	575	191.67		50	61	229	5	594	118 8
	100	64	411	0	604	604		100	61	229	5	844	168 8
	0 1	59	152	99	83 8	0.85		0.1	58	158	0	178	∞
	1	58	158	85	178	2.09		1	58	158	0	178	∞
h	10	57	147	66	743	11.26	t	10	58	158	0	178	∞
	50	57	147	66	3383	51.26		50	58	158	0	178	∞
	100	57	147	66	6683	101 26		100	58	158	0	178	∞
	0.1	60	331	4	393 6	98 4							
	1	59	335	5	396	79 2							
θ	10	60	415	23	66	2.87							
	50	67	556	63	-2633	-41 79							
	100	67	617	63	-5670	-90							

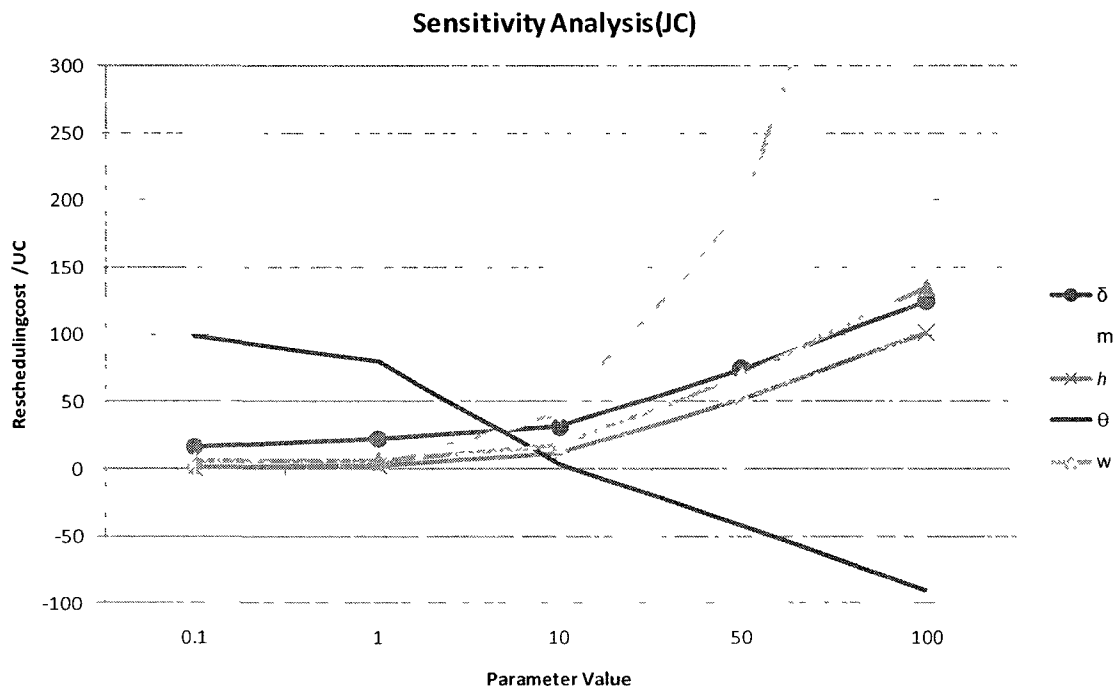


Figure 2.10: Cost of rescheduling changes due to changing the coefficients in case of job cancellation.

2.7.3 Order Arrival

The results of sensitivity analysis for the new order arrival are represented in Table 2.10 and Figure 2.11. The results reveal that, μ, as the coefficient of expediting, θ, as the coefficient of saving on due dates, and η, as the coefficients of extra idle time could be considered as sensitive parameters in case of a new order arrival.

High impact of changes in Cost/UC due to changes in the values of θ is observed from Figure 2.11. These results reveal that this parameter could be highly sensitive to changes in the objective function. The MKs as a result of increasing the θ increases and the corresponding cost decreases as a result.

Increasing the value of μ results in degrading the makespan and decreasing the U/C. the changes in Mks, Dev, and cost are more significant for higher values of μ as shown in Table 2.10. Figure 2.11 also represents that increasing the μ results in significant increase in the Cost/UC value.

Figure 2.11 shows that the results of Cost/UC may change significantly by changing the η. By increasing the cost of extra idle time, machines tend to be loaded for longer time during the horizon; accordingly operations may change their positions to another machine with longer process time, hence the makespan of schedule increases as a result.

According to the Figure 2.11, the coefficients of extra lateness, δ, holding, h, and reallocating, ω, and extra processing time, τ, have quite similar trends in terms of Cost/UC. They have lower impact of changes in Cost/UC than μ, η and θ and they are robust between 0.1 and 10.

Table 2.12: Sensitivity analysis of coefficients for new order arrival

		Order arrival										
		Mks	Dev	U/C	Cost	Cost / UC			Mks	Dev	U/C	Cost	Cost / UC
	0 1	62	316	78	400.2	5.13		0.1	63	357	26	520 6	20.02
	1	63	537	87	544	6 25		1	63	357	26	544	20 92
δ	10	63	334	69	1392	20.17	w	10	64	327	25	744	29.76
	50	64	353	68	5282	77.68		50	68	509	16	1578	98 63
	100	64	353	68	10132	149		100	70	508	13	2100	161.54
	0 1	64	323	31	482 1	15.55		0.1	61	309	9	471.9	52.43
	1	63	357	15	544	36 27		1	63	357	8	544	68
m	10	64	375	5	655	131	η	10	66	462	3	767	255 67
	50	67	502	0	743	743		50	67	509	3	826	275.33
	100	67	502	0	743	743		100	67	450	3	1051	350.33
	0 1	63	320	306	226	0 74		0 1	63	357	34	513.4	15 1
	1	63	357	342	544	1.59		1	63	357	34	544	16
h	10	62	305	276	2965	10.74	t	10	62	282	22	616	28
	50	62	305	276	14005	50.74		50	61	229	12	924	77
	100	62	305	276	27805	100 74		100	61	263	8	1184	148
	0.1	62	338	1	516 9	516 9							
	1	63	357	3	544	181 33							
θ	10	65	369	17	400	23.53							
	50	68	458	25	-485	-19.4							
	100	69	493	29	-2109	-72 72							

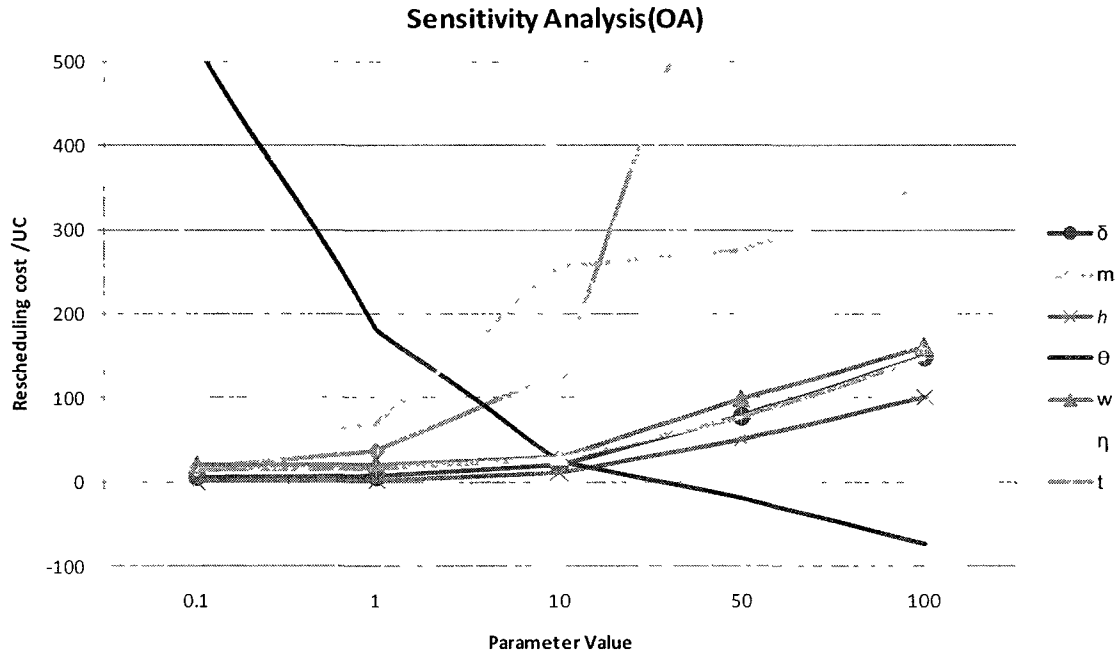


Figure 2.11: Cost of rescheduling changes due to changing the coefficients in case of new order arrival.

2.8 Conclusions

This chapter has attempted to address a new practical method in rescheduling problems in flexible manufacturing systems. For this purpose, a comprehensive cost measure is developed in order to assess the negative impact of changes due to rescheduling actions. A filtered beam search algorithm is applied in order to generate schedules in a cost-efficient manner. An illustrative example is represented in order to compare the performance of the proposed methodology with the traditional methods in the literature. A sensitivity analysis is performed to evaluate the impact of each cost function parameter on the optimal solution level. The conclusions can be outlined in the following points:

- Cost based rescheduling approach outperforms the traditional methods in terms of rescheduling cost, while the updated schedules have competing efficiency and acceptable stability compared with the updated schedules generated by other methods.

- The difference in rescheduling cost between the cost based approach and other methods are more significant for early disruptions.

- The efficiency of the updated schedules generated by the time based measures is slightly better than the ones generated by the cost based measure, but as the time-based measures fail to consider the effect of actual changes into the system, the corresponding cost due to rescheduling by these criteria are higher than the cost-based measure, which makes them impractical for implementation.

- Employing the cost based measure through the procedure of rescheduling may ensure to generate schedules having the acceptable efficiency and stability, as the cost function inherently includes of time based and stability based criteria. Hence, the rescheduling cost can be used as a surrogate measure which incorporates the both aspects of the time based and stability based measures.

- The results of sensitivity analysis reveals that the cost of expediting the material and the coefficient of saving on due dates are key parameters in the proposed cost function and changing them results in significant changes in the optimal solution.

Chapter 3

Cost based scheduling repair in FMS using FBS

Responding to disruptions with total rescheduling (TR), explained in Chapter 2, may lead to a new optimized schedule, but it creates system nervousness and causes high operational cost due to substantial changes from the original schedule. For example, performing TR in case of a job cancellation may cause significant changes in machine allocation and starting times of operations. These changes create extra cost due to expediting, holding and reallocating the material. In these cases, only modifying the affected operations is preferable to total rescheduling where we optimize a time related objective function.

Partial rescheduling or repair methods revise the disrupted schedule and generate the updated schedule with minor deviations from the original one. This may help generating the updated schedules with a lower rescheduling cost and fewer deviations more rapidly than TR.

In this chapter, a real time scheduling repair methodology is developed in order to respond to disruptions in a cost efficient manner in FMS environments. A modified Filtered-Beam-Search-heuristic repair algorithm (MFBSR) is proposed to generate a pre-specified number of cost efficient suboptimal repair schedules. For this purpose, the cost measure developed in Chapter 2 is applied in generating the schedules with MFBSR. The performance of the proposed repair methodology is compared with Total Rescheduling (TR) and modified Affected Operations Rescheduling (mAOR) in terms of rescheduling cost and makespan efficiency. A factorial experiment is performed to illustrate the effect of different rescheduling methods on the performance levels for various types and magnitude of disruptions at different flexibility levels of a manufacturing system. The results reveal that the proposed repair methodology could be considered as a competitive method to repair schedules in flexible manufacturing systems and to identify the decision point to implement a total rescheduling approach.

3.1 Problem Statement

In this study, a flexible manufacturing system with partial flexibility is considered. There are a number of jobs to be scheduled, each having a number of operations with alternative machines capable of performing the same operation albeit with different processing times. During the execution of the schedule, the following types of disruptions are considered: machine breakdown, job cancellation, and new order arrival. At the time of disruption (TOD), a repaired schedule will be generated based on the availability of the machines and remaining available jobs while minimizing the rescheduling costs of switching from the initial schedule. The initial schedule is generated using filtered beam search method (FBS) according to a makespan based objective function [39] . The

updated schedule is generated by applying modified version of FBS, in order to minimize the cost of rescheduling. Following are the assumptions considered in this study:

- The jobs are non-preemptive.
- An operation cannot be performed on more than one machine at the same time.
- Each machine cannot perform more than one operation at the same time.
- The machines are independent of each other and all are available at $t=0$.
- Machines' set-up times and material handling time are not considered.
- The jobs are independent of each other and can be done at any time separately.
- Processing time is deterministic and fixed during the horizon based on the process plan.

The following section will describe the proposed FBS based methodology to generate updated schedules. An illustrative example will follow where the proposed methodology is compared with TR and mAOR under different disturbance scenarios, system configuration levels, and performance metrics.

3.2 Cost based Repair Methodology Using FBS

FBS, which is explained in section 2.4.1, is an AI-based heuristic search method widely used in scheduling/rescheduling problem in FMS environments due to its high quality of solutions and calculation speed. An FBS-based total rescheduling (TR) methodology was presented in Chapter 2 in order to generate the cost efficient updated schedules upon the occurrence of disruptions. In this section, the FBS is employed in developing the partial rescheduling methodology in order to repair the schedules cost-efficiently. Towards this end, a filtered beam search heuristic based method called MFBSR is developed.

3.2.1 Heuristic based Repair Action

MFBSR is a modified version of FBS introduced by Wang *et al.* [39] which is applied to generate several cost efficient repaired schedules. This method can be described as follows:

In order to generate the updated schedule by MFBSR, the remaining operations and availability of each machine need to be determined. We define the remaining operations that have not begun by the time of disruption (TOD). This implies that the operations which are in-progress at TOD should be completed first, and each machine's available time is obtained as a result. For each disruption at TOD, the schedule has to be repaired for the remaining operations.

Two main significant considerations have been made in branching procedure (M_NONDELAY) of FBS [39] to make the repaired schedule close to the initial one. These modifications are as follows:

- It is considered that the sequence of selecting the schedulable operations is the same as the sequence of their starting times in the initial schedule. This leads to creating schedules with the lower deviation in the starting time of operations compared to the initial one.
- It is assumed that operations are allocated to the same machine as the initial schedule allocation. Keeping the same allocation of machines results in less deviations and corresponding changes in the manufacturing system.

In this study, the proposed branching scheme named PM_Nondelay is explained as follows:

Let PS_l be a partial schedule containing l scheduled operations, $S^{initial}$ be a list of all scheduled operations in the initial schedule sorted based on their starting times, $s_{ij}^{initial}$. Let S_l^{temp} be the set of schedulable operations at level l, and S_l^{new} be a subset of S_l^{temp} consisting of operations with minimum starting time according to the initial schedule. Let s_{ij}^{new} be the earliest possible starting time of operation in the S_l^{new}, and $s^{new}_{(i+1)j}=s^{new}_{ij}+\gamma_{ij}$ in which $\gamma_{ij}=\min_k (p_{ijk})$. M_{ijk} is the machine assigned to each operation in the initial schedule, and M_avl_k is the updated available time for machine k at TOD. The PM_Nondelay procedure is explained as follows:

Step 1: Determine $T^* = \min_{O_{ij} \in S_l^{temp}} \{s_{ij}^{initial}\}$, and add the operations O_{ij} with $s_{ij}^{initial}=T^*$ in the set S_l^{new}.

Step 2: Select an operation $O_{ij} \in S_l^{new}$, assign the M_{ijk} to it, and generate a new node corresponding to partial schedule, in which O_{ij} is added to PS_l. Update starting times as s_{ijk}^{new}= Max (s_{ij}^{new}, M_avl_k) indicating that O_{ij} is allocated to machine k with a starting time of s_{ijk}^{new}.

3.2.2 Proposed MFBSR algorithm

The procedural form of the MFBSR is given as follows:

Step 1: Initialization:

Let *bn=0, l=0*; input *beamwidth b*, *filterwidth f*; input number of machines, operations, and jobs; let the partial schedule PS_l , initial schedule IS_l , and remaining list RL_l be empty.

Step 2: Re-initialization:

(i) Let all machines availability be at the time of disruptions, M_avl_k = TOD and go to step 2 (ii).

(ii) Get the original schedule and put it into IS_l and go to step 2 (iii).

(iii)For operations in IS_l, if ST_{ijk}< TOD, then add them to PS_l and go to step 2 (iv); else add them to RL_l and go to step 2 (v).

(iv)Update machine availability (M_avl_k) of the operations in PS_l based on the type of disruption and go to step 2 (vi).

(v) Update the RL_l at TOD based on the type of disruptions.

(a) For order arrival, add the operations of new order to RL_l.

(b) For job cancellation, remove the operations of cancelled job from RL_l.

(vi)Let l be the number of operations in the PS_l.

Step 3: Determining the beam nodes:

(i) Generate nodes from the root nodes by PM_Nondelay from RL_l. Check the total number of nodes generated N. Let $l=l+1$, updated PS_l and RL_l with the generated nodes.

(ii) If N< b, then go to the next level and generate nodes using PM_Nondelay with PS_l as a partial schedule, update PS_l, update N. If N< b go to step 3(ii); else go to step 3(iii).

(iii)Compute the global evaluation function for each node and select the best b ones (initial beam nodes). Determine the potential sets of $PS_l(1)$, $PS_l(2)$,... $PS_l(b)$.

Step 4: $bn=bn+1$; if $bn>b$, then go to step 5; else go to step 4(i).

(i) For each initial beam node generated, form beam nodes of level l as follows:

(a) $l=l+1$, if $l>n$, then go to step 4 (ii); else move on.

(b) Generate new nodes from the beam node according to PM_Nondelay with $PS_l(n)$ as partial schedule.

(c) Compute the local evaluation function values for all nodes generated and select f numbers of nodes with the best values for future evaluation

(d) Compute the global evaluation function values for f number of nodes, select the nodes with the best value and add the node into the partial schedule $PS_l(bn)$. Go to (a).

(ii) Formulate the bnth complete schedule $PS(bn)$.

Step 5: Select the schedule with the best objective function value among the final b schedules set; Stop.

The proposed methodology is described in details for three types of disruptions, i.e., machine breakdown, order arrival, and job cancellation as follows:

3.2.2.1 Machine Breakdown

In case of machine breakdown at TOD, the pre-identified repair time is inserted into the schedule for the failed machine and earliest available time of each machine is updated. Having the remaining operations' starting time and machine availability, the system is ready to be repaired for remaining operations. This action is carried out by MFBSR in order to repair the existing schedule.

3.2.2.2 New Order Arrival

In case of a new order arrival to the system at TOD, all operations of the new job are considered as regular remaining operations to be repaired by MFBSR. The new operations are put in the set of schedulable operations and are scheduled to minimize the rescheduling cost.

3.2.2.3 Job Cancellation

When a job is cancelled at TOD, all the remaining operations of the cancelled jobs are removed. If an operation of the cancelled job is in-progress, it is stopped at TOD and the remaining part is removed. Lastly, all the remaining operations are repaired by MFBSR.

3.3 Experimental Design

A factorial experimental analysis is carried out on the three mentioned types of disruptions to evaluate the performance of the proposed repair methodology. The quality of the methodology is compared with mAOR and total rescheduling (TR) with respect to rescheduling cost and efficiency measures. Moreover, the effects of various disruption scenarios and system flexibility levels on the performance metrics are studied.

3.3.1 Experimental Factors

After a careful study of the relevant literature and problem characteristics, four experimental factors are selected: rescheduling method, timing of disruption, magnitude of disruption, and flexibility of the system.

- Reactive scheduling Method (MTD): This factor is used to identify the effects of different rescheduling methods on the performance levels of updated schedules.
- Time of the disruption (TIM): This factor refers to the time of occurrence of the disruptions relative to the makespan. This timing can be early or late during the horizon.
- Magnitude of the disruption (MAG): The magnitude of disruptions is considered as the number of disruptions that occurs at the time of disruption.
- Flexibility of the system (FLX): Flexibility of the system is defined as the average number of capable machines for each operation.

3.3.2 Dimensions of the experiments

MTD has three treatment levels reflecting the proposed methodology, TR and mAOR. For the remaining factors, two levels of treatment, low and high, are specified as shown in Table 3.1. This results in 2^3=8 experimental combinations for each type of disruption and method. The designed experiments shown in Table 3.2 are performed in ten replications. The assumptions regarding the occurrence of each type of disruption are as follows:

- Machine breakdown: A machine is selected to fail randomly.

- New order arrival: It is supposed that a new order is randomly selected from the existing set of jobs.
- Order cancellation: A job is randomly selected to be cancelled.

The values of initial schedule parameters are the same as the ones used in Chapter 2 as shown in Table 2.6. The selected coefficients of the proposed rescheduling cost functions are given in Table 2.3.

Table 3.1: Levels of experiment

Experimental factor		Level 1 (Low)	Level 2 (High)
Time of disruption (TIM)		60-90% of makespan Late	5-40% of makespan Early
Magnitude of disruption (MAG)			
	Machine breakdown	1 machine breakdown	3 machine breakdown
		Repair Time: Uniformly Distributed [5,10]	
	New order arrival	1 order arrival	3 order arrival
	Order cancellation	1 job cancellation	3 job cancellation
Felxibility of system (FLX)			
		30-40% system flexibility 2.1 - 2.8 mach/operation	80-100% system flexibility 5.6 - 7 mach/operation

Table 3.2 : Experimental combinations

Expt. No.	FLX	MAG	TIM
1	Low	High	Early
2	Low	High	Late
3	Low	Low	Early
4	Low	Low	Late
5	High	High	Early
6	High	High	Late
7	High	Low	Early
8	High	Low	Late

3.3.3 Performance measures

3.3.3.1 Efficiency

Efficiency of the updated schedule is defined as a percentage change in its makespan with respect to the initial schedule [13,18] as shown in Equation (2-12).

3.3.3.2 Rescheduling Cost

The cost function developed in Section 2.3 is used as a performance measure to compare the rescheduling cost of different methods in various circumstances. It should be noted that, during the experiments, rescheduling cost is used as an objective function in generating the schedules by TR and MFBSR. In order to compare the rescheduling cost among methods, the amount resulting from TR is considered as a base level. Then, the performance levels of mAOR and MFBSR are expressed as a percentage relative to this base level. Considering the rescheduling cost as a performance measure allows incorporating the characteristics of stability measure through Equations (2-6, 2-7 and 2-8) as well as schedule efficiency measures, i.e., due dates and makespan through Equations (2-4 and 2-5). Therefore, the rescheduling cost and efficiency are considered in this study to evaluate the performance of the updated schedules.

3.4 Experiment Results

This section outlines the results of experiments outlined in Table 3.2. The average change in performance metrics as a result of changing the level of each factor are represented for each type of disruption. The main effect of changing the flexibility levels of the system (FLX) at different factors' levels is studied. All the algorithms are coded in JAVA,

Eclipse V.3.5.1. Figures 3.1-3(a,b) show the effects of different methods (MTD) on efficiency and rescheduling cost, for each disruption type.

3.4.1 Machine Breakdown

The results for machine breakdown show that, the efficiency of MFBSR is always dominant to mAOR as shown in Figure 3.1(b). This is because the MFBSR algorithm performs local and global search for each of the remaining operation in order to generate a cost efficient updated schedule. In addition, as shown in Figure 3.1(a), MFBSR performs better than mAOR in terms of rescheduling cost for late disruptions (Experiments 2, 4, 6, and 8). In early disruptions, MFBSR results in a higher rescheduling cost than mAOR. This is due to the fact that in the early stages of a schedule there is higher number of operations to be reallocated.

Comparing the efficiency of TR and MFBSR, Figure 3.1(b) shows that in Experiments 6 and 8, TR performs better and creates schedules with lower makespan than MFBSR while also resulting in low cost as indicated in Table 3.3. These cases refer to the late disruptions in a high flexible system. In those cases, TR may create schedules with lower makespan and also lower cost by exploiting flexibility of the manufacturing system. For early disruptions, the TR reallocates operations to alternative machines which require longer processing time, due to the fact that the initial schedule was generated based on SPT.

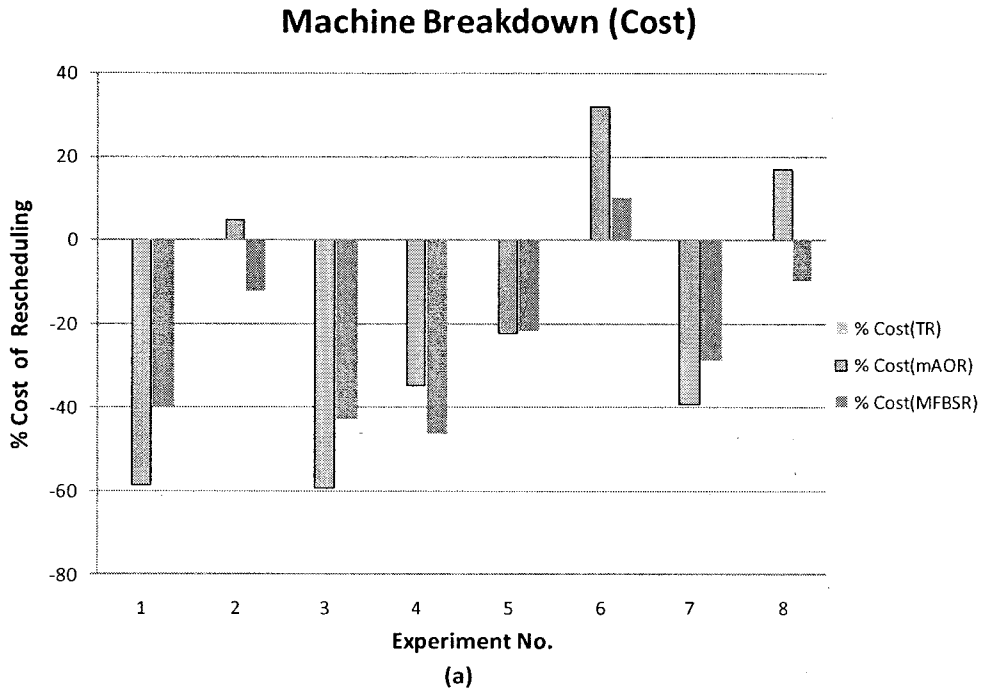
The results also reveal that, by increasing the flexibility level of the system, the efficiency of the updated schedules by TR is increased. Figure 3.1(b) represents that the efficiencies of Experiments 5, 6, 7, and 8 when generated by TR are higher than Experiments 1, 2, 3, and 4. Fahmy *et al.* [13] also mention that introducing flexibility into

the system results in increasing efficiencies of updated schedules. In a low flexible system, since there is not enough alternative for operations to switch, the makespan and cost of TR are higher than MFBSR.

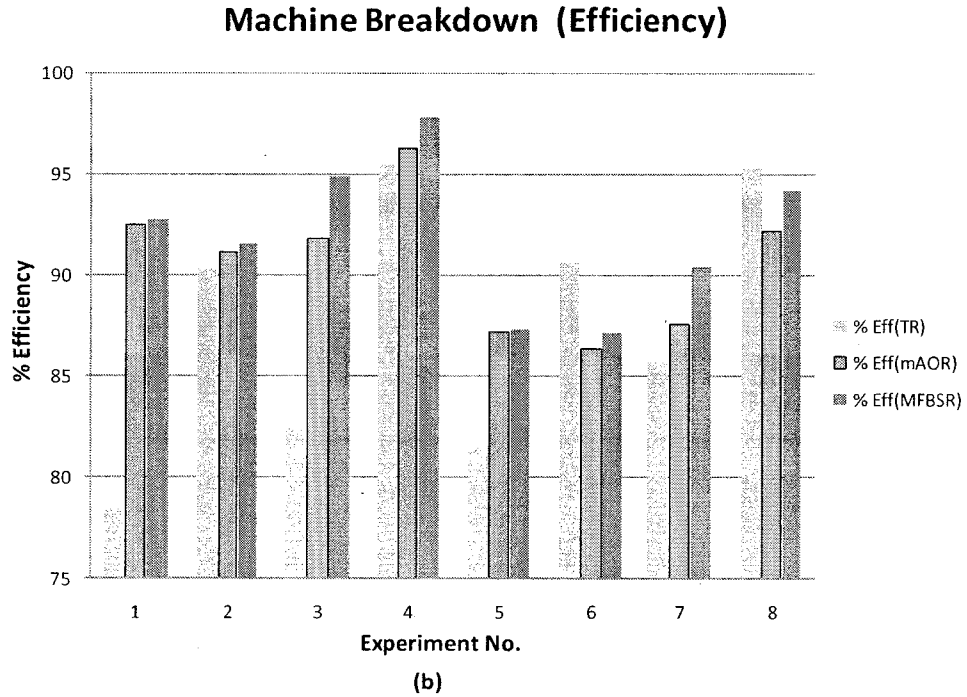
The results also show that flexibility level has direct impact on the rescheduling cost. Table 3.5 shows that the cost of rescheduling in a highly flexible system is less than rescheduling cost in a low flexible system regardless of the rescheduling method.

Table 3.3: Rescheduling cost performance of disruption scenarios

	Disruption								
	Machine breakdown			Job cancellation			Order arrival		
Experiment	TR	mAOR	MFBSR	TR	mAOR	MFBSR	TR	mAOR	MFBSR
1	4112.9	1705.6	2472.3	1352.5	1644.8	2035.9	7295	4238.4	5220.9
2	1415.8	1482.7	1244.4	482.1	286.4	291.1	3915.2	2882.4	3377.5
3	3402.6	1388.9	1954.9	1957.6	619.6	1401.4	3343.3	1699.8	2440.1
4	960.4	628.2	516.5	432.5	71.8	129.2	1708.1	1333.2	1187.1
5	2920.4	2270.2	2283.2	1019.2	1244.2	1541.6	5826.2	2779.2	4337.4
6	800.3	1056.1	881.1	117.4	59.2	55.8	2077.9	1689.6	1862.9
7	2604.7	1582.3	1857.3	1239.5	524.4	995.7	2812.2	1233	1973.8
8	369.2	431.8	333.2	108.9	31.8	36.7	932.6	934.8	854.1



(a)



(b)

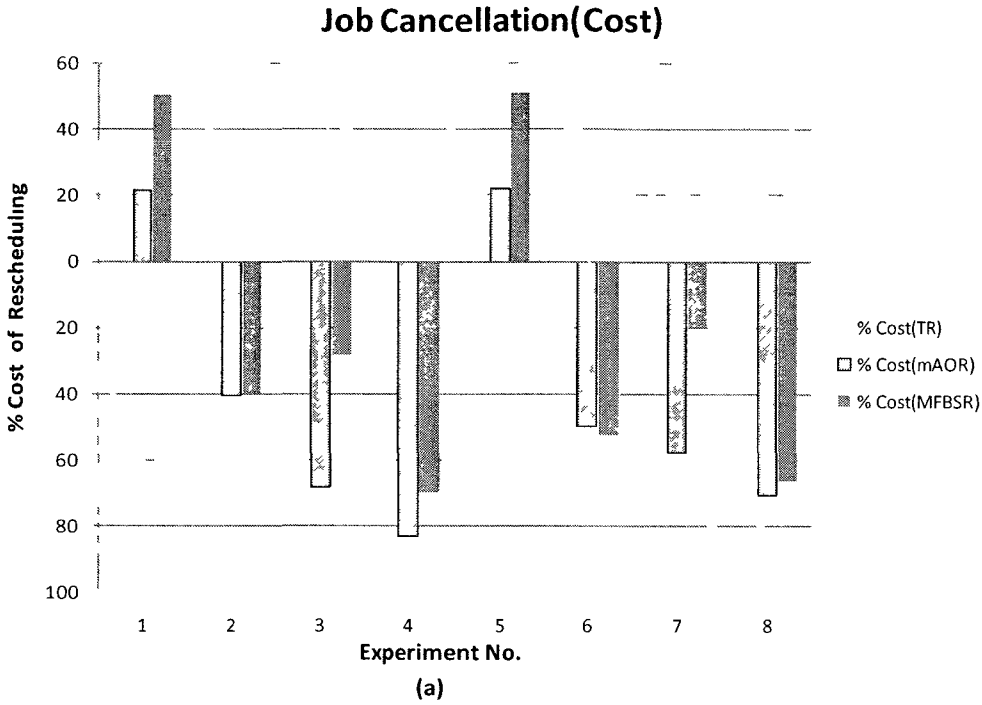
Figure 3.1: Machine breakdown: (a) cost of rescheduling; (b) efficiency.

3.4.2 Job Cancellation

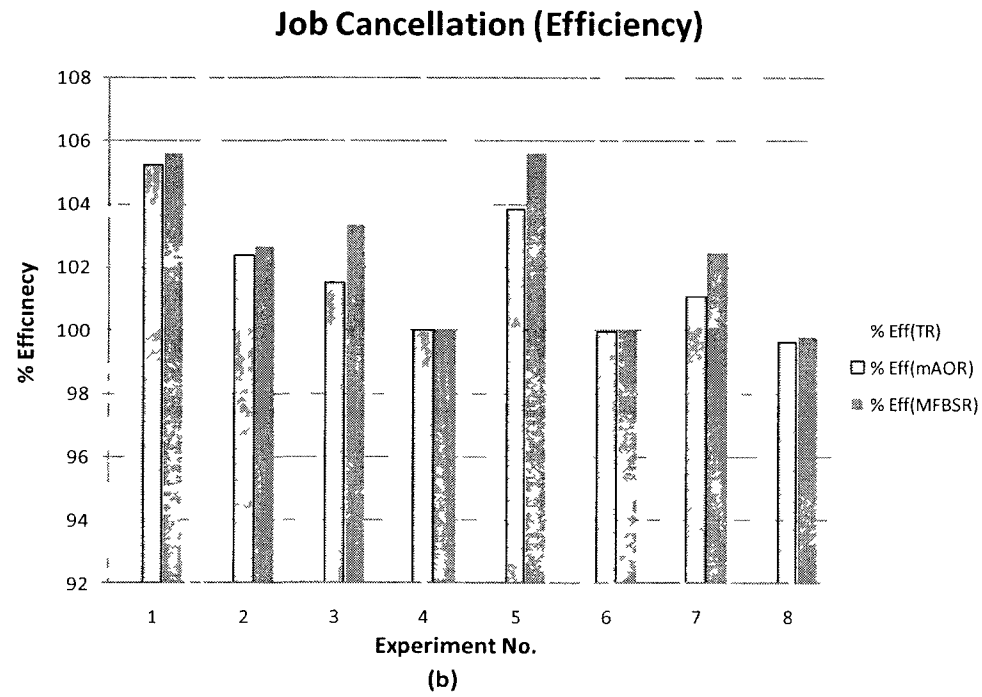
The results for job cancellation show that MFBSR outperforms mAOR in terms of efficiency as shown in Figure 3.2(b). For late disruptions, Figure 3.2(a) shows that MFBSR generates similar cost levels compared to mAOR while dominating in efficiency regardless of the magnitude. Therefore, MFBSR should be preferred in responding late job cancellations.

For early and high magnitude disruptions, TR is capable of generating low rescheduling costs while keeping the efficiency levels close to the original makespan. This means that if there is early and significant job cancellations a TR approach should be preferred. For early and low magnitude disruptions, repair methods tend to work well in terms of efficiency and cost.

It can be seen from Figure 3.2(b) that, by increasing the flexibility level of the system, the efficiency of TR is significantly improved. Hence, the gap between the efficiency of TR and others is decreased as a result. Similarly, the rescheduling cost of updated schedules decreases by enhancing flexibility level of the system regardless of rescheduling method.



(a)



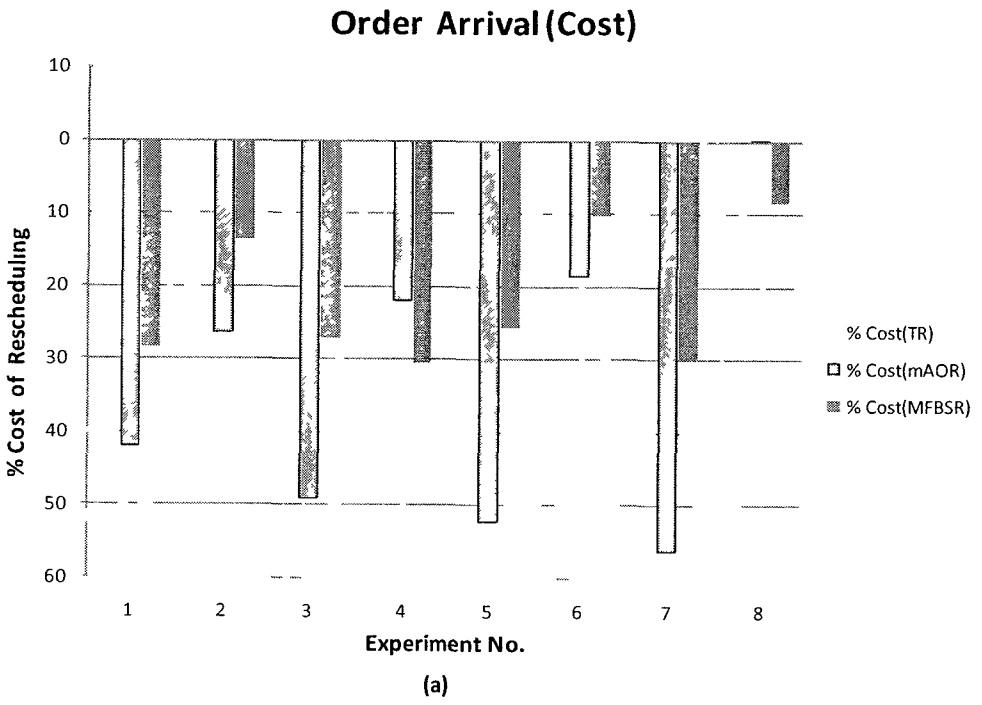
(b)

Figure 3.2 : Job cancellation: (a) cost of rescheduling; (b) efficiency

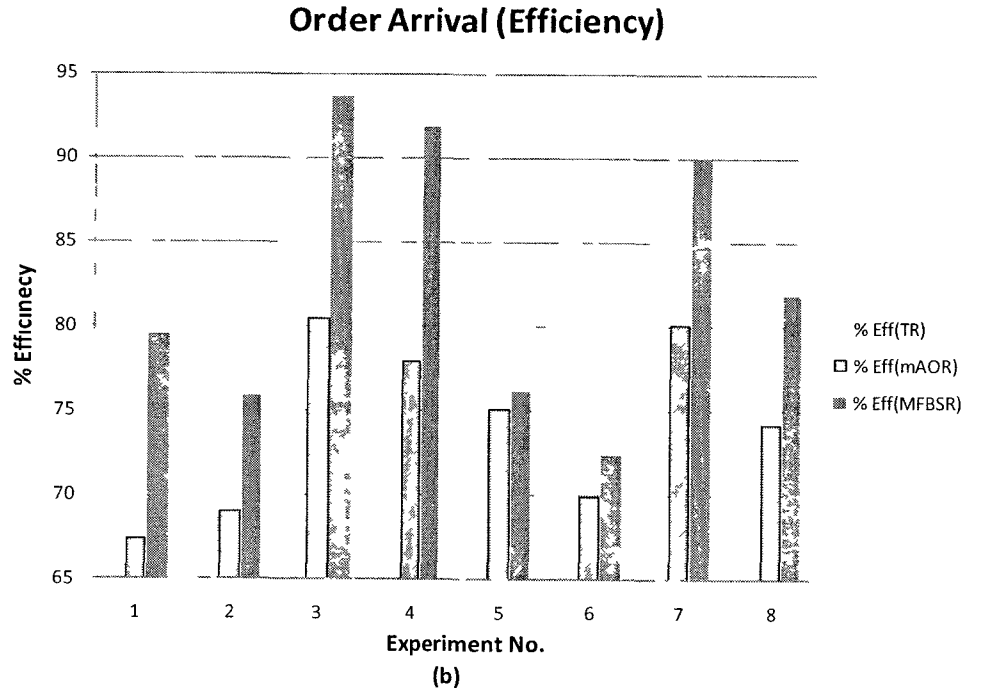
3.4.3 New order arrival

The results of the new order arrival in Figure 3.3(b) show that, the efficiency of updated schedules by MFBSR is significantly higher than the ones generated by mAOR. As shown in Figure 3.3(a), adding the new orders to the end by mAOR may create lower rescheduling cost; however, the differences between the efficiencies of the updated schedules are significant. In case of a new regular order arrival, mAOR checks the available slots in the original schedule against the processing time of new jobs' operations. If there is an available slot, the operation is allocated; otherwise, it is allocated at the end of original schedule [11]. In other words, mAOR keeps the initial schedule the same and update it by adding the new operations in a suitable position. Hence, the schedules generated by this method have minimal deviations from the original one, at the expense of makespan.

It can be seen from Figure 3.3(b) that the efficiency of updated schedules by MFBSR is slightly better than the ones updated by TR in early disruptions. In late disruptions, the efficiency of TR is slightly better than MFBSR, due to exploiting the flexibility in rescheduling process. But in terms of rescheduling cost, Table 3.3 shows that TR is always much worse than MFBSR. As a result, MFBSR is an effective method in dealing with order arrivals compared to TR approach.



(a)



(b)

Figure 3.3: Order arrival: (a) cost of rescheduling, (b) efficiency

3.5 Conclusions

This chapter introduced a cost based reactive methodology for repairing the schedules on FMS environments using a modified FBS algorithm. The FBS is also employed in generating the initial schedules and the updated schedules by total rescheduling (TR). A compound cost measure is developed to incorporate the competing effects of efficiency and stability and assessing the negative impact of changes due to rescheduling. This measure could be used as an objective function in generating cost-efficient schedules. An extensive comparative study is done in order to evaluate performance of the proposed repair method with other rescheduling methods from literature. The conclusions can be outlined in the following points:

- MFBSR noticeably outperforms the mAOR method in terms of efficiency of the updated schedules. The difference is more significant in early disruptions.
- Regardless of the approach used, increasing the flexibility level of the system results in a lower rescheduling cost.
- For early disruptions, while mAOR creates the updated schedules with lower rescheduling cost, MFBSR performs significantly better in terms of efficiency while its rescheduling cost is better than TR.
- For late disruptions, the gap between rescheduling cost of mAOR and MFBSR decreases, while MFBSR keeps the efficiency of updated schedules higher.
- In a highly flexible system, TR performs better in terms of efficiency for late disruptions while the rescheduling cost associated with these schedules are higher than the ones generated by repair methods.

- The efficiency of the updated schedules generated by the repair methods decreases by increasing the flexibility level of the system. The initial schedules generated in a low flexible system have lower machine utilization and result in a higher makespan compared to a system with increased flexibility. This is due to the slack times in the initial schedules resulting from low flexibility. Hence, if the robustness is the primary concern, introducing slack times into the schedule generates better results than introducing flexibility.

Chapter 4

Conclusions and Future work direction

Updating the production schedule in order to respond to uncertainties and disturbances is a crucial task in flexible manufacturing systems. Traditional rescheduling methods are designed based on either time based criterion in order to satisfy customers' needs and production rate, or stability based criterion to minimize the changes due to rescheduling. These two criteria are contradictory objectives, which mean that improving one may lead to degrading the performance of the other one. Hence, the first objective of this thesis is to develop a proficient measure, called the rescheduling cost, in order to comprise all aspects of the former measures whilst considering the corresponding cost of rescheduling action. Second, a cost based rescheduling methodology has been developed to totally reschedule the system at the time of each disruption. Third, a cost-based scheduling repair methodology has been introduced to generate the updated schedules with fewer

changes, less cost and in faster speed than TR. Unlike the traditional repair methodologies, the proposed scheduling repair approach generates the updated schedule based on an optimization procedure, which minimizes the rescheduling cost to enhance the quality of the updated schedules. In order to achieve these objectives, the following issues have been explored within this work:

- First, a new cost function has been developed representing the cost of rescheduling action in FMS environments. The cost function incorporates the aspects of time based and stability based measures and can be used as an objective function in generating the updated schedules.

- Second, a cost based total rescheduling methodology has been presented by using the proposed cost function into a filtered beam search heuristic algorithm process in order to generate cost efficient updated schedules.

- Third, a cost based scheduling repair methodology has been developed by using the proposed cost function into a modified filtered beam search algorithm process in order to generate cost-efficient updated schedules.

- The case studies and experimental design have been performed in order to validate and compare the results of each objective with other methods in the literature.

The outcome of these methods is a cost-efficient updated schedule responds to various types of disruptions and circumstances in FMS environments.

4.1 Conclusions

A factorial design of experiment has been presented for each objective of this thesis in order to demonstrate the use of the developed approaches and compare their performance with similar approaches in the literature. The following results can be pointed out from this research:

1. The results of the cost based rescheduling approach show the advantage of using the rescheduling cost as a performance measure in generating the updated cost-efficient schedules in FMS environments. The updated schedules generated by TR and MFBSR in cost based approach outperform other approaches in the literature.
2. Using the rescheduling cost as a performance measure results in generating the updated schedules that have an acceptable level of efficiency and stability, as the rescheduling cost function inherently consists of time based and stability based criteria.
3. The efficiency of the updated schedules generated by the time based measures are slightly better than the efficiency of the ones generated by the cost based approach. However, due to increased level of changes in rescheduling process, the corresponding rescheduling cost are much higher than cost based approach schedules, which makes them impractical to implement in FMS environments.
4. The efficiency of the updated schedules by the proposed repair approach (MFBSR) is noticeably higher than the ones generated by traditional repair approach (mAOR). It has been shown that for early disruptions, this deference is more significant.

5. The rescheduling cost of updating the schedules by mAOR is lower than MFBSR and TR, especially for early disruptions. As the method only shifts the affected operations when responding to disruptions, the corresponding changes and rescheduling cost would be lower than TR and MFBSR as a result. Hence, mAOR would fail in satisfying the efficiency criteria.
6. Increasing the flexibility level of the system results in decreasing the cost of rescheduling action regardless of methods and approaches.
7. By comparing the results of TR and MFBSR for each disruption scenario, the cost-efficient methodology for handling disruptions in FMS has been presented. This thesis shows that if a disruption occurs late in a highly flexible system, TR performs better in terms of efficiency of the updated schedules while their corresponding rescheduling cost is higher than the ones generated by MFBSR. For other cases of disruption scenarios, MFBSR could be the choice as it generates schedules with higher efficiency than mAOR whilst the corresponding rescheduling cost is comparable.
8. Introducing the slack time into the initial schedule generation procedure to make the schedule robust, results in better efficiency of the updated schedules than introducing flexibility into the system.

4.2 Future research directions

The following topics can be further explored for extension of the present research work:

1. Using the cost based approach in order to develop a reactive scheduling tool in responding to multiple disruptions in FMS environments. This can be performed by assigning the more values to the cost function parameters for the short term period after a disruption, in order to reduce the effects of immediate changes in the schedule.

2. Comparing the cost of introducing slack time in schedule generation with the cost of investing in system flexibility in order to reduce rescheduling costs.

3. Additional types of disruption can be studied using the cost based approach in order to investigate the performance of the updated schedules.

Bibliography

[1] Vieira GE, Herrmann JW, Lin E. Rescheduling manufacturing systems: a framework of strategies, policies, and methods. *Journal of Scheduling.* 2003;6(1):39-62.

[2] Groover MP. Automation, production systems, and computer-integrated manufacturing. Upper Saddle River, N.J.: Prentice Hall, 2008.

[3] Hassanzadeh P, Maier-Speredelozzi V. Dynamic flexibility metrics for capability and capacity. *International Journal of Flexible Manufacturing Systems.* 2007;19(3):195-216.

[4] Dutta A. Reacting to scheduling exceptions in FMS environments. *IIE Transactions (Institute of Industrial Engineers).* 1990;22(4):300-314.

[5] Abumaizar RJ, Svestka JA. Rescheduling job shops under random disruptions. *International Journal of Production Research.* 1997;35(7):2065-2082.

[6] Sabuncuoglu I, Karabuk S. A beam search-based algorithm and evaluation of scheduling approaches for flexible manufacturing systems. *IIE Transactions.* 1998;30(2):179-91.

[7] Shafaei R, Brunn P. Workshop scheduling using practical (inaccurate) data - Part 1: The performance of heuristic scheduling rules in a dynamic job shop environment using a rolling time horizon approach. *International Journal of Production Research.* 1999;37(17):3913-3925.

[8] Shafaei R, Brunn P. Workshop scheduling using practical (inaccurate) data Part 2: An investigation of the robustness of scheduling rules in a dynamic and stochastic environment. *International Journal of Production Research.* 1999;37(18):4105-4117.

[9] Min HK, Yeong-Dae Kim. Simulation-based real-time scheduling in a flexible manufacturing system. *Journal of Manufacturing Systems.* 1994;13(2):85-93.

[10] Jain AK, Elmaraghy HA. Production scheduling/rescheduling in flexible manufacturing. *International Journal of Production Research.* 1997;35(1):281-309.

[11] Subramaniam V, Raheja AS. mAOR: A heuristic-based reactive repair mechanism for job shop schedules. *International Journal of Advanced Manufacturing Technology.* 2003;22(9):669-680.

[12] Suwa H, Sandoh H. Capability of cumulative delay based reactive scheduling for job shops with machine breakdowns. *Computers & Industrial Engineering.* 2007;53(1):63-78.

[13] Fahmy SA, Balakrishnan S, Elmekkawy TY. A generic deadlock-free reactive scheduling approach. *International Journal of Production Research.* 2009;47(20):5657-5676.

[14] Sabuncuoglu I, Goren S. Hedging production schedules against uncertainty in manufacturing environment with a review of robustness and stability research. *International Journal of Computer Integrated Manufacturing.* 2009;22(2):138-157.

[15] Te-Wei Chiang, Hai-Yen Hau. Cycle detection in repair-based railway scheduling system. In: *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 3. New York, NY, USA: IEEE, 1996. 2517-22.

[16] Hall NG, Potts CN. Rescheduling for new orders. *Operations research.* 2004;52(3):440-453.

[17] Li L, Jiang Z. Self-adaptive dynamic scheduling of virtual production systems. *International Journal of Production Research.* 2007;45(9):1937-1951.

[18] Subramaniam V, Raheja AS, Rama BR. Reactive repair tool for job shop schedules. *International Journal of Production Research.* 2005;43(1):1-23.

[19] Honghong Y, Zhiming W. The application of adaptive genetic algorithms in FMS dynamic rescheduling. *International Journal of Computer Integrated Manufacturing.* 2003;16(6):382-97.

[20] Wang Shi-jin, Xi Li-feng, Zhou Bing-hai. Filtered-beam-search-based algorithm for dynamic rescheduling in FMS. *Robotics and Computer-Integrated Manufacturing.* 2007;23(4):457-68.

[21] Sabuncuoglu I, Erel E, Gocgun Y. Analysis of serial production lines: Characterisation study and a new heuristic procedure for optimal buffer allocation. *International Journal of Production Research.* 2006;44(13):2499-2523.

[22] Church LK, Uzsoy R. Analysis of periodic and event-driven rescheduling policies in dynamic shops. *International Journal of Computer Integrated Manufacturing.* 1992;5(3):153-63.

[23] Fang J, Xi Y. Rolling horizon job shop rescheduling strategy in the dynamic environment. *International Journal of Advanced Manufacturing Technology.* 1997;13(3):227-232.

[24] Sabuncuoglu I, Kizilisik OB. Reactive scheduling in a dynamic and stochastic FMS environment. *International Journal of Production Research.* 2003;41(17):4211-4231.

[25] Aytug H, Lawley MA, McKay K, Mohan S, Uzsoy R. Executing production schedules in the face of uncertainties: A review and some future directions. *European Journal of Operational Research.* 2005;161(1):86-110.

[26] Sabuncuoglu I, Goren S. Hedging production schedules against uncertainty in manufacturing environment with a review of robustness and stability research. *International Journal of Computer Integrated Manufacturing.* 2009;22(2):138-157.

[27] Sabuncuoglu I, Karabuk S. Rescheduling frequency in an FMS with uncertain processing times and unreliable machines. *Journal of Manufacturing Systems.* 1999;18(4):268-83.

[28] Yamamoto M, Nof SY. Scheduling/rescheduling in the manufacturing operating system environment. *International Journal of Production Research.* 1985;23(4):705-722.

[29] Wu SD, Eui-Seok Byeon, Storer RH. A graph-theoretic decomposition of the job shop scheduling problem to achieve scheduling robustness. *Operations research.* 1999;47(1):113-24.

[30] Wu SD, Erkoc M, Karabuk S. Managing Capacity in the High-Tech Industry: A Review of Literature. *The Engineering Economist: A Journal Devoted to the Problems of Capital Investment.* 2005;50(2):125.

[31] Raheja AS, Subramaniam V. Reactive recovery of job shop schedules - A review. *International Journal of Advanced Manufacturing Technology.* 2002;19(10):756-763.

[32] Jensen MT. Generating robust and flexible job shop schedules using genetic algorithms. *Evolutionary Computation, IEEE Transactions on.* 2003;7(3):275-288.

[33] Wu SD, Storer RH, Pei-Chann Chang. One-machine rescheduling heuristics with efficiency and stability as criteria. *Computers & Operations Research.* 1993;20(1):1-14.

[34] WANG K. A resource portfolio model for equipment investment and allocation of semiconductor testing industry. *European Journal of Operational Research.* 2007;179(2):390.

[35] Low C, Yip Y, Wu T. Modelling and heuristics of FMS scheduling with multiple objectives. *Computers and Operations Research.* 2006;33(3):674-694.

[36] Min-Hong Han, Yoon KN, Hogg GL. Real-time tool control and job dispatching in flexible manufacturing systems. *International Journal of Production Research.* 1989;27(8):1257-67.

[37] Hutchison J, Leong K, Snyder D, Ward P. Scheduling approaches for random job shop flexible manufacturing systems. *International Journal of Production Research.* 1991;29(5):1053-67.

[38] Caumond A, Lacomme P, Moukrim A, Tchernev N. An MILP for scheduling problems in an FMS with one vehicle. *European Journal of Operational Research.* 2009;199(3):706-22.

[39] Wang S, Zhou B, Xi L. A filtered-beam-search-based heuristic algorithm for flexible job-shop scheduling problem. *International Journal of Production Research.* 2008;46(11):3027-3058.

[40] Pierreval H, Mebarki N. Dynamic selection of dispatching rules for manufacturing system scheduling. *International Journal of Production Research.* 1997;35(6):1575-1591.

[41] Blackstone Jr. JH, Phillips DT, Hogg GL. A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research.* 1982;20(1):27.

[42] Stecke KE, Solberg JJ. Loading and control policies for a flexible manufacturing system. *International Journal of Production Research.* 1981;19(5):481-90.

[43] Ishii N, Talavage JJ. A transient-based real-time scheduling algorithm in FMS. *International Journal of Production Research.* 1991;29(12):2501-20.

[44] Chan FTS, Chan HK, Lau HCW, Ip RWL. Analysis of dynamic dispatching rules for a flexible manufacturing system. In: *9th International Manufacturing Conference*, vol. 138. Switzerland: Elsevier, 2003. 325-31.

[45] Pierreval H, Mebarki N. Dynamic selection of dispatching rules for manufacturing system scheduling. *International Journal of Production Research.* 1997;35(6):1575-91.

[46] Balogun OO, Popplewell K. Towards the integration of flexible manufacturing system scheduling. *International Journal of Production Research.* 1999;37(15):3399-428.

[47] Brandimarte P. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research.* 1993;41(1-4):157-83.

[48] 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.* 1997;70:281-306.

[49] Najid NM, Dauzere-Peres S, Zaidat A. A modified simulated annealing method for flexible job shop scheduling problem. In: *2002 IEEE International Conference on Systems, Man and Cybernetics, October 6, 2002 - October 9*, vol. 5. Yasmine Hammamet, Tunisia: Institute of Electrical and Electronics Engineers Inc, 2002. 89-94.

[50] Xia W, Wu Z. An effective hybrid optimization approach for multi-objective flexible job-shop scheduling problems. *Computers & Industrial Engineering.* 2005;48(2):409-25.

[51] Ow PS, Morton TE. Filtered beam search in scheduling. *International Journal of Production Research.* 1988;26(1):35-62.

[52] De S, Lee A. Flexible manufacturing system (FMS) scheduling using filtered beam search. *Journal of Intelligent Manufacturing.* 1990;1(3):165-83.

[53] Liu J, MacCarthy BL. Classification of FMS scheduling problems. *International Journal of Production Research.* 1996;34(3):647-656.

[54] Li R, Shyu Y, Adiga S. Heuristic rescheduling algorithm for computer-based production scheduling systems. *International Journal of Production Research.* 1993;31(8):1815-1826.

[55] Wu H-, Li R-. New rescheduling method for computer based scheduling systems. *International Journal of Production Research.* 1995;33(8):2097-2110.

[56] Olumolade MO, Norrie DH. Reactive scheduling system for cellular manufacturing with failure-prone machines. *International Journal of Computer Integrated Manufacturing.* 1996;9(2):131-44.

[57] Bean JC, Birge JR, Mittenthal J, Noon CE. Matchup scheduling with multiple resources, release dates and disruptions. *Operations research.* 1991;39(3):470-470.

[58] Akturk MS, Gorgulu E. Match-up scheduling under a machine breakdown. *European Journal of Operational Research.* 1999;112(1):81-97.

[59] Mason SJ, Jin S, Wessels CM. Rescheduling strategies for minimizing total weighted tardiness in complex job shops. *International Journal of Production Research.* 2004;42(3):613-628.

[60] Mehta SV, Uzsoy RM. Predictable scheduling of a job shop subject to breakdowns. *IEEE Transactions on Robotics and Automation.* 1998;14(3):365-78.

[61] Hoitomt DJ, Luh PB, Pattipati KR. Practical approach to job-shop scheduling problems. *IEEE Transactions on Robotics and Automation.* 1993;9(1):1-13.

[62] Sotskov Y, Sotskova N, Werner F. Stability of an optimal schedule in a job shop. *Omega.* 1997;25(4):397-414.

[63] Leon VJ, Wu SD, Storer RH. Robustness measures and robust scheduling for job shops. *IIE Transactions.* 1994;26(5):32-43.

[64] Daniels RL, Kouvelis P. Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management Science.* 1995;41(2):363-76.

[65] Goren S, Sabuncuoglu I. Robustness and stability measures for scheduling: single-machine environment. *IIE Transactions.* 2008;40(1):66-83.

[66] Gan H-, Wirth A. Comparing deterministic, robust and online scheduling using entropy. *International Journal of Production Research.* 2005;43(10):2113.

[67] Kopanos GM, Capon-Garcia E, Espuna A, Puigjaner L. Costs for rescheduling actions: A critical issue for reducing the gap between scheduling theory and practice. *Industrial and Engineering Chemistry Research.* 2008;47(22):8785-8795.

[68] Naseri E, Kuzgunkaya O. Cost-based Rescheduling in a Flexible Manufacturing System Using Filtered-Beam Search. In: *Proceedings of the IERC 2010*: IIE, 2010.

[69] Naseri E, Kuzgunkaya O. Cost Effective Handling of Disruptions in Production Management. In: *Proceedings of 21st POMS Annual Conference*, 2010. 015-855.

[70] Sabuncuoglu I, Bayiz M. Job shop scheduling with beam search. *European Journal of Operational Research.* 1999;118(2):390-412.

[71] Baker KR. Introduction to sequencing and scheduling. Chichester, Sussex, UK: Wiley, 1974.

[72] Hillier FS, Lieberman GJ. Introduction to operations research. Boston: McGraw-Hill Higher Education, 2005.

Appendices

Appendix A: JAVA model for TR in case of Disruptions (MB- JC-OA)

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Locale;
import java.util.Scanner;

import jxl.Sheet;
import jxl.Workbook;
import jxl.WorkbookSettings;
import jxl.read.biff.BiffException;

public class mainclass {
      static final int MaxStartTime=200000;
      static final int MaxProcessTime=200000;
      static final int MNum=7;
      static final int JNum=20;
      static final int ONum=5;
      static final int BeamWidth=15;
      static final int FilterWidth=25;
      static int[][] ProcessTime;
      static final double[] ai={0.1, ,0.1};
      static final int tnow=10;  //time of disruptions
      static final int[] broken_index={4};
      static final int[] repair_time={4};
      static final int[] broken_j_index={15};
      static final int arrival_job_size=1;
      static node initial_node;
      static final double[]delta={12, ,12}; //penalty cost of lateness
      static final double[]save={12, ,12};  //saving on due dates
      static final double[] mio={10, ,10};  //Cost of expediting
      static final double[] h={1, ,1};     //Cost of holding
      static final int omega=4;         //Cost of reallocating
      static final double[] eta={6, ,6}; //Cost of idle time
      static final double tilda=6;      //cost of extra PT

public static void main(String[]args) throws
CloneNotSupportedException, BiffException, IOException{

initialise_process_time_table("C:\\Documents and
Settings\\Administrator\\Desktop\\Test_problems\\initial-input.xls");
            initial_node = initialise_start_nodes("C:\\Documents and
Settings\\Administrator\\Desktop\\Test_problems\\initial-output.xls");
            node s=(node)initial_node.clone();
            s.level=0;
            for (int i=0;i<MNum;i++)s.available_time[i]=tnow;
            for(int i=0;i<JNum;i++){
                  for(int j=0;j<ONum;j++){
                        int mindex = s.O[i][j][1];
                        if(mindex!=MNum){
                              if(s.O[i][j][0]>=tnow){
                                    s.O[i][j][0]=-1;
                                    s.O[i][j][1]=-1;
s.CompletionTime[i][j]=-1;
```

```
for(int k=0;k<broken_j_index.length;k++){
      if(i==broken_j_index[k]){
            s.O[i][j][0]=0;
            s.O[i][j][1]=MNum;
            s.level++;
      }
}
      }else{
      boolean broke=false;
      for(int k=0;k<broken_j_index.length;k++){
            if(i==broken_j_index[k]){
            broke=true;
            }
      }
            double end_time = s.CompletionTime[i][j];
                  if(!broke){
if(s.available_time[mindex]<end_time)s.available_time[mindex]=end_time;
}else{
if(s.available_time[mindex]<end_time){
s.available_time[mindex]=Math.min(end_time, tnow);
                                                }
                                          }
                                          s.level++;
                                    }
                              }
                        }
            }
            //set available time for broken machines
            for(int i=0;i<broken_index.length;i++){
                  int mindex= broken_index[i];
                  double rtime= repair_time[i];
                  if(s.available_time[mindex]>tnow){
                        s.available_time[mindex]+=rtime;
                        }else{
                        s.available_time[mindex]=tnow+rtime;
                  }
            }
            for (int i=JNum;i<JNum+arrival_job_size;i++){
                  for (int j=0;j<ONum;j++){
                        s.O[i][j][0]=-1;
                        s.O[i][j][1]=-1;
                        s.CompletionTime[i][j]=-1;
                  }
            }
      LinkedList<node> newlist= initialize_list(s);
      File intout=new File("C:\\Documents and
Settings\\Administrator\\Desktop\\Test_problems\\intoutinitial2.txt");
      BufferedWriter out= new BufferedWriter(new FileWriter(intout));
      Iterator<node> temp = newlist.iterator();
            int lmin=ONum*(JNum+arrival_job_size);
            while (temp.hasNext()){
                  int level = temp.next().level;
                  if (level<lmin)lmin=level;
            }
for (intlevel=lmin+1;level<=ONum*(JNum+arrival_job_size);level++){
      out.write("job\toper\tm\tst\tf\tw1\tw2\tw3\tlevel\n");
      System.out.print("job\toper\tm\tst\tf\tw1\tw2\tw3\tlevel\n");
```

```
                    System.out.println(level);
                    newlist = branch_bound(newlist);
                    temp = newlist.iterator();
                    while (temp.hasNext()){
                          out.write("\t\t\t\t\t\t\t\t\t"+level+"\n");
                    System.out.print("\t\t\t\t\t\t\t\t\t"+level+"\n");
                          out.write(temp.next().toString()+"\n");
                          out.write("----------------------------\n");
                          System.out.print("----------------------\n");
                    }
              out.write("=========================================\n");
        System.out.print("=========================================\n");
              }
              out.close();
              for(int w=0;w<BeamWidth;w++){
                    node p=(node)newlist.get(w);
                    node t=(node)initial_node;
                    double makespan=0;
                    double [][]completion_time_new = new
                    double[JNum+arrival_job_size][ONum] ;
                    double [][]completion_time_old=new
                    double[JNum][ONum];
                    double sum_comp1=0;
                    double sum_comp2=0;
                    double sum_com=0;
                    for(int j=0;j<JNum+arrival_job_size;j++){
                          for(int k=0;k<ONum;k++){
                                boolean Mbroke=false;
                                double rt=0;
                                if(p.O[j][k][1]!=MNum){
                          for (int m=0;m<broken_index.length;m++){
if((broken_index[m]==p.O[j][k][1])&(p.O[j][k][0]<tnow)&
      (ProcessTime[j*ONum+k][p.O[j][k][1]]+p.O[j][k][0]>tnow)){
                          Mbroke=true;
                          rt = repair_time[m];
}
if(Mbroke){
      completion_time_new[j][k]=p.O[j][k][0]+
      ProcessTime[j*ONum+k][p.O[j][k][1]]+rt;
              }else{
      completion_time_new[j][k]=p.O[j][k][0]+
      ProcessTime[j*ONum+k][p.O[j][k][1]];
                                          }
                                    }
              boolean jbroke=false;
              for (int h=0;h<broken_j_index.length;h++){
                    if((broken_j_index[h]==j)&(p.O[j][k][1]<MNum)){
      if((p.O[j][k][0]<tnow)&
      (ProcessTime[j*ONum+k][p.O[j][k][1]]+p.O[j][k][0]>tnow)){
                                          }
                    jbroke=true;
                                    }
                              }
if(!jbroke){
      completion_time_new[j][k]=p.O[j][k][0]+ProcessTime[j*ONum+k][p.O[
j][k][1]];
              }
```

```
            if (jbroke){
      completion_time_new[j][k]=tnow;
                                    }
if(completion_time_new[j][k]>makespan)
      makespan=completion_time_new[j][k];
                                    }
                              }
                        }
                        for(int j=0;j<JNum;j++){
                              for(int k=0;k<ONum;k++){
                                    if(t.O[j][k][1]!=MNum){
      completion_time_old[j][k]=
      t.O[j][k][0]+ProcessTime[j*ONum+k][t.O[j][k][1]];

      if(completion_time_new[j][k]>completion_time_old[j][k]){
      sum_compl+=(completion_time_new[j][k]-completion_time_old[j][k]);
      }
      if(completion_time_new[j][k]<completion_time_old[j][k]){
      sum_comp2+=(completion_time_old[j][k]-completion_time_new[j][k]);
                                    }
                              }
                        }
                        sum_com=sum_compl+sum_comp2;
                        System.out.println("makespan "+makespan);
                        System.out.println("Stability-> "+sum_com);
                        for (int i=0;i<JNum+arrival_job_size;i++){
                              for(int j=0;j<ONum;j++){
                                    boolean Mbroke=false;
                                    double rt=0;
                                    if(p.O[i][j][1]!=MNum){
                        for (int m=0;m<broken_index.length;m++){
      if((broken_index[m]==p.O[i][j][1])& (p.O[i][j][0]<tnow)&
      (ProcessTime[i*ONum+j][p.O[i][j][1]]+p.O[i][j][0]>tnow)){
            Mbroke=true;
            rt = repair_time[m];
      }
      if((p.O[i][j][0]>=tnow)||(p.O[i][j][1]==broken_index[m])){
            if(Mbroke){
            p.CompletionTime[i][j]=(int)
(p.O[i][j][0]+ProcessTime[i*ONum+j][p.O[i][j][1]]+rt);
      }else{
      p.CompletionTime[i][j]=p.O[i][j][0]+
      ProcessTime[i*ONum+j][p.O[i][j][1]];
            }
      }
}
      boolean Jbroke=false;
      for (int m=0;m<broken_j_index.length;m++){
      if((broken_j_index[m]==i)&(p.O[i][j][1]<MNum)){
if((p.O[i][j][0]<tnow)&
      (ProcessTime[i*ONum+j][p.O[i][j][1]]+p.O[i][j][0]>tnow)){
                                                      }
Jbroke=true;
            }
if((p.O[i][j][0]>=tnow)||(broken_j_index[m]==i)){
            if(!Jbroke){
```

```
                    p.CompletionTime[i][j]=p.O[i][j][0]+
                    ProcessTime[i*ONum+j][p.O[i][j][1]];
                          }
              if (Jbroke){
if(p.O[i][j][0]+ProcessTime[i*ONum+j][p.O[i][j][1]]<tnow)
p.CompletionTime[i][j]=p.O[i][j][0]+ProcessTime[i*ONum+j][p.O[i][j][1]]
;
else{
      p.CompletionTime[i][j]=tnow;
                                                      }
                                                }
                                          }
                                    }
                               }
                          }
              }
intout=new File("C:\\Documents and
Settings\\Administrator\\Desktop\\Test_problems\\initial-output.xls");
              out= new BufferedWriter(new FileWriter(intout));
              out.write("job\toper\tm\tst\tf\tw1\tw2\tw3\n");
              temp = newlist.iterator();
              while (temp.hasNext()){
                    out.write(temp.next().toStringfinal()+"\n");
                    out.write("\n");
              }
              out.write("\n\n");
              out.close();
      }

static LinkedList<node> initialize_list(node s) throws
CloneNotSupportedException{
              LinkedList<node> level1 = new LinkedList<node>();
              //generate first level
              LinkedList<opcand> candidates = new LinkedList<opcand>();
              //check the precedence of the candidates
              for (int i=0;i<JNum+arrival_job_size;i++){
                    for (int j=0;j<ONum;j++){
      if((precedence_ok(s, i, j)&
      (initial_precedence(initial_node,s, i, j)))){
                                opcand c = new opcand();
                                c.jobindex=i;
                                c.opindex=j;
                                c.start_time=0;
                                candidates.add(c);
                          }
                    }
              }
              Iterator<opcand> opiterator = candidates.iterator();
              while(opiterator.hasNext()){
                    opcand c = opiterator.next();
                    int row = (c.jobindex)*ONum+c.opindex;
                    for (int i=0;i<MNum;i++){
if(ProcessTime[row][i]>0){
      node temp = (node) s.clone();
      temp.level=s.level+1;
      if(c.opindex>0){
```

```
            boolean prev_broken=false;
            int prev_mindex = temp.O[c.jobindex][c.opindex-1][1];
            int broke_index = 0;
            for(int b=0;b<broken_index.length;b++){
      if (prev_mindex==broken_index[b]){
            prev_broken=true;
            broke_index=b;
            }
      }
      if(prev_broken){
            double rtime= repair_time[broke_index];
            if((temp.O[c.jobindex][c.opindex-1][0]<tnow)&&
            (temp.CompletionTime[c.jobindex][c.opindex-1]>tnow)){
      temp.O[c.jobindex][c.opindex][0]=
      (int)Math.max(temp.available_time[i],
      temp.CompletionTime[c.jobindex][c.opindex-1]+rtime);
      }else{
      temp.O[c.jobindex][c.opindex][0]=
      (int)Math.max(temp.available_time[i],
      temp.CompletionTime[c.jobindex][c.opindex-1]);
      }
            }else{
      temp.O[c.jobindex][c.opindex][0]=
      (int)Math.max(temp.available_time[i],
      temp.CompletionTime[c.jobindex][c.opindex-1]);
      }

      }else{
      temp.O[c.jobindex][c.opindex][0]=(int)temp.available_time[i];
      }
      temp.O[c.jobindex][c.opindex][1]=i;
      temp.available_time[i] = temp.O[c.jobindex][c.opindex][0]+
      ProcessTime[c.jobindex*ONum+c.opindex][i];

level1.add(temp);
                              }
                        }
                  }
//first level nodes have been generated (all possible machines for each
operation)
            double [] global_cost = new double[level1.size()];
            for (int i=0;i<level1.size();i++){
                  global_cost[i] = evaluate(level1.get(i));
            }
//select the beamwidth best nodes index that has minimum costs
            int [] best_nodes_index = new int[BeamWidth];
            //initilize to 0 and 1 ... (beamwidth-1)
            int max_index = 0;
            for (int i=0;i<BeamWidth;i++){
                  best_nodes_index[i]=i;
if (global_cost[i]>global_cost[max_index])max_index=i;
            }
      for (int i=BeamWidth;i<level1.size();i++){
            if(global_cost[i]<global_cost[max_index]){
                        best_nodes_index[max_index] = i;
                        //update the max_index
                        max_index = 0;
```

```
                    for (int j=1;j<BeamWidth;j++){
            if (global_cost[j]>global_cost[max_index])max_index=j;
                    }
                }
            }
            //add the best beamwidth nodes to a new list and return it
            LinkedList<node> bounded_level1 = new LinkedList<node>();
            for (int i=0;i<BeamWidth;i++){
                bounded_level1.add(level1.get(best_nodes_index[i]));
            }
            return bounded_level1;
        }

    private static node initialise_start_nodes(String file_name) throws
    FileNotFoundException {
            LinkedList<node> list = new LinkedList<node>();
            Scanner s= new Scanner(new File(file_name));
            //read a title row
            s.nextLine();
            int count=0;
            double min_fval = MaxProcessTime;
            int best_node_index=-1;
            for (int i=0;i<BeamWidth;i++){
                count++;
                node temp = new node();
                for (int j=0;j<JNum;j++){
                    for (int k=0;k<ONum;k++){
                        String l = s.nextLine();
                        String line[] = l.split("\t");

        temp.O[j][k][0]=Integer.parseInt(line[3]);
        temp.O[j][k][1]=Integer.parseInt(line[2]);
        temp.CompletionTime[j][k]=Integer.parseInt(line[4]);
        if(temp.O[j][k][1]!=MNum){
                    temp.available_time[temp.O[j][k][1]]=
                Math.max(temp.available_time[temp.O[j][k][1]],
                    temp.CompletionTime[j][k]);
                        }
                    }
                }
                temp.level=ONum*JNum;
                temp.branch=i;
                list.add(temp);
                //read extra lines (garbage)
                String line[]=s.nextLine().split("\t");
                double fval = Double.parseDouble(line[4]);
                if(fval<min_fval){
                min_fval = fval;
                best_node_index = i;
                }
                if(count!=BeamWidth){
                    s.nextLine();
                    s.nextLine();
                }
            }
            s.close();
            return list.get(best_node_index);
```

```
    }

private static void initialise_process_time_table(String filename)

throws BiffException, IOException {

            ProcessTime = new int[(JNum+arrival_job_size)*ONum][MNum];
            WorkbookSettings ws = new WorkbookSettings();
            ws.setLocale(new Locale("en", "EN"));
            Workbook workbook = Workbook.getWorkbook(new
            File(filename),ws);
            Sheet s  = workbook.getSheet(0);
      for (int i=0;i<MNum;i++){
            for (int j=0;j<(JNum+arrival_job_size)*ONum;j++){
                        ProcessTime[j][i] =
Integer.parseInt(s.getCell(i+1,j+1).getContents());
                  }
            }
            workbook.close();
      }

private static double evaluate(node n) throws

CloneNotSupportedException {

            node copy = (node) n.clone();
            //Generate new schedules from this node
            int l = copy.level;
            for (int i=0;i<(JNum+arrival_job_size)*ONum - l;i++){
                  best_jom (copy);
            }
            //evaluate the copy node
            double[] ws = new double[JNum+arrival_job_size];
            double sum=0;
            double new_makespan=0;
            double initial_makespan=0;
            for (int i=0;i<JNum+arrival_job_size;i++){
                  boolean Mbroke=false;
                  double rt=0;
                  if(copy.O[i][ONum-1][1]!=MNum){
                        for (int m=0;m<broken_index.length;m++){
      if((broken_index[m]==copy.O[i][ONum-1][1])&
      (copy.O[i][ONum-1][0]<tnow)&
      (ProcessTime[(i+1)*ONum-1][copy.O[i][ONum-1][1]]+
      copy.O[i][ONum-1][0]>tnow)){
                  Mbroke=true;
                  rt = repair_time[m];
                  }
                  if(Mbroke){
ws[i]=copy.O[i][ONum-1][0]+
ProcessTime[ONum*(i+1)-1][copy.O[i][ONum-1][1]]+rt;
                        }else{
ws[i]=copy.O[i][ONum-1][0]+
ProcessTime[ONum*(i+1)-1][copy.O[i][ONum-1][1]];
                        }
                  boolean Jbroke=false;
for (int h=0;h<broken_j_index.length;h++){
```

```
        if((broken_j_index[h]==i)&&(copy.O[i][ONum-1][1]<MNum)){
        if((copy.O[i][ONum-1][0]<tnow)&&
        (ProcessTime[ONum*(i+1)-1][copy.O[i][ONum-1][1]]+
        copy.O[i][ONum-1][0]>tnow)){
              Jbroke=true;
        }
        }
}
              if('Jbroke){
ws[i]=copy.O[i][ONum-1][0]+
ProcessTime[ONum*(i+1)-1][copy.O[i][ONum-1][1]];
              }
              if (Jbroke){
              ws[i]=tnow;
              }
              if(ws[i]>new_makespan)new_makespan=ws[i];
                          }
                    }
              }
              node eqnode = initial_node;
              double [] wss=new double [JNum+arrival_job_size];
              for (int j=0;j<JNum;j++){
                    if(eqnode.O[j][ONum-1][1]'=MNum){
                    wss[j]=
                    eqnode.O[j][ONum-1][0]+
                    ProcessTime[ONum*(j+1)-1][eqnode.O[j][ONum-1][1]];
              if(wss[j]>initial_makespan)initial_makespan=wss[j];
                    }
              }
              double F1=0;
              double F11=0;
              for(int i=0;i<JNum;i++){
                    if(ws[i]>wss[i]){
                          if(copy.O[i][ONum-1][1]'=MNum){
                                F1+=(ws[i]-wss[i])*delta[i];
                          }
                    }
                    F11=(ws[JNum]-wss[JNum])*delta[JNum];
              }
              double F_save_duedates=0;
              for(int i=0;i<JNum;i++){
                    if(ws[i]<wss[i]){
                          if(copy.O[i][ONum-1][1]'=MNum){
                                F_save_duedates+=(wss[i]-ws[i])*save[i];
                          }
                    }
              }
              double F2=0;
              double F3=0;
              for(int i=0;i<JNum;i++){
                    for(int j=0;j<ONum;j++){
                          if(copy.O[i][j][1]'=MNum){
                                if(copy.O[i][j][0]<eqnode.O[i][j][0]){
              F2+=(eqnode.O[i][j][0]-copy.O[i][j][0])*mio[i];
                          }else{
              F3+=(copy.O[i][j][0]-eqnode.O[i][j][0])*h[i];
                          }
```

```
            }
        }
}
double F4=0;
int count=0;
for(int i=0;i<JNum;i++){
        for(int j=0;j<ONum;j++){
if((copy.O[i][j][1]'=MNum)&(eqnode.O[i][j][1]'=MNum)){
        if(copy.O[i][j][1]'=eqnode.O[i][j][1]){
        count++;
                    }
            }
        F4=(count*omega);
        }
}
double F5=0;
double new_sum=0;
double Fa6=0;
double Fb6=0;
double F6=0;
double [] WL = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
double [] IWL = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int p=-1;
int q=-1;
double[] new_idle_time=new double[MNum];
double[] initial_idle_time=new double[MNum];
for(int j=0;j<JNum+arrival_job_size;j++){
        for(int k=0;k<ONum;k++){
              int row = (j)*ONum+k;
              p=copy.O[j][k][1];
              if(p'=MNum){
                    WL[p]+=ProcessTime[row][p];
                    Fa6+=ProcessTime[row][copy.O[j][k][1]];
              }
        }
}
for(int b=0;b<broken_index.length;b++){
        int m=broken_index[b];
        WL[m]+=repair_time[b];
}
for(int h=0;h<MNum;h++){
        new_idle_time[h]=new_makespan-WL[h];
}
for(int j=0;j<JNum;j++){
        for(int k=0;k<ONum;k++){
              int row = (j)*ONum+k;
              q=eqnode.O[j][k][1];
              if(q'=MNum){
                    IWL[q]+=ProcessTime[row][q];
                    Fb6+=ProcessTime[row][eqnode.O[j][k][1]];
              }
        }
}
for(int h=0;h<MNum;h++){
        initial_idle_time[h]=initial_makespan-IWL[h];
}
for (int h=0;h<MNum;h++){
```

```
                    if(new_idle_time[h]>initial_idle_time[h]){
                    F5+=(new_idle_time[h]-initial_idle_time[h])*eta[h];
                    }
              }
              if (Fa6>Fb6)F6=tilda*(Fa6-Fb6);
sum=F1+F2+F3+F4+F5+F6+F11;
new_sum=sum-F_save_duedates;
System.out.println("F1->"+F1+", F1'->"+F11+" ,F2->"+F2+" ,F3->"+F3+"
,F4->"+F4+" ,F5->"+F5+" ,F6->"+F6);
System.out.println("save due dates- >" + F_save_duedates+ "updated
Cost- >"+ new_sum);
              return sum;
              }

private static void best_jom(node n) {
              int jindex=-1;
              int oindex=-1;
              int mindex=-1;
              int min_spt=MaxProcessTime;
              for (int j=0;j<JNum+arrival_job_size;j++){
                    for (int k=0;k<ONum;k++){
                          if(precedence_ok(n, j, k)
                                for(int m=0;m<MNum;m++){
if((ProcessTime[j*ONum+k][m]>0)&&(ProcessTime[j*ONum+k][m]<min_spt)){
                                                mindex=m;
                                                jindex=j;
                                                oindex=k;
min_spt=ProcessTime[j*ONum+k][m];
                                }
                                else
if((ProcessTime[j*ONum+k][m]>0)&&(ProcessTime[j*ONum+k][m]==min_spt)){
                                                int load1=0;
                                                int load2=0;
for(int p=0;p<JNum+arrival_job_size;p++){
      for(int l=0;l<ONum;l++){
      int row = (p)*ONum+l;
      if(n.O[p][l][1]==mindex){
      load1+=ProcessTime[row][mindex];
      }
      else if (n.O[p][l][1]==m){
      load2+=ProcessTime[row][m];
            }
                  }
                        }
                                                if(load1<load2){
                                                      mindex=m;
                                                      jindex=j;
                                                      oindex=k;

      min_spt=ProcessTime[j*ONum+k][m];
                                                }
                                          }

                                    }
                              }
                        }
              }
```

```
            if(oindex>0){
                  boolean prev_broken=false;
                  int prev_mindex = n.O[jindex][oindex-1][1];
                  int broke_index = 0;
                  for(int i=0;i<broken_index.length;i++){
                        if (prev_mindex==broken_index[i]){
                              prev_broken=true;
                              broke_index=i;
                        }
                  }
                  if(prev_broken){
                        double rtime= repair_time[broke_index];
if((n.O[jindex][oindex-1][0]<tnow)&&
(n.CompletionTime[jindex][oindex-1]>tnow)){
n.O[jindex][oindex][0]=(int)Math.max(n.available_time[mindex],
n.CompletionTime[jindex][oindex-1]+rtime);
                        }else{
n.O[jindex][oindex][0]=(int)Math.max(n.available_time[mindex],
n.CompletionTime[jindex][oindex-1]);
                        }
                        }else{
n.O[jindex][oindex][0]=(int)Math.max(n.available_time[mindex],
(n.CompletionTime[jindex][oindex-1]));
                  }
            }else if(oindex==0){

n.O[jindex][oindex][0]=(int)n.available_time[mindex];
            }
n.O[jindex][oindex][1]=mindex;
n.available_time[mindex] =
n.O[jindex][oindex][0]+ProcessTime[jindex*ONum+oindex][mindex];
            n.level++;
            }

static boolean precedence_ok(node current, int i, int j){
            if (current.O[i][j][0]>=0)return false;
            if (j==0)return true;
            if (current.O[i][j-1][0]>=0)return true;
            return false;
      }

static LinkedList<node> branch_bound(LinkedList<node> currentlist)
throws CloneNotSupportedException{
            LinkedList<node> children =new LinkedList<node>();
            Iterator<node> iterator = currentlist.iterator();

            while (iterator.hasNext()) {
                  node current = iterator.next();

                  if(current.level>=(JNum+arrival_job_size)*ONum){
                        children.add(current);
                  }else{
                        children.add(branch_bound(current));
                  }

            }
            return children;
```

```
    }

private static node branch_bound(node current) throws
CloneNotSupportedException {
            LinkedList<node> next_level = new LinkedList<node>();
            //generate next level
            LinkedList<opcand> candidates = new LinkedList<opcand>();
            //check the precedence of the candidates
            double min_start_time=MaxStartTime;
            for (int i=0;i<JNum+arrival_job_size;i++){
                  for (int j=0;j<ONum;j++){
if(precedence_ok(current, i, j)&
(initial_precedence(initial_node,current, i, j))){
                              opcand c = new opcand();
                              c.jobindex=i;
                              c.opindex=j;
      if(j>0){
            boolean prev_broken=false;
            int prev_mindex = current.O[i][j-1][1];
            int broke_index = 0;
            for(int b=0;b<broken_index.length;b++){
            if (prev_mindex==broken_index[b]){
                        prev_broken=true;
                        broke_index=b;
                                    }
                              }
                        if(prev_broken){
                        double rtime= repair_time[broke_index];
      if((current.O[i][j-1][0]<tnow)&&
(current.O[i][j-1][0]+ProcessTime[i*ONum+j-1][prev_mindex]>tnow)){
c.start_time=(int)(current.O[i][j-1][0]+
ProcessTime[i*ONum+j-1][prev_mindex]+rtime);
      }else{

      c.start_time=(int)current.O[i][j-1][0]+
      ProcessTime[i*ONum+j-1][prev_mindex];
                  }
                  }else{
      c.start_time = current.O[i][j-1][0]+
      ProcessTime[i*ONum+j-1][current.O[i][j-1][1]];
                  }
                  }else{
                  c.start_time=0;
                        }
      if(c.start_time<min_start_time)min_start_time=c.start_time;
                              candidates.add(c);
                        }
                  }
            }
      LinkedList<opcand> candidates_with_min_st = new
      LinkedList<opcand>();
            Iterator<opcand> opiterator = candidates.iterator();
            while(opiterator.hasNext()){
                  opcand c = opiterator.next();
                  if (c.start_time==min_start_time){
                        candidates_with_min_st.add(c);
                  }
```

```
            }
            //candidates with minimum starting time are in the list
      LinkedList<opcand> can_list_machine_assigned = new
      LinkedList<opcand>();
            opIterator = candidates_with_min_st.iterator();
            while(opIterator.hasNext()){
                  opcand c = opIterator.next();
                  int row = (c.jobIndex)*ONum+c.opIndex;
                  for (int i=0;i<MNum;i++){
                        if(ProcessTime[row][i]>0){
                              opcand c2 = new opcand();
                              c2.jobIndex = c.jobIndex;
                              c2.opIndex = c.opIndex;
                              c2.start_time = c.start_time;
                              c2.machine = i;
                              can_list_machine_assigned.add(c2);
                              }
                  }
            }

            candidates_with_min_st = null;
            //filtering
            double[] pt = new double[can_list_machine_assigned.size()];

            for (int i=0;i<can_list_machine_assigned.size();i++){
                  opcand c = can_list_machine_assigned.get(i);
                  //c.machine[0] is greater than -1 always
pt[i] = ProcessTime[c.jobIndex*ONum+c.opIndex][c.machine];
            }
for (int i=0;i<can_list_machine_assigned.size()-FilterWidth;i++){
                  int index=0;
                  for (int j=1;j<pt.length;j++){
                        if(pt[i]>pt[index])index=i;
                  }
                  pt[index]=-1;
            }
            LinkedList<opcand> filtered_list = new
            LinkedList<opcand>();
            for (int i=0;i<can_list_machine_assigned.size();i++){
                  if(pt[i]>0){
                        opcand c = can_list_machine_assigned.get(i);
                        filtered_list.add(c);
                  }
            }
            //end filtering
            //create child nodes
            opIterator = filtered_list.iterator();
            while(opIterator.hasNext()){
                  opcand c = opIterator.next();
                  int row = (c.jobIndex)*ONum+c.opIndex;
                  //generate the children and add it to level1 list
                  node temp = (node) current.clone();
                  temp.level++;
temp.O[c.jobIndex][c.opIndex][0]=Math.max(c.start_time,
(int)current.available_time[c.machine]);//set start time
temp.O[c.jobIndex][c.opIndex][1]=c.machine;//set machine
                  //update availability time of machines
```

```
		temp.available_time[c.machine]=
		temp.O[c.jobindex][c.opindex][0]+ProcessTime[row][c.machine];
				next_level.add(temp);
			}
			//next level is ready to bound
			double [] global_cost = new double[next_level.size()];
			for (int i=0;i<next_level.size();i++){
				global_cost[i] = evaluate(next_level.get(i));
			}
		//select the beamwidth best nodes index that has minimum costs
			int best_index = 0;
			for (int i=1;i<next_level.size();i++){
				if(global_cost[i]<global_cost[best_index]){
					best_index = i;
				}
			}
			return		next_level.get(best_index);
		}

	static class opcand {
			int jobindex,opindex;
			int start_time;
			int machine;
		}

	static class node implements Cloneable {
			int[][] CompletionTime=new
	int[JNum+arrival_job_size][ONum];
			int level;
			int branch=-1;
			double[] available_time = new double[MNum];
			public Object clone() throws CloneNotSupportedException {
				node copy = new node();
				for (int i =0;i<JNum+arrival_job_size;i++){
					for (int j=0;j<ONum;j++){
						copy.O[i][j][0]=this.O[i][j][0];
		//start time of this operation
						copy.O[i][j][1]=this.O[i][j][1];
		//machine assigned to this operation

		copy.CompletionTime[i][j]=this.CompletionTime[i][j];
					}
				}
				for (int i=0;i<MNum;i++){
					copy.available_time[i]=this.available_time[i];
				}
				copy.level = this.level;
				copy.branch=this.branch;
				return copy;
			}
			public String toString(){
			String s = new String("");
			for (int i =0;i<JNum+arrival_job_size;i++){
				for (int j=0;j<ONum;j++){
					s =
	s.concat(""+i+"\t"+j+"\t"+this.O[i][j][1]+"\t"+this.O[i][j][0]+"\n");
				}
```

```
            }
            try {
                        s = s.concat("\t\t\t\t"+evaluate(this)+"\n");
                  } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                  }
            return s;
      }
            public String toStringfinal(){
            String s = new String("");
            for (int i =0;i<JNum+arrival_job_size;i++){
                  for (int j=0;j<ONum;j++){
                        s =
s.concat(""+i+"\t"+j+"\t"+this.O[i][j][1]+"\t"+this.O[i][j][0]+"\t"+Com
pletionTime[i][j]+"\n");

                  }
            }
            try {
                        s = s.concat("\t\t\t\t"+evaluate(this)+"\n");
                  } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                  }
            return s;
      }

      int [][][] O = new int[JNum+arrival_job_size][ONum][2];
      }
}
```

Appendix B: JAVA model for MFBSR in case of Disruptions (MB-JC-OA)

```
import java.io.BufferedWriter;

public class mainclass {

/Input Data  /


public static void main(String[]args) throws

CloneNotSupportedException, BiffException, IOException{


}


static LinkedList<node> initialize_list(node s) throws

CloneNotSupportedException{


}


private static node initialise_start_nodes(String file_name) throws

FileNotFoundException {


}


private static void initialise_process_time_table(String filename)

throws BiffException, IOException {


}

private static double evaluate(node n) throws

CloneNotSupportedException {


}

private static void best_job(node n) {
            int jindex=-1;
```

```
            int oindex=-1;
            int mindex=-1;
            int min_spt=MaxProcessTime;
            for (int j=0;j<JNum+arrival_job_size;j++){
                  for (int k=0;k<ONum;k++){
                        if(precedence_ok(n, j, k)&
                        (initial_precedence(initial_node,n, j, k))){
                              for(int m=0;m<MNum;m++){
if((ProcessTime[j*ONum+k][m]>0)&&(ProcessTime[j*ONum+k][m]<min_spt)){
                                                mindex=m;
                                                jindex=j;
                                                oindex=k;
min_spt=ProcessTime[j*ONum+k][m];
                              }
                              else
if((ProcessTime[j*ONum+k][m]>0)&&(ProcessTime[j*ONum+k][m]==min_spt)){
                                                int load1=0;
                                                int load2=0;
for(int p=0;p<JNum+arrival_job_size;p++){
      for(int l=0;l<ONum;l++){
      int row = (p)*ONum+l;
      if(n.O[p][l][1]==mindex){
      load1+=ProcessTime[row][mindex];
      }
      else if (n.O[p][l][1]==m){
      load2+=ProcessTime[row][m];
            }
                  }
                        }
                                                if(load1<load2){
                                                      mindex=m;
                                                      jindex=j;
                                                      oindex=k;

      min_spt=ProcessTime[j*ONum+k][m];
                                                }
                                          }

                                    }
                        }
                  }
            }
            if(oindex>0){
                  boolean prev_broken=false;
                  int prev_mindex = n.O[jindex][oindex-1][1];
                  int broke_index = 0;
                  for(int i=0;i<broken_index.length;i++){
                        if (prev_mindex==broken_index[i]){
                              prev_broken=true;
                              broke_index=i;
                        }
                  }
                  if(prev_broken){
                        double rtime= repair_time[broke_index];
if((n.O[jindex][oindex-1][0]<tnow)&&
(n.CompletionTime[jindex][oindex-1]>tnow)){
n.O[jindex][oindex][0]=(int)Math.max(n.available_time[mindex],
```

```
n.CompletionTime[jindex][oindex-1]+rtime);
                        }else{
n.O[jindex][oindex][0]=(int)Math.max(n.available_time[mindex],
n.CompletionTime[jindex][oindex-1]);
                        }
                        }else{
n.O[jindex][oindex][0]=(int)Math.max(n.available_time[mindex],
(n.CompletionTime[jindex][oindex-1]));
                  }
            }else if(oindex==0){

n.O[jindex][oindex][0]=(int)n.available_time[mindex];
            }
n.O[jindex][oindex][1]=mindex;
n.available_time[mindex] =
n.O[jindex][oindex][0]+ProcessTime[jindex*ONum+oindex][mindex];
            n.level++;
            }

static boolean precedence_ok(node current, int i, int j){



}

static boolean initial_precedence(node initial, node current, int i,
int j){
            for (int p=0;p<JNum;p++){
                  for(int q=0;q<ONum;q++){
if((initial.O[p][q][0]<initial.O[i][j][0])&(current.O[p][q][1]==-1)){
                              return false;
                        }
                  }
            }
            return true;
      }

static LinkedList<node> branch_bound(LinkedList<node> currentlist)

throws CloneNotSupportedException{



}

private static node branch_bound(node current) throws
CloneNotSupportedException {
            LinkedList<node> next_level = new LinkedList<node>();
            //generate next level
            LinkedList<opcand> candidates = new LinkedList<opcand>();
            //check the precedence of the candidates
            double min_start_time=MaxStartTime;
            for (int i=0;i<JNum+arrival_job_size;i++){
                  for (int j=0;j<ONum;j++){
                        if((precedence_ok(current, i, j)&
            initial_precedence(initial_node, current, i, j))){
                              opcand c = new opcand();
```

```
                              c.jobindex=i;
                              c.opindex=j;
                        if(j>0){
      c.start_time = current.O[i][j-1][0]+
       ProcessTime[i*ONum+j-1][current.O[i][j-1][1]];
                        }else{
                              c.start_time=0;
                              }

      if(c.start_time<min_start_time)min_start_time=c.start_time;
                              candidates.add(c);

                        }
                  }
            }

LinkedList<opcand> candidates_with_min_st = new LinkedList<opcand>();
            Iterator<opcand> opiterator = candidates.iterator();
            while(opiterator.hasNext()){
                  opcand c = opiterator.next();
                  if (c.start_time==min_start_time){
                        candidates_with_min_st.add(c);

                  }
            }
            //candidates with minimum starting time are in the list
            LinkedList<opcand> can_list_machine_assigned = new
LinkedList<opcand>();
            opiterator = candidates_with_min_st.iterator();
            while(opiterator.hasNext()){
                  opcand c = opiterator.next();

                  int y =c.jobindex;
                  int i=-1;
                  int row = (c.jobindex)*ONum+c.opindex;

                  if(y<JNum){
                        i=current.O[c.jobindex][c.opindex][1];
                  }
                  if(y>=JNum){
                        int min_spt=10000;
                        for (int h=0;h<MNum;h++){
                              if(ProcessTime[row][h]>0){
                                    if(ProcessTime[row][h]<min_spt){
                                          min_spt=ProcessTime[row][h];
                                          i=h;
                                    }
                              }
                        }

                  }
                              opcand c2 = new opcand();
                              c2.jobindex = c.jobindex;
                              c2.opindex = c.opindex;
                              c2.start_time = c.start_time;
                              c2.machine = i;
                              can_list_machine_assigned.add(c2);
```

```
                        }

            candidates_with_min_st = null;

//filtering Process
{
...}
// end of filtering
opiterator = can_list_machine_assigned.iterator();
            while(opiterator.hasNext()){
                  opcand c = opiterator.next();
                  int row = (c.jobindex)*ONum+c.opindex;
                  //generate the children and add it to level1 list
                  node temp = (node) current.clone();
                  temp.level++;

      temp.O[c.jobindex][c.opindex][0]=Math.max(c.start_time,
(int)current.available_time[c.machine]);//set start time
                  temp.O[c.jobindex][c.opindex][1]=c.machine;//set
machine
                  //update availability time of machines
temp.available_time[c.machine]=
temp.O[c.jobindex][c.opindex][0]+ProcessTime[row][c.machine];
next_level.add(temp);
                  //System.out.println(temp.temptoString());

            }
            //next level is ready to bound
double [] global_cost = new double[next_level.size()];
for (int i=0;i<next_level.size();i++){
global_cost[i] = evaluate(next_level.get(i));
                  //System.out.println(global_cost[i]);
            }
            //select the beamwidth best nodes index that has minimum
costs
            int best_index = 0;
            for (int i=1;i<next_level.size();i++){
                  if(global_cost[i]<global_cost[best_index]){
                        best_index = i;
                  }
            }

            return      next_level.get(best_index);

}

static class opcand {

...

}

static class node implements Cloneable
                                    ,

...

}
```