CORE
Metadata, citation and similar papers at core.ac.uk
Provided by SHAREOK repository

UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

ASSEMBLY JOB SHOP SCHEDULING PROBLEMS WITH COMPONENT AVAILABILITY CONSTRAINTS

A DISSERTATION

SUBMITTED TO GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

By

KAI-PEI CHEN
Norman, Oklahoma
2007

UMI Number: 3291943

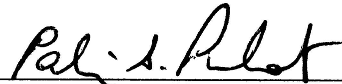
UMI Microform 3291943
Copyright 2008 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

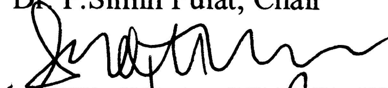
ASSEMBLY JOB SHOP SCHEDULING PROBLEMS WITH COMPONENT AVAILABILITY CONSTRAINTS

A DISSERTATION APPROVED FOR THE
SCHOOL OF INDUSTRIAL ENGINEERING

BY

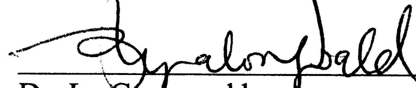


Dr. P.Simin Pulat, Chair

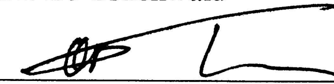


Dr. Scott A. Moses, Co-Chair

Dr. F. Hank Grant



Dr. Le Gruenwald



Dr. Theodore B. Trafalis

© Copyright by KAI-PEI CHEN 2007
All Rights Reserved.

Acknowledgments

I would like to thank the many people who have supported me to complete this dissertation. First, I thank my advisors, Dr. Pulat and Dr. Moses, for their supervision, guidance and friendly attitude throughout the process of this research. I would also like to express my appreciation to other committee members, Dr. Grant , Dr. Gruenwald and Dr. Trafalis, for their review of this work and helpful comments. Finally, I would like to thank my parents, my wife, Hui-Ling, and my beautiful daughter, Rou-An, for their patience, understanding and constant love during the progress of this dissertation. This dissertation would not have been possible without their sacrifice.

Table of Contents

List of Figures viii
List of Tables vii
Abstract ix
Chapter 1. Introduction 1
1.1 Research Motivation 1
1.2 Research Overview 5
1.3 Research Summary 9
1.4 Contribution 14
1.5 Organization 15
Chapter 2. Literature Review 17
2.1 Classical Job Shop Scheduling 20
2.2 Assembly Job Shop Scheduling 25
2.3 Assembly Job Shop Scheduling with component availability constraints 29
2.4 Shifting bottleneck procedures 34
2.4.1 The Generic Shifting Bottleneck Procedure 35
2.4.2 The Shifting Bottleneck Procedure with parallel machines 39
2.4.3 Efficient Shifting Bottleneck heuristic for parallel machines (ESBPM) .. 40
Chapter 3. Methodology 42
3.1 Problem description and assumptions 45
3.2 The extended disjunctive graph and mathematical formulation 47
3.3 Optimization Approach: 53
3.4 Heuristic approach 57
3.4.1 Modified shifting bottleneck procedure (MSBP) 58
3.4.2 Efficient shifting bottleneck procedure (ESBP) 67
3.4.3 Efficient shifting bottleneck procedure + Rolling horizon procedure (RHP) 69
3.5 Dispatching rules 72
Chapter 4. Computational Results 77
4.1 Design of Experiments 78
4.2 Performance Measures and Computational Results 80
4.3 Computational results for optimization methods 82
4.4 Computational results for dispatching rules: 86
4.5 Computational results for heuristic approach 1 – Modified shifting bottleneck procedure (MSBP) : 89
4.6 Computational results for heuristic approach 2 – Efficient shifting bottleneck procedure (ESBP) : 99
4.7 Computational results: ESBP vs. MSBP vs. Dispatching rules 112
4.8 Computational results for RHP : 115
4.9 Recommendations for choosing appropriated methods: 118
Chapter 5. Future research directions 119
5.1 Dynamic assembly job shop scheduling problems with components availability constraints 119
5.2 The effect of different approaches on the overall performance of a production system 122
5.3 Other future research and directions 123

Chapter 6. Conclusions 125
Reference 127
Appendix A. Example of Input data format – 5 jobs, 5 machines and 5 components... 133
Appendix B. Source Code of Xpress-MP Model – MIP 135
Appendix C. Source Code of Xpress-MP Model – MSBP 141
Appendix D. Source Code of Xpress-MP Model – ESBP 180

List of Tables

Table 2.1 Classification for relevant studies: 19
Table 3.1 The summary of shifting bottleneck procedures for assembly job shop scheduling problem 44
Table 3.2 Notation of the assembly job shop scheduling problem 52
Table 3.3 Notation of the assembly job shop scheduling problem 54
Table 3.4 The description of MSBP methods 62
Table 3.5 The description of ESBP methods 69
Table 3.6 The summary of dispatching rules 76
Table 4.1 Experimental design for randomly generated test problems 80
Table 4.2 Experimental design for randomly generated test problems – Optimization methods 83
Table 4.3 Computational results for optimization methods 85
Table 4.4 Experimental design for randomly generated test problems – Dispatching rules 87
Table 4.5 Computational results for dispatching rules 87
Table 4.6 Experimental design for randomly generated test problems – MSBP 90
Table 4.7 The summary of SB3 methods and dispatching rule used in this experiments 92
Table 4.8 Computational results for SB3 methods 93
Table 4.9 The summary of SB4 methods and dispatching rule used in this experiments 95
Table 4.10 Computational results for SB4 methods 96
Table 4.11 Computational results for MSBP methods 98
Table 4.12 Experimental design for randomly generated test problems - ESBP 100
Table 4.13 The summary of ESBP1 methods used in this experiments 102
Table 4.14 Computational results for ESBP1 methods 103
Table 4.15 The summary of ESBP2 methods used in this experiments 105
Table 4.16 Computational results for ESBP2 methods 106
Table 4.17 The description of ESBP3 methods used in this experiments 108
Table 4.18 Computational results for ESBP3 methods 109
Table 4.19 Computational results for ESBP methods 111
Table 4.20 Computational results for ESBP vs. MSBP 114
Table 4.21 Experimental design for randomly generated test problems –RHP 116
Table 4.22 Computational results for RHP – CPU(H) 117
Table 4.23 Computational results for RHP – TWT(H) 117
Table 5.1 Notation of dynamic assembly job shop scheduling problems 121

List of Figures

Figure 1.1 Example of different product structures 11
Figure 2.1 Example of the disjunctive graph G 36
Figure 3.1 Example for product structures of assembly job 46
Figure 3.2 A example of a disjunctive graph for assembly job shop scheduling with component availability constraints 49
Figure 3.3 Figure 3.3 Mathematical formulation for assembly job shop scheduling problems 54
Figure 4.1 CPU(H) and TWT(H) for optimization methods 86
Figure 4.2 TWT(H) – Mean and SD for dispatching rules 89
Figure 4.3 CPU(H) and TWT(H) for MSBP methods 99
Figure 4.4 CPU(H) and TWT(H) for ESBP methods 112
Figure 4.5 CPU(H) and TWT(H) for MSBP vs. ESBP 115
Figure 4.6 Decision tree to choose appropriated methods 118

Abstract

Job shop scheduling has been widely studied for several decades. In generalized of the job shop scheduling problem, n jobs are to be processed on m machines under specific routings and due dates. The majority of job shop scheduling research concentrates on manufacturing environments processing string-type jobs with a linear routing where no assembly operations are involved. However, many manufacturing environments produce complex products with multi-level assembly job structures and cannot be scheduled efficiently with existing job shop scheduling techniques. Little research has been done in the area of assembly job shop scheduling, and we are not aware any of those studies consider on the availability of purchased components and the impact of component availability on the performance of assembly job shops. This research focuses on scheduling job shops that process jobs requiring multiple-levels of assembly and it also considers the availability of components that are procured from outside suppliers. By considering material constraints during production scheduling, manufacturers can increase resource utilization and improve due date performance.

To represent assembly job shop scheduling problems with component availability constraints, a modified disjunctive graph formulation is developed in this research. A mixed-integer programming model with the objective of minimizing the total weighted-tardiness is also developed in this research. Several heuristic methods, described as modified shifting bottleneck procedure (MSBP), efficient shifting bottleneck procedure (ESBP) and rolling horizon procedure (RHP), are proposed to reduce the computational time required for assembly job shop scheduling problems. These methods are extended from the shifting bottleneck procedure. The performance of various flavors of the MSBP

and ESBP is demonstrated on a set of test instances and compared with different dispatching rules that are widely used in practice. Results show that MSBP and ESBP outperform the dispatching rules by 18% to 16% on average.

This dissertation not only studies the assembly job shop scheduling problem with component availability constraints, but also demonstrates how the decomposition methodology can reduce the complexity of NP-hard problems. Based on the relative preference of solution quality and computational time, recommendations for appropriate methods to solve assembly job shop scheduling problems with different problem sizes are given in the conclusions of this dissertation.

Chapter 1. Introduction

Globally competition is getting intensive in past few decades. To compete with other companies globally, manufacturers not only need to quote shorter lead time to customers, but also keep delivery promises to them. To allocate limited resources efficiently, production scheduling play an important role in manufacturing environments. Thus, job shop scheduling has been one of most complicated production scheduling problems being studied for researchers. This dissertation focuses on the assembly job shop scheduling problem which is a special case of the job shop scheduling problem. In an assembly job-shop, a job is not only spending time on machines queue, but is also waiting for assemblies, sub-assemblies and components which are necessary to form a final product. Unlike other literatures in assembly job shop scheduling, we assume that components can not be manufactured in-house and have to be procured from vendor outside. This makes the assembly job shop scheduling problem with component availability constraints a problem that can not be resolved by existing job shop scheduling techniques. The research motivation is described in the next section and follows by research overview summarizing the related literatures in the past. Then, the summary of methodologies used in this dissertation is outlined in the next section. Finally, the contributions and organization of this dissertation is illustrated in the end of chapter.

1.1 Research Motivation

Most of the past literatures in job shops scheduling problems concentrated on shop models processing string-type jobs (Adam et al., 1987) with a linear routing where no

assembly operations are performed. However, in the real world, jobs in a typical manufacturing environment often require assembly operations which parallel components that have to be processed and assembled together through the shop. It is common that many manufacturing environments are assembly systems, such as auto and electronics companies. Scheduling with string-type jobs, which involves only one type of job delay, is much simpler than scheduling with assembly-type jobs because of different degrees of job delays. There are some unique problems that only happen when scheduling with assembly-type jobs. In job shop scheduling with string type jobs, job delays are primarily due to capacity constraints, such as the lack of available resources which are necessary to perform operations. However, scheduling in assembly job shops requires not only consideration of capacity constraints but also assembly (precedence) constraints which an item may need to wait for its parallel components before performing assembly operations. If one of its parallel components is not available, the assembly operation will not perform until all components become available, even resources (machines) are ready at that time. In other words, assembly, sub-assembly and components belonging to the same job are extremely dependent on each other compared to the string-type job. Hence, job delays in an assembly job shop may occur due to the lack of production capacity or the lack of components which are necessary to form an assembly or sub-assembly. It makes the assembly job shop scheduling problem becomes the most difficult problem in job shop scheduling.

Moreover, in some industries, material constraints are more intensive than capacity constraints. According to Karmarkar and Ramakrishnan (1996), *Computerworld* (March 29, 1993) reported that IBM's $ 1 billion backlog for its ThinkPad 700 line of

notebook computers can be partly attributed to the shortage of, literally, “a nickel part”. This report shows what the severe damages due to the material shortages can do to a multi-billion dollars company. To avoid the unexpected delays on shipments, manufacturers must keep enough materials on hand to meet the schedules for final products. Hence, companies can maintain a substantial level of inventory to avoid the late delivery due to material shortage. However, inventories are the tangible assets of a company and holding excess inventory will lead to a low return on investment (ROI). Holding a substantial level of inventory to relax the material constraints is the worst solution for manufacturers. Additionally, the availability of materials is usually fixed in short-range production plans due to long vendor lead times. Thus, material shortage may occur easily once the demand increases rapidly. How to provide a best level of customer service at minimum inventory costs becomes an essential problem for manufacturers. To accomplish these tasks successfully, manufacturers need to consider the availability of materials during scheduling. Musselman and Uzsoy (2001) state that it is an important advantage that material and capacity are considered simultaneously as elements that may constrain production to ensure that the material plan is in agreement with capacity schedule to avoid the potential violations.

Over the past decades, cost-effective production continues to play a critical role for manufacturers to survive in global competition. Companies have to quote shorter lead times (delivery times) to customers and fulfill these orders as promised. Failure to meet the customer due date may result in the loss of future business. In practice, how to classify high margin orders and satisfy these customers becomes important to the growth of a company because these customers can bring more profit to manufacturers than others

in the same resource utilization. How to allocate available resources to production activities efficiently becomes an important issue to a production system. These resource allocation problems can be addressed as production scheduling problems and the job shop scheduling problem is one of the most complicated problems in production scheduling. Due to the lack of computational power in past decades, the conventional resource allocation methods aggregate the requirements into different appropriate buckets to reduce the computational time. Although the bucket approach reduces the complexity of the problem, it also sacrifices accuracy. In addition, these methods sometimes ignore order priority and use a fixed lead time to account for the capacity constraints. These methods generate inaccurate required times and could cause the delay of high priority orders. To improve the accuracy of conventional resource allocation methods, most research on the resource allocation problem formulates production problems as a mixed integer programming model and combines with relaxation techniques to reduce the computational effort. But these approaches are not scalable and hence, are not efficient in a production environment. Additionally, capacity requirements are still approximated in these relaxation models.

It is interesting to study the scheduling problems to satisfy both the capacity schedules and material plans in an assembly job shop at the same time. It is also important to distinguish different classes of customers in order to satisfy high margin orders for future business. By doing so, manufacturers can attain the better due date performance and have higher margin in the same resource utilization. However, assembly job shop scheduling problems without component availability constraints have been known as NP-hard problems. To capable of solving large scale problems in practice, it

would be a challenging to develop efficient algorithms that can not only find feasible solutions, but also obtain acceptable solutions which are close to optima in a reasonable amount of time.

1.2 Research Overview

Production scheduling problems have been studied widely for past decades. In a broader view, the definition of production scheduling is the assignment of scarce production resources to competing activities over time in order to optimize certain performance criteria (Musselman and Uzsoy, 2001). There are three well-studied shop models in production scheduling, which are flow shop, job shop and open shop problems. In open shop problems, operations of a job can be performed in any order through the shop. In job shop problems, operations have to be processed in specific orders through the shop as stated on a predetermined routings. The flow shop problem is a special case of job shop problems because each job follows the same route through the shop. In this dissertation, we are only interested in job shop scheduling problems which is one of the most difficult combinatorial optimization problems and widely studied production scheduling problem.

In a job-shop, jobs must be processed on machines in a specified order to meet the due date. When there are only one or two machine types with one machine of each machine type, the makespan can be minimized using Johnson's algorithm (Johnson, 1954). However, a job shop with ten jobs and ten machines cannot usually be scheduled optimally (Adams et al., 1988). However, when there are more than two machines, the problem becomes strongly NP-hard.

Thus, attempts at developing optimization algorithms have focused on the variations of the branch-and-bound approach. While these methods assure optimality, they are impractical because of intensive computational requirements for large problems. Thus, much effort has been given to develop heuristics that can find solutions that are close to optimal in a reasonable amount of time. Dispatching rules are the most commonly used techniques in job shop scheduling in practice. They have fast computational times and are easy to implement, but they may perform poorly in the long run. Thus, heuristics have been researched that can find improved solutions at a cost of increased computational time, such as the shifting bottleneck procedure (Adams et al. 1988).

Although job shop scheduling is one of the most common problems in production systems, it is very complex and a strong NP-hard problem. To avoid the intensive computational time to solve large instances, it is reasonable to divide the original problem into several small subproblems which can be solved efficiently and still be able to achieve good solutions. The idea of shifting bottleneck procedure (Adams et al., 1988) is to breakdown the job-shop scheduling problem into numerous single-machine subproblems which can be solved sequentially based on a subproblem criticality and use a disjunctive graph to present operations precedence constraints and reflect machine sequence decisions. Adams et al. (1988) showed that, compared to the best priority dispatching rules, the shifting bottleneck procedure determined much lower and often optimal makespans. They solved the famous 10 job-10 machine problem of Muth and Thompson (1963) to optimality using the shifting bottleneck procedure with enumeration (SBII method) in 320 seconds. Adams et al. (1988) also used dispatching rules, the

shifting bottleneck procedure without enumeration (SBI), and SBII to solve 40 other test problems. In 38 of the problems, SBI found solutions superior to the best solutions found by dispatching rules, while the running time was about the same. In addition, SBII usually found solutions that were considerably better than solutions found by SBI.

Most of the past research concentrated on job shop scheduling problems with string-type jobs to simplify the problems as discussed above, even assembly operations are very common in manufacturing environments today. In an assembly job shop, there are two types of job delays to be considered during scheduling. One is caused by the availability of machine capacity and another one is due to the lack of assembly or sub-assembly items which are needed in parallel before starting assembly operations. It is undoubted that the assembly job shop scheduling problem is one of the most difficult production scheduling problems. Most literatures tends to use dispatching rules to solve the assembly job shop scheduling problems. Some of them also studied the impact of different job structures on the overall performance in an assembly job shop. Maxwell and Mehea (1968) test the performance of several dispatching rules and concluded that use the composite priority rules can achieve the best performance in assembly job shop. Russell and Taylor (1985) found that the dispatching rules, which produce good solutions in simple job shops, are not appropriate for an assembly job shop. They also concluded that the job structures do affect the performance of dispatching rules significantly. Fry et. al. (1989) evaluated the performance of numerous dispatching rules and concluded that due-date oriented dispatching rules perform better than other dispatching rules. They also studied the effect of stage delays on the performance due to different job structures. They figured that taller-structured BOM (Bill-of-Materials) produces more tardy jobs than

shorter-structured BOM because taller-structured BOM has more chances to create stage delays. There are some new dispatching rules developed by researches to work on assembly job shop in past years. Philipoom et al. (1991) introduced a new set of sequencing rules which incorporate important ratio (IR) based on the remaining number of operations on a particular branch to the one on the longest path to job completion. Reeja and Rajendran (2000) created a new dispatching rules based on "operation synchronization rules".

Dispatching rules had been studied intensively in assembly job shop scheduling problems. Only limited number of researches uses different methodologies, such as mathematical formulations, to solve assembly job shop scheduling problems. Park and Kim (1999) created a non-linear mathematical model for an assembly job shop scheduling problems under due date constraints. Park and Kim (2000) extended their previous research and created a mixed-integer programming model with lagrangian relaxation method to solve the same problem. Pathumnkul and Egbelu (2006) also studied the scheduling problems with objective of minimizing the weighed earliness penalty in an assembly job shop. A heuristics algorithm was created to reduce the complexity of the problems.

Although some articles on assembly job shop scheduling problems have been published, we are not aware of any literature related to assembly job shop scheduling problems with component availability. In an assembly job-shop, the operations of a job are performed both in series and parallel as defined by precedence relations. The operations are not only waiting for resources but also waiting for parallel components which belonging to the same assembly or sub-assembly. However, most of researches

assumed components required to make a product always arrive before the release of jobs or there are unlimited quantities of components on hand. In other words, materials are always available at the start of the first operation of a job and there are no material constraints in their models. However, there are many industries tends to material intensive in their manufacturing environments. Thus, material and capacity should be considered simultaneously to achieve the best performance when making scheduling decisions (Musselman and Uzsoy, 2001). Unlike other researches in assembly job shop scheduling, this dissertation is not only studying on the assembly job shop scheduling, but also brings attention to the availability of components which required to assembly for final products. We assume the availability of components is dependent on deliveries made by vendors based on different arrival times and the quantities of components may not be sufficient for all jobs in the beginning. It means that the assembly operations in the lowest level of the BOM (bill of materials) are not only waiting for parallel components but also the arrival of components from suppliers. If a component does not have sufficient quantity on hand, a job's completion time will be delayed and the due date will be missed even there is an infinite capacity in the production system. It is important to have the capacity schedules which are in agreement with material plans to reduce total tardiness of all orders in an assembly job shop.

1.3 Research Summary

The assembly job shop studied in this dissertation consists of m machines and n jobs. Each machine is unique and can only process one job at a time. Each job has to process by the specific machines through the shop as stated on predetermined routing. A

job represents an order for a final product which requested by customers. The due date, priority and operation time for each job are known in advance. Unlike conventional job shop with string-type jobs, a final product in an assembly consists of assemblies, sub-assemblies and components based on its Bill-Of-Materials (BOM) structure. The quantity of assemblies, subassemblies and components needed to form a final product can be computed according to its BOM. In most assembly job shop scheduling problems, it assumes that components are produced in-house or no components needed at all. But there is a different assumption for assembly job shop models in this dissertation, which is components can not be manufactured in-house and need to be purchased from outside suppliers. A job can not be processed if there are insufficient quantities of components. An example for different product structures is shown in Figure 1.1.

The demands for final products are known in advance. Thus, the arrival time and quantities of components from outside vendors is also deterministic. In addition, all jobs are ready to process in the beginning but there is no inventory in an assembly job shop. Once a job allocates necessary components to form a subassembly or an assembly, it is ready to be processed by specific machines according to its routing. However, there could be more than one job to request the same components or specific machines. If these resources utilization are tight, a decision must be made to allocate limited resources to most urgent jobs first to avoiding miss their due dates. The goal of assembly job shop scheduling problems in this dissertation is to schedule all jobs and complete them before their due dates under resources constraints. Failure to meet these due dates will result in penalties and the loss of future business. To measure the performance related to due dates

and the importance of customer orders, the objective function of the assembly job shop scheduling problem is to minimize the total weighted tardiness of all jobs.

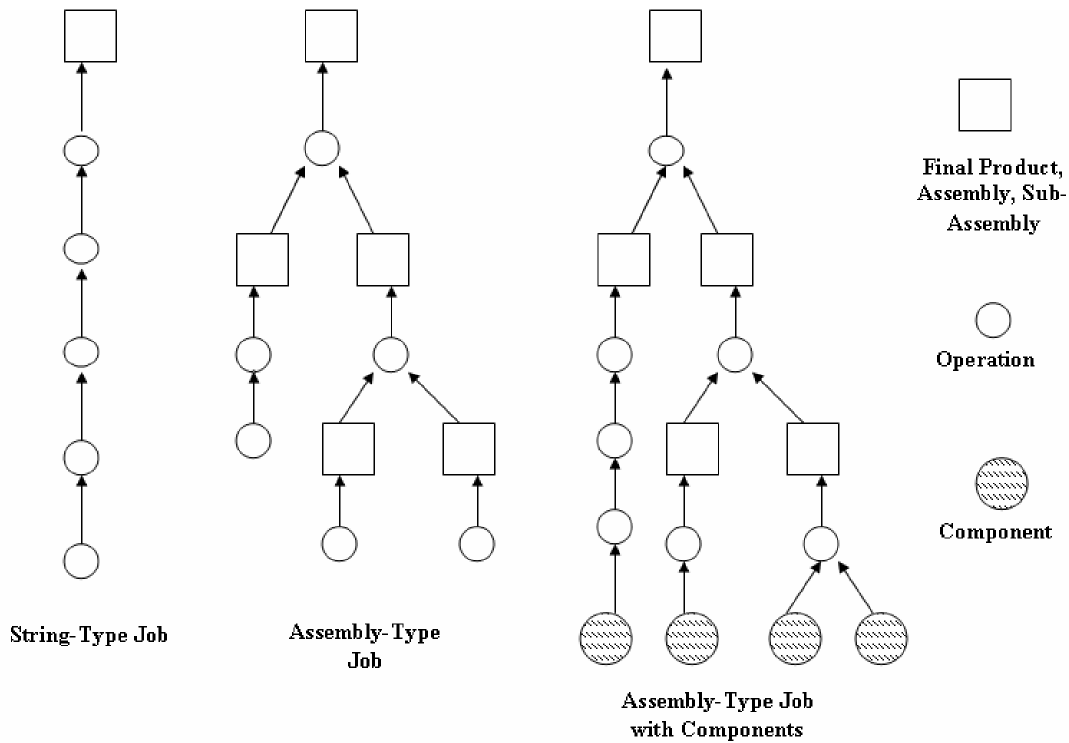


Figure 1.1 Example of different product structures

This dissertation intends to achiever two objectives. First, unlike traditional assembly job shop scheduling, it is interesting to study the scheduling problems to satisfy both the capacity constraints within machines and the material constraints between different components in an assembly job shop. This work can help manufactures make better decisions on production scheduling and improve the due date performance. Second, although assembly job shop scheduling is common in practice, it is intrinsically a NP-

hard problem when solving by optimization methods. Most of the current research uses dispatching rules to solve assembly job shop scheduling to reduce its complexity. However, these dispatching rules are myopic and lack the ability to consider the broader view of the entire problem. Solutions obtained from dispatching rules cannot compare with the solutions obtained from the optimization method. While optimization methods assure optimality, they are impractical because of intensive computational requirements for large problems. Dispatching rules avoid computational difficulties, but they are doubtful to provide good solutions. Based on the consideration of solution quality and computational time, it is reasonable to develop efficient algorithms that can find acceptable solutions which are close to optimal solutions in a limited amount of computational efforts.

A generic assembly job shop scheduling problem with component availability constraints is formulated as a mixed-integer programming model and solved by using commercial optimization software. The objective function of our models is to minimize the total weighted tardiness, which designs to measure due date performance. The optimization approach here not only provides a lower bound compared to the performance of heuristic techniques, but also applies to real-world problems if the computational time is acceptable. Several scalable and computationally efficient heuristics are also developed based on the shifting bottleneck procedure concept to allow trade-offs between the computation time and the solution quality. The original shifting bottleneck procedure (Adams et al. 1988) was designed to solve the job shop scheduling problem with scheduling a single machine of each machine type to minimize makespan. However, it is common to have an assembly system in many manufacturing

environments and existing job shop scheduling techniques are not suitable for them. In this dissertation, the shifting bottleneck procedure is extended to solve the assembly job shop scheduling problem with component availability constraints.

The modified shifting bottleneck procedure (MSBP) is to decompose the original problem into numerous different subproblems which present material plans for components or capacity schedules for machines. The extended disjunctive graph is used to describe the relation between different jobs. After solving each subproblem, machine schedules or material plans are updated iteratively until the whole problem being solved. By doing so, it is able to reduce the complexity of the original problem and keep solutions at acceptable levels. Additionally, subproblems can be solved by dispatching rules or efficient algorithms to reduce the computational times. Based on the degree of solution accuracy obtained from subproblems, there are four MSBP methods developed in this research.

However, the computational time may still grow exponentially for MSBP as the size of problems increases. Because the total number of subproblems required in MSBP and the computational times needed to solve a subproblem are extremely large. To reduce the computational efforts, we introduce the efficient shifting bottleneck procedure (ESBP), where bottleneck resource selection procedure is avoided and re-optimization procedure is constrained to only few iterations. It designs to reduce the total number of subproblems to be involved in whole shifting bottleneck procedure and also provides acceptable solutions within realistic computational effort. By constraining the total number of subproblems, ESBP is suited for solving assembly job shop scheduling

problems in practice. Moreover, based on the degree of solution accuracy, there are three ESBP methods developed in this dissertation.

Although ESBP reduces the total number of subproblems involved to solve the assembly job shop scheduling problem, the size of subproblem is still not constrained properly. This means that computational times required to solve subproblems could grow exponentially as the size of subproblems increase. To capable of solving realistic problems, it is necessary to reduce the complex of subproblem. The rolling horizon procedure (RHP), which implemented the rolling horizon procedure to solve subproblems, is designed to constrain the size of subproblems. By applying RHP in each subproblem, it divides the subproblem into numerous time windows and each time window contains the total number of operations can be optimized independently within a short computational efforts. By doing this, the complexity of overall problem will be restricted in an accepted level.

Additionally, a set of test problems are used to verify the performance of optimization and the effectiveness of heuristics approaches and demonstrate the impact of the assembly job shop scheduling on the due date performance of a production system.

1.4 Contribution

The benefits of the research in the assembly job shop scheduling problems with component availability constraints are summarized as follows:

(1). This research extends the study of assembly job shop scheduling problems with component availability constraints.

(2). The modified disjunctive graph is developed to represent the assembly job shop model with consideration of components purchased from outside vendors.

(3). The mathematical formulation is created to represent the assembly job shop model.

(4). Four different approaches, which address the different degree of scalability, to solve the assembly job shop scheduling problems with component availability constraints, included MSBP, ESBP, RHP and dispatching rules.

(5). All approaches are coded in commercial mathematical modeling and optimization software XPRESS-MP. Numerous experiments are carried out to evaluate the performance of proposed approaches.

(6). This dissertation demonstrates how the decomposition methodology can reduce the complexity of NP-hard scheduling problems.

(7). The mathematical formulation is formed to represent the dynamic assembly job shop with component availability constraints.

1.5 Organization

This dissertation is organized into six chapters. The first chapter describes the statements of problems and the overview of general approaches related to the problem. The contribution of this research is also addressed in this chapter. Then, it is followed by the organizations of this dissertation. The rest of the dissertation is organized as follows: Chapter 2 reviews the relevant literature on assembly job shop scheduling with component availability constraints; Chapter 3 describes a mixed-integer programming model for the assembly job shop. The optimization approach and several heuristics based

on the concept of the shifting bottleneck procedure are introduced to solve assembly job shop problems; Chapter 4 outlines the details of experiments and demonstrates the computational results of our methodologies. Future research directions are given in Chapter 5 and finally, conclusions are discussed in Chapter 6.

Chapter 2. Literature Review

Scheduling is a decision-making process that is used daily as an instruction to perform certain activities in many organizations. The functionality of scheduling is to allocate limited resources to complete given tasks (or activities) over time in order to optimize the performance of a system (Pinedo, 2005). This dissertation focuses on the scheduling problems in manufacturing environments, which are referred to as the production scheduling problem. In production scheduling problems, resources could be machines, tooling, components or labor. Activities are operations, setups or transportation. The purpose of production scheduling is to schedule jobs and control their flow through a production process efficiently. A schedule for a production system should contain a set of operations with start times and machine assignments for each operation of all jobs to be scheduled. Each activity may have a due date which could be committed externally or internally. Moreover, to address the importance of an activity, each activity may have a priority level and can be used to expedite high priority jobs through the production process.

On time delivery and short customer lead times are important to keep future business. They are much easier to achieve if resource utilization is very low and inventory level is always enormous. However, manufacturers expect to have high resource utilization and low inventory level to survive in today's global competitive business environments. It is hard to achieve the high utilization and keep delivery promises at the same time. Furthermore, the performance measures of a production system usually set to maximize the utilization of resources, minimize cycle time, reduce system WIP or lower tardy ratio. But, it is impossible to satisfy all of the above objectives

unless manufacturers have unlimited resources, which is impractical in today's manufacturing environments. To increase the effectiveness of the production system, the ability to allocate critical resources to production activities efficiently becomes a key concern for manufacturers.

In addition, the production schedule also provides visibility for possible conflicts between critical resources in the future. It allows manufacturers to identify criticality of production resources (capacity, materials, labors and so on) in advance and balance them to avoid further delay on delivery. All of the above reasons make production scheduling an important tool in today's manufacturing environments. Moreover, more research agrees that in addition to the classical scheduling problem, decisions such as order release, due date quotation and lot sizing are related closely to the production scheduling problem (K. Kempf et al., 2000).

Job shop scheduling is one of most complicated production scheduling problems in practice. In this chapter, we will first review the related research on the job shop scheduling problem and then extend it to the assembly job shop scheduling problem which involves jobs with assembly operations in their product structure. We will also provide a review on various scheduling techniques which are used to solve job shop scheduling problems, such as shifting bottleneck procedures (Adams et al., 1988). In the following section, a short review on component availability problems in job shop scheduling is also given as shown in Table 2.1.

Table 2.1 Classification for relevant studies:

Class	**Related Research**
Classical job shop scheduling	Cheng and Cupta (1989), Lawler *et al.* (1993), Adams et al. (1988), Pinedo (2006)
Assembly job shop scheduling	Conway and Maxwell (1962), Maxwell and Mehea (1968), Russell and Taylor (1985), Elvers and Treleven (1985), Adam et al. (1987), Fry et al. (1989), Philipoom et al. (1991), Reeja and Rajendran (2000) , Park and Kim (1999) , Park and Kim (2000), Pathumnakul and Egbelu (2006)
Component availability	Tang (1988), Luss and Smith (1988), Klein et al. (1994), Klein et al. (1994) , Guide and Srivastava (2000) , Karmarker and Nambimadom (1996), Kolisch (2000) , Balakrishnan and Geunes (2000), Klein et al. (1995)
Shifting bottleneck procedure	Chen et al. (2006) , Adams et al. (1988), Ovacik and Uzsoy (1997) , Uzsoy and Wang (2000) , Demirkol et al. (1997)

2.1 Classical Job Shop Scheduling

The production scheduling problem has received great attention from researchers in past decades. According to Cheng and Gupta (1989), the scheduling problem can be classified as static vs. dynamic, deterministic vs. stochastic, single-product vs. multi-product, single-processor vs. multi-processor facilities and theory vs. practice. In the static job shop model, all jobs are available to process at the same starting time. In the dynamic job shop model, the number of jobs available for processing varies over time, jobs continually enter and leave the production system in a random manner governed by some probabilistic laws. This dissertation focuses on a static, deterministic, multi-product production scheduling problem.

The problem of production scheduling involves the allocation of limited resources to perform given tasks over time to optimize the performance of a system. Job shop scheduling is one of the most classical and complicated scheduling problems. The job shop scheduling problem concerns the allocation of n jobs to m machines and each job follows a predetermined routing through the shop. There are three approaches for solving job shop-scheduling problems: dispatching rules, combinational optimization, local search and constraints programming. Optimization approaches include the branch and bound method, the lagrangian relaxation based method and the dynamic programming method to provide an optimal solution, but they are only capable of solving small-sized problems and require enormous amounts of computational time. In job shop, each job has a different order to visit specific machines with different operation times as stated on the routing. Since every job has a different routing, there is up to $n!$ sequences for each machine and $nm!$ possible solutions if each job visits every machine once. Not all of them

are feasible and it is impracticable to test all possible sequences within a reasonable computational time. Hence, finding a good solution for the job shop scheduling problem is very difficult in large scale instances. Conflicts with constraints arise frequently especially in the allocation of critical resources. To overcome this, research tends to develop heuristic algorithms to identify possible sequences and provide acceptable solutions with reasonable computational efforts. The dispatching rule is used to prioritize the jobs which are waiting in the machine queue and choose the one with highest value to process. The dispatching rule is widely used in practice because it produces feasible solutions for large scale instances within acceptable computational time. Moreover, production systems in real world are always stochastic. Machines can be broken at any moment and components are not always available. In this dissertation, we consider the availability of components and present an algorithm performing in situations encountered large instances in practice.

The job shop scheduling problem (JSP) is a class of combinational optimization problems known as NP-hard one. In the last decades, many researchers have become interested in such problems and job shop scheduling problem was received significant attentions because it determines jobs sequences and resources utilization in production planning. In a classical job shop scheduling problem, there are n jobs available to be processed on m machines. Each job has a predetermined routing which contains operations to direct the order through he shop in a specific order. Each operation has its own processing time and required machine. Moreover, each job has restrictions on release time and due date which are set externally by production planner or customers. An earliness or tardiness penalty may be incurred if a job has been completed early or

late. The objective of the job shop scheduling problem is to identify the job sequences on machines in order to optimize the performance of the production system, which includes maximizing resources utilization, minimizing lead times, minimizing work in process, minimizing makespan or minimizing order tardiness.

There are many different objectives on job shop scheduling problems. Following the three-field notation as described by Lawler *et al.* (1993), we shall refer to the job-shop scheduling problem of minimizing makespan as $J//C$max, and that of minimizing maximum lateness as $J//L$ max. Adams et al. (1988) define the job shop scheduling problem of minimizing makespan as follows. Let N represent the set of operations with operations S and T representing the dummy operations "start" and "finish" operations. Let M represent the set of machines. Let A denote the set of pairs of operations constrained by precedence relations representing the condition that the sequence of machines is prescribed for each job. Let E_k represent the set of pairs of operations to be performed on machine k with non-overlapping durations. Hence, each machine can only process one job at any given time. Let p_i and t_i represent the processing time and start time of operation i, respectively. The minimum makespan job-shop scheduling problem can be modeled as:

$$\min t_T$$

$$t_j - t_i \geq p_i, \qquad (i, j) \in A,$$

$$t_i \geq 0, \qquad i \in N,$$

$$t_j - t_i \geq p_i \vee t_i - t_j \geq p_j, \qquad (i, j) \in E_k, k \in M. \qquad (P)$$

A feasible solution to (P) is called a schedule. The first constraint ensures the operations sequence of a job following a predetermined order as stated in job routings.

The second constraint assures the start time of operations cannot be negative. The third constraint guarantees no two operations can be processed at the same time. Moreover, the first constraint can be referred as precedence constraint, second, as non-negativity constraint and third, as capacity constraint or disjunctive constraint.

According to Pinedo (2006), we can classify the methodologies to solve job shop scheduling problems as follows: (1) Optimization approach; (2) Heuristics approach; (3) Constraint Programming.

(1). Optimization approach: A number of researchers have focused on exact solution procedures to solve job shop scheduling problems optimally. These scheduling problems are usually formulated as integer programming models and solved by various methods, included branch and bound procedures, cutting plane methods and hybrid methods. However, these scheduling problems are also referred to as NP-hard and the computational time of the optimization methods grows exponentially in the worst case when the size of problem instances increased. Hence, most optimization approaches are only suitable for small problems practically. In addition, the largest instances that have been solved to optimality are 20 jobs and 10 machines up-to-date.

(2) Heuristics approach: Extensive research has been studied on heuristics that can produce near-optimal solutions with less computational times in the past decades. These heuristics are referred as polynomial time algorithm. Heuristic approaches for job shop scheduling problems can mainly be divided into two categories: constructive procedures and local search procedures. The local search approaches are all based on neighborhood

structures defining how to obtain a new solution from the current ones, such as tabu search and simulated algorithm. Instead of trying to improve an initial solution using moves in a given neighborhood, the constructive procedures build a schedule from scratch by inserting unscheduled operations into a partial schedule until the schedule is complete. Dispatching rules, which are extensively discussed in many literatures, are widely used in practice. Whenever a machine becomes free, these rules examine the currently available jobs and select the next job to be scheduled based on a priority index derived from job and machine characteristics. The advantages of dispatching rules are easy implementation and low computational burden. However, their myopic nature can result in poor performance due to lack of consideration of the effects of local decisions on the whole shop. Another well-known constructive procedure is the shifting bottleneck procedure developed by Adams *et al.* (1988), which decomposes the original problem into numerous subproblems and solves them iteratively. Shifting bottleneck procedure will be discussed comprehensively in the following section.

(3). Constraint programming: Constraint programming was originally from the constraint satisfaction problems which are required a search for feasible solutions which satisfied all constraints. According to Pindeo (2006), constraint programming is not only used to solve feasible solutions, but also used to solve optimization problems in the last decade. These approaches include the Optimization Programming Language (OPL), which designed for modeling and solving optimization models through both constraint programming techniques and mathematical programming procedures.

2.2 Assembly Job Shop Scheduling

Scheduling problems are generally very complex in nature, particularly in the case of assembly job-shop scheduling where products with multiple components are manufactured. This dissertation studies a multi-level assembly job shop scheduling in a static make-to-order environment. An assembly job shop in this dissertation refers to a shop that carries both processing and assembly operations where consumable resources (assemblies, sub-assemblies or components) have to be assembled together and processed by specific non-consumable resources as stated on job routings. A massive amount of research has studied job shop scheduling problems with the assumption of simple string-type jobs with a linear routing to simplify the problems over past decades. But most products may require some assembly operations in industry. Moreover, there are more shops processing multi-level assembly jobs than shops performing simple string-type jobs in today's manufacturing environment, particularly in electric industry. However, literatures related to assembly job shop scheduling are somewhat limited.

In an assembly job shop, the operations of an item are not only performed in serial following the precedence relationships, but are also carried out in parallel. In this context, an item, a component, or a sub-assembly may need to wait for the components belonging to the same item before the required assembly operations can take place. Moreover, a job in an assembly job shop may not only spend time on the queue of non-consumable resources, but may also wait for components which may carry out in parallel. To produce a feasible schedule in an assembly job shop, one has to consider not only the capability of non-consumable resources, but also the availability of consumable resources (assemblies, sub-assemblies or components) as stated on job routings. This makes the scheduling

problem in assembly job shop scheduling much more difficult than conventional job-shop scheduling.

Conway and Maxwell (1962) showed that priority rules based on slack time performed much better than the shortest processing rule (SPT) when shop loads were relatively high. They demonstrated the need to coordinate or synchronize the completion of the parts in the same job. There are several interesting points that must be mentioned in the attempt to summarize the research on priority scheduling rules for job shops with assembly operations. First, the number of studies reported is insignificant when compared with those on priority rules for primary job is not appropriate for jobs with assembly operations. This is because of the need to coordinate the completion time of related parts and inability of SPT to achieve this.

Maxwell and Mehea (1968) tested the performance of several basic priority rules and some composite rules in their experiments when multi-level "symmetric tree structured" job structure is used. They concluded that the performance of the shop can be improved if job structure is considered in the composite priority rules.

Russell and Taylor (1985) studied the impact of product structure on the performance of sequencing rules. Two product structures were created to represent two different product structures, one tall BOM and one flat BOM. Several sequencing rules were evaluated by using a simulation analysis of a hypothetical assembly shop and simulation results are analyzed via an ANOVA procedure that identifies major differences in the results of several performance measures. In their study, they concluded that the sequencing rule has a significant effect on the mean flow time, and root mean square from tardiness of jobs completed by an assembly shop. Moreover, sequencing

rules that excel in a simple job shop are not necessarily appropriate for an assembly shop. Although the sequencing rule's performance is affected by the structure of jobs processed, the ranking of sequencing rules does not differ significantly under job structure.

Treleven and Elvers (1985) have shown that the routing pattern of jobs has no significant impact on the relative effectiveness of various dispatching rules. Their study concludes that if lateness criteria are considered most important, then earliest due date and least slack per remaining number of operations dispatching rules are considered good choices.

Adam et al. (1987) studied the priority rules for assembly job shops which processed multiple-level assembly jobs. They classified the job lead time into two components: flow time and job staging delays. They developed a set of priority rules which the structural complexity of jobs can be incorporated explicitly to reduce the staging delays. They concluded that the combined rules, such as a pacing rule in conjunction with an acceleration rule, can improve the performance of lead time. Also these combined rules also show a significant improvement over other priority rules if higher levels of assembly jobs are involved.

Fry et al. (1989) studied the performance of priority dispatching rules on three different product structures which are flat, tall and complex. In their experiments, a six-machines assembly shop was simulated to evaluate the performance of fourteen sequencing rules. Results indicate that taller-structured BOMs tend to have more tardy orders than the flat BOMs because more staging delays are involved. Hence, due-date oriented sequencing rules will perform better when product structures become taller. Moreover, the SPT rule is not appropriate to use in an assembly job shop even though it

performs better in a single-stage job shop. Also, they found no evidence that using the operation due-date reduces order tardiness. They suggested that using specific sequencing rules for certain product structures in an assembly shop is important to improve due date performance.

Philipoom et al. (1991) proposed a new set of sequencing rules, called importance ratio (IR). It is based on the ratio of the remaining number of operations on a particular branch or path to job completion, to the remaining number of operations on the longest path to job completion. In their study, importance ratio rules perform better on jobs with higher levels of assembly. They also evaluated the performance of several sequencing rules which incorporate multiple attributes of assembly job shop. In their experiments, they concluded that it is not necessary to incorporate both attributes of job shop and assembly shop scheduling into sequencing rules to achieve best performance.

Reeja and Rajendran (2000) studied the performance of dispatching rules on assembly job shops. New dispatching rules based on "operation synchronization date" were introduced and tested the performance on a simulation study. The results indicated that new dispatching rules are superior to others in most performance measurements.

Park and Kim (1999) developed a non-linear mathematical model for a production planning problem in an assembly system with discrete time representation. In their model, all jobs must be completed before their due date and the objective is to minimize the total holding costs including work-in-process and final product inventory. A heuristic algorithm based on a network model was also presented and used to generate a production plan from downstream workstations to upstream workstations. In their experiment, they concluded that performance of their algorithms was superior to

commercial software and the backward finite-loading methods they tested. Park and Kim (2000) also presented a mixed integer programming model with continuous time representation to solve the same problem. A branch and bound algorithm based on a lagrangian relaxation method is introduced and tested on a set of randomly generated problems with a maximum number of products up to seven and the number of machines equals to three. Results from their experiments demonstrated that the branch and bound algorithm can find optimal solutions within reasonable computational time.

Pathumnakul and Egbelu (2006) studied the problem of minimizing the weighted earliness penalty in assembly job shops. In their study, tardiness is not allowed and all orders must be completed before due dates. A mathematical model based on continuous time representation was presented and a heuristics algorithm based on several local optimality properties was developed to reduce intensive computational time associated with the problem. The performance of heuristics is closed to optimal solutions obtained from 50 test problems with maximum number of machines and products up to five.

2.3 Assembly Job Shop Scheduling with component availability constraints

As mentioned in the previous section, research on job shop scheduling has concentrated on the problems with simple string-type jobs. Few of them studied the job shop scheduling problems with multi-level assembly operations. Most of them have an assumption that all components are always available at the beginning of operations to simplify problems. In other words, there are no insufficient components problems and capacity constraints are major concerns in their models. However, to reduce the lead time of orders, manufacturers may release partial orders to the shop floor before arrivals of all

necessary components from outside suppliers. Thus, components from outside suppliers could arrive late and rush orders could be released to the shop floor at any time. Only considering capacity constraints in assembly job shop models may not be suitable for manufacturers which tended to be material intensive.

Most famous material allocation example in practice is Material Requirement Planning (MRP). MRP was designed to plan and control flows of materials and provide valid order due dates for production systems. MRP provides components priorities in the form of component due dates by offsetting component gross requirements by expected lead times. Once manufacturing jobs are ready to be released to the shop floor, order release function in MRP will allocate available components to manufacturing jobs. When a conflict arises between different manufacturing jobs, the order release function will decide which manufacturing jobs are released to the shop floor and which ones remain staying in the job pool until the arrival of components in the future. By doing so, manufacturers only need to deal with capacity constraints in production scheduling problems. However, the fixed-lead time assumption of MRP tends to make the schedule inaccurate. Changes on demand frequently in current business environment will cause the disruption of production schedules generated by MRP. Thus, poor schedules will result in inappropriate delivery from outside suppliers. If supplier deliveries are too early, it causes too much materials inventory. If they arrive behind schedule, production lines will be half due to insufficient quantities of inventory. To overcome these drawbacks, a new scheduling approach, called Advanced Planning and Scheduling (APS), generates schedules that consider both material and capacity simultaneously as elements that may constrain production (Musselman and Uzsoy, 2001). This ensures that the material plan is

in agreement with capacity schedule. In this section, we review the research which related to resource allocation problems.

The resource allocation problem has been studied in different fields, such as supply chain management, logistics and store space design. Tang (1988) describes a nonsimplex-based integer algorithm to solve a max-min allocation model to optima. The original problem is decomposed into subproblems and the optimal solution is obtained by literately solving a set of subproblems. It is clear that the computational time of his algorithm depends on how fast subproblems can be solved optimally. Other researchers have studied max-min problems in the past. However, Tang (1988) is one of few researchers to apply max-min allocation models to manufacturing problems, such as storage space allocation, line balance, production quantity allocation and raw material allocation problems. Luss and Smith (1988) describe a minimax approach to solve multiperiod allocation problems on electronic assembly industry. The objective function of their model is to minimize the maximum weighted deviation of cumulative activity levels (jobs) from the cumulative demands. That is, their model allocates resources (electronic components) to activities (jobs) over a multiperiod and minimizes the percentage of incomplete jobs. Their objective is capable of solving large problems very quickly. Klein et al. (1994) describe resource allocation models for the electronic assembly industry and solve those using relaxation-based approaches. They also integrate the component substitution constraints into their models. Although their models can be applied to manufacturing resource allocation problems, they are still too simple to apply in practice due to single-level BOM and single material structure assumptions.

Guide and Srivastava (2000) survey the recent techniques that deal with the uncertainty in material requirements. Those methodologies include lead time setting, the choice of lot-size and shop floor control. Some researchers suggest carrying safety stock to deal with quantity uncertainty and safety lead-time against timing uncertainty.

Karmarker and Nambimadom (1996) propose a formal model for the material allocation problem in MRP. In their paper, they address the flaws in the material allocation function of MRP, such as the simple due-date sequence allocation rule and lack of priority issue. They use the lagrange relaxation technique to solve this problem and provide a lower bound on the optimal solutions. They also develop four heuristics based on modified MRP logic to achieve shorter computational time for large scale problems. Moreover, an improvement procedure is also designed to decrease tardiness. The objective of these models is to minimize penalties of late orders. Their models consider multi-BOM structures. In their experiments, the Lagrangian relaxation heuristic performs 5% to 10% worse than the optimal solution. The best modified MRP logic heuristic reduced cost by 25% to 40% compared to MRP. It is noted that the computational time of Lagrangian relaxation heuristic is about 50 times longer than that of MRP.

Kolisch (2000) describes a mixed integer programming model to solve assembly scheduling problems in a make-to-order (MTO) environment. In his model, jobs are scheduled based on capacity, assembly spaces, parts availability and jobs precedence constraints. The objective function is to minimize the total tardiness of orders. Due to the excess computational time, a list-scheduling heuristic is proposed in his paper as well. It is worth noting that the part assignment problem in this paper is solved as a transportation problem.

Balakrishnan and Geunes (2000) propose a dynamic requirement planning model with BOM flexibility based on mixed-integer programming. The BOM-flexibility function allows manufacturing to switch from one material to another based on cost or availability. Given the BOM flexibility and demand for each product, a requirements planning with substitutions problem (RPS) is represented by a generalized network-flow model. In the model, there are three different nodes: origin nodes, component nodes and demand nodes, which represent component type, inventory level and actual demand component, respectively. Three different arcs: productions arcs, inventory arcs and usage arcs, which describe the activities of the production system. After the RPS problem is translated into a generalized network-flow model, it is solved as a shortest path problem. Balakrishnan and Geunes (2000) develop a dynamic programming to solve the shortest path problems. Their experiment indicates that flexibility-BOM can save 8.7% in production cost. The objective of RPS is to reduce production cost but not the safety stocks requirement that has been widely studied in this field.

Klein et al. (1995) propose a multiperiod allocation model with substitution constraints. In their model, resources are storable and can be used in subsequent periods in the case of surplus resources. The objective function is to minimize the largest cumulative deviation from cumulative demands. It can be explained in a way which minimizes the largest tardy jobs ratio in material allocation problems of the electronic assembly industry. They propose a relaxation-based algorithm to solve multiperiod allocation problems. For each iteration, the usage of activity (jobs) is given and a multiperiod allocation feasibility problem is solved to determine the corresponding

feasibility solution. They formulate the multiperiod allocation feasibility problem as a network flow model and solved using maximal flow problems.

2.4 Shifting bottleneck procedures

Job shop scheduling involving more than two machines is one of well known NP-hard problems for researchers. To avoid the intensive computational time to solve large instances, it is reasonable to divide the original problem into several small subproblems which can be solved efficiently and still able to achieve good solution. The shifting bottleneck procedure is one of popular algorithm to adopt this concept on job shop scheduling problems. The idea of shifting bottleneck procedure (Adams et al. 1988) is to breakdown the job-shop scheduling problem into numerous single-machine subproblems which can solve sequentially based on subproblem criticality and use disjunctive graph to present operations precedence constraints and reflect machine sequence decisions. In their experiments, shifting bottleneck procedure is superior to other algorithms in term of computational time and solution quality. However, there are usually numerous machines with similar functionality to be group as work-centers in practice. Instead of solving one machine subproblems, parallel machines (work-centers) subproblems in shifting bottleneck procedure can be solved to capable of solving real-world job shop scheduling problems (Ovacik and Uzsoy, 1997). However, shifting bottleneck procedure still requires the excess of computational times to solve job shop scheduling problems with parallel machines due to numerous subproblems need be solved. Chen et al. (2006) introduce the efficient shifting bottleneck heuristic (ESBPM) which designs to reduce

total number of subproblems to be involved in shifting bottleneck procedure and also capable to solve job-shop problems with parallel machines in practice.

2.4.1 The Generic Shifting Bottleneck Procedure

Adams et al. (1988) define the job-shop scheduling problem as follows. Let N represent the set of operations with operations S and T representing the dummy operations "start" and "finish" operations. Let M represent the set of machines. Let A denote the set of pairs of operations constrained by precedence relations representing the condition that the sequence of machines is prescribed for each job. Let E_k represent the set of pairs of operations to be performed on machine k with non-overlapping durations. Hence, each machine can only process one job at any given time. Let p_i and t_i represent the processing time and start time of operation i, respectively. The job-shop scheduling problem can then be formulated as:

$$\min t_T$$

$$t_j - t_i \geq p_i, \qquad (i, j) \in A,$$

$$t_i \geq 0, \qquad i \in N,$$

$$t_j - t_i \geq p_i \vee t_i - t_j \geq p_j, \qquad (i, j) \in E_k,\ k \in M. \qquad (P)$$

A feasible solution to (P) is called a schedule. It is helpful to use a disjunctive graph G to represent this problem as shown in Figure 1. Let $G = (N, A, E)$, where N is the node set, A is the ordinary (conjunctive) arc set, and E is the disjunctive arc set. Each node is an operation. The directed arcs correspond to precedence relations. The pairs of disjunctive arcs correspond to the pairs of operations that are performed on the same

machine. The processing time of an operation is shown on an arc. The set of disjunctive arcs, E, consists of cliques E_k, one for each machine k, where $E = \cup(E_k: k \in M)$.

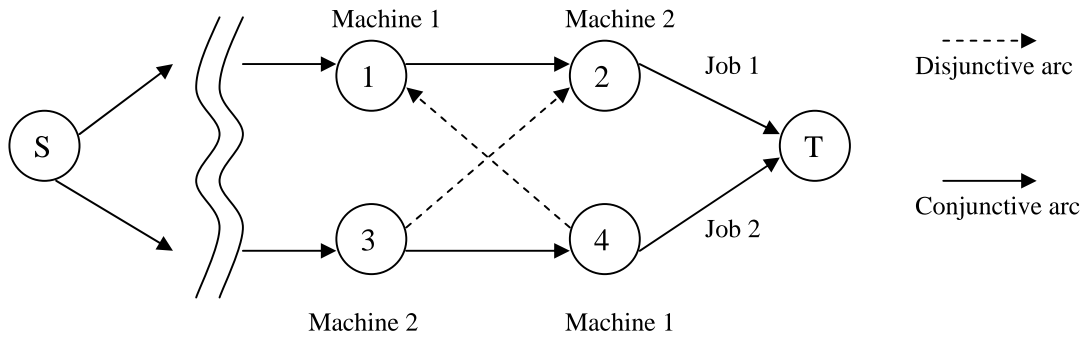


Figure 2.1 Example of the disjunctive graph G

The directed graph that results from taking out the disjunctive arcs from G is represented by $D = (N, A)$. S_k is called a selection if it includes one and only one member of each disjunctive arc pair of E_k. If a selection does not have a directed cycle, it is acyclic. Since each acyclic selection (S_k) corresponds to a particular ordering of the operations associated with machine k, sequencing a machine k is equivalent to an acyclic selection in E_k. The union of the selections S_k (one in each E_k, $k \in M$) is called a complete selection S, and a partial selection consists of the union taken over a subset (M_0) of M. An ordinary directed graph, $D_S = (N, A \cup S)$, results from choosing a complete selection S, that is, putting the ordinary conjunctive arc set S in place of the disjunctive arc set E. The length of a longest path in D_S is equivalent to the makespan of a schedule correspond to the selection S. Therefore, the goal is to find an acyclic, complete selection $S \subset E$ that minimizes the length of a longest path in the directed graph D_S.

Based on the disjunctive graph representation, we can describe the shifting bottleneck procedure defined by Adams et al (1988). Let M be the set of all machines. M_0, a subset of M, is defined as the set of machines for which a selection of disjunctive arcs (representing a sequence of jobs) has been determined in the earlier iterations. Initially, M_0 is an empty set. The result of a typical iteration is the selection of a machine from $M - M_0$ to be added to M_0. In addition, the processing order of the operations for those machines must be specified. The goal is to determine the unscheduled machine that causes the severest disruption in production if not scheduled immediately. This bottleneck machine will be the next one added to M_0. In order to find the bottleneck machine, in the original directed graph, all the disjunctive arcs for the machines that have yet to be scheduled are removed. In addition, only the relevant disjunctive arcs of the machines that have already been scheduled are retained. Call this graph G'. There are one or more critical paths in G' that determine the graph's makespan. Call this makespan $C_{\max}(M_0)$. Note that machine k will become machine type k when parallel machines (work-centers) are considered.

The three key tasks in the shifting bottleneck procedure are described below:

(1). Bottleneck machine selection procedure:

The critical paths in G' determine the release dates and the due dates for all the operations. The definitions of the release date(r_i) and due date(d_i) for an operation are: $r_i = L(0, i)$ and $d_i = L(0, n) - L(i, n) + p_i$, where p_i is the processing time for operation i and $L(i, j)$ is the length of a longest path from i to j in the graph. Let each of the machines that have yet to be sequenced be considered as a separate $1 \mid r_j \mid L_{\max}$ scheduling problem.

After solving the set of jobs sequence problems, the machine with the greatest maximum lateness is designated as the bottleneck machine and included in M_0 next. Call this machine, machine k, and its maximum lateness $L_{max}(k)$.

(2). Procedure to update the disjunctive graph G':

Schedule machine k using the solution found in solving the job sequence subproblem for machine k. Add the disjunctive arcs that specify the order of operations on machine k to G'. With the addition of these disjunctive arcs, the makespan for the partial selection goes up by $L_{max}(k)$: $C_{max}(M_0 \cup k) = C_{max}(M_0) + L_{max}(k)$.

(3). Reoptimization procedure:

The machines that have already been sequenced are resequenced. This is accomplished by taking a machine, machine l, form the set M_0. Graph G'' is formed by deleting the disjunctive arcs that specify the order of operations on machine l from G'. Machine l is resequenced by solving the corresponding job sequence subproblem $(1 \mid r_j \mid L_{max})$ using the release dates and due dates determined by the critical path(s) in G'' (Pinedo 1995). This may result in a new makespan.

After resequencing machine l, the machine is added back into M_0 at the same place it was taken out and the disjunctive arcs that specify the new order of operations on machine l are added to the graph G″. The makespan may have to be adjusted according to the maximum lateness determined by reoptimizing the sequencing for machine l. Resequencing all of the machines in the original set M_0 (excluding the machine just sequenced, machine k) completes one cycle of reoptimization.

If there is no improvement in the makespan after complete cycle of reoptimization, then the first local reoptimization procedure is complete. Otherwise, the first local reoptimization procedure continues until there is not any improvement within an entire cycle. The shifting bottleneck procedure will return to bottleneck machine selection procedure to select the next bottleneck machine. The procedure terminates when M_0 contains all machines types.

2.4.2 The Shifting Bottleneck Procedure with parallel machines

For job shop scheduling problems with parallel machines, not only jobs need to be sequenced a given machine type, but also the specific machine within a work-center which will process the job has to be determined. Chen et. al (2006)) describe a procedure similar to Ovacik and Uzsoy (1997) that uses the earliest-due-date first (EDD) dispatching rule to assign a machine within the work-center to a given job. In the next section, we described efficient version of the shifting bottleneck procedure. EDD dispatching rule is simple but effective for parallel machine allocation and sequence subproblem and provides the schedule without presenting cycles and delay precedence problem. The SB procedure with parallel machines (SBPM) can be described as follows:

Step 1. Let $M_0 = \varnothing$ and M represent the set of machine types.

Step 2. Remove all the disjunctive arcs in the graph. Obtain the makespan, C_{max}, of the graph by finding the longest path from the source node to the sink node.

Step 3. For each of the machine types in $M - M_0$, solve a $P \mid r_j \mid L_{max}$ problem with release dates and due dates determined by using the graph. Solve the parallel machine

allocation and sequence subproblems using the earliest-due-date-first (EDD) dispatching rule.

Step 4. Designate the machine type k with the greatest maximum lateness $L_{max}(k)$ as the "bottleneck machine". If there exists a tie for largest maximum lateness, designate the machine type k with the greatest average lateness $L_{ave}(k)$ as the "bottleneck machine". If there is a tie for largest average lateness, then designate the first machine that is tied for largest average lateness the "bottleneck" (an arbitrary rule). Add machine type k to the set M_0 and schedule machine type k using the optimal solution found in the $P \mid r_j \mid L_{max}$ problem for machine type k.

Step 5. Add the disjunctive arcs that specify the order of the operations on machine type k to the graph. Revise the makespan using the equation $C_{max}(M_0 \cup k) = C_{max}(M_0) + L_{max}(k)$.

Step 6. If $|M_0| = 1$, then local reoptimization is not needed, go to step 3. If $|M_0| > 1$, then resequence the machine types in M_0 one at a time until a full cycle goes by without any improvement. After resequencing a machine, say machine type l, put the machine type back into M_0 in the same place that it was removed from and add the disjunctive arcs that specify the order of the operations on machine type l to the graph. Adjust the makespan: $C_{max}(M_0 \cup l) = C_{max}(M_0) + L_{max}(l)$. If $|M_0| = |M|$, then terminate the procedure. Otherwise, go to step 3.

2.4.3 Efficient Shifting Bottleneck heuristic for parallel machines (ESBPM)

The job shop scheduling problem with parallel machines is harder to solve than job shop scheduling problems with single machine work-centers. In addition to solving a job

sequencing problem, it also needs to solve a machine allocation problem. Chen et al. (2006) propose an efficient shifting bottleneck heuristic for parallel machine work-centers, where bottleneck machine selection procedure is avoided and re-optimization procedure is constrained to few iterations. Hence, the total number of subproblems is expected to decrease. A generic efficient shifting bottleneck heuristic (ESBPM) can be described as follows:

Step 1. Choose an efficient dispatch rule or any other parallel machine algorithm to obtain job sequence and machine allocation of each machine type as the initial solution.

Step 2. Use re-optimization procedure to improve the initial solution. The parallel machine allocation and sequence subproblems can be solved by using a dispatching rule.

Chapter 3. Methodology

There has been a significant amount of research focused on assembly job shop scheduling, but we are not aware of any studies considered the availability of components. Moreover, most of these studies concentrated on the problems involved only capacity constraints as discussed in chapter 2. Despite of capacity constraints, the most popular example to allocate components to material requirements in practice is the First-come First serve (FCFS) dispatching rule or so-called MRP-logic (Material requirement planning). In MRP logic, the material required time is calculated as a function of the fixed lead time which represents capacity requirements or vendor lead times. Moreover, allocation of on-hand inventories and scheduled receipts are typically executed by a fixed decision rule based on the FCFS sequence. Once jobs held enough materials, they are released to the shop floor. In other words, material constraints have relaxed in advance and only capacity constraints remain in the scheduling problem. However, some components may have long vendor lead times and have to be procured in advance based on the forecast of demand. It could increase customer lead times if jobs are waiting for all required components before releasing them to the shop floor. Thus, releasing partial orders into the shop floor may help to improve the due date performance (Bragg et. al. 1999). Second, frequent change in demand and late delivery from outside vendors may delay the shipments and cause the loss of business reputation. A scheduling technique which incorporates both a material plan and a capacity schedule is necessary for a production system in order to respond quickly to these changes. Unlike discrete time representation used in conventional resource allocation methods, our model uses a continuous time representation to improve accuracy and defines tardiness cost as the

difference between the completion time and the due date. Moreover, the lead time in our model is dynamic and computed from both material plans and capacity schedules.

The goal of this dissertation is to develop algorithms to solve the assembly job shop scheduling problems, which are scalable to deal with real world problems. To accomplish this task, we must first describe the assembly job shop scheduling model with component availability constraints as a mixed-integer programming model. To overcome the excess of computational effort, several heuristics are developed to solve the same problem efficiently, including the modified shifting bottleneck procedure (MSBP), the efficient shifting bottleneck procedure (ESBP) and the rolling horizon procedure (RHP). The performance of these heuristics is compared to solutions obtained from several dispatching rules in terms of solution quality and computation time. The shifting bottleneck procedures developed in this dissertation is summarized in Table 3.1.

A modified shifting bottleneck procedure (MSBP) is designed to decompose the original problems into several subproblems which related to material plans or machine schedules in order to reduce the complexity of the problem. In the following sections, the efficient shifting bottleneck procedure (ESBP), which is based on the concept of shifting bottleneck procedure and the findings from MSBP, is introduced. ESBP is designed to reduce the total number of subproblems by avoiding the bottleneck resources selection procedure and constraining the number of re-optimization cycles. It consists of two main steps depicted as follows: Obtain initial solutions and perform re-optimization cycles. However, subproblems in both MSBP and ESBP are still NP-hard problems because of the nature of the mixing integer programming. The computational effort may not be acceptable if the size of problem is extremely large. To reduce the complexity of

subproblems, the rolling horizon procedure (RHP) is presented to obtain the solutions from subproblems rapidly in large size instances by dividing the scheduling time horizon into several different time windows. The performance of MSBP and ESBP is tested on a set of test problems in experiments and compared with various dispatching rules which are normally used in practice. The results from our experiments show that MSBP and ESBP outperform the best dispatching rule by an average of 19% to 22% for the all test problems.

Table 3.1 The summary of shifting bottleneck procedures for assembly job shop scheduling problem

Method	Bottleneck selection procedure	Re-optimization Procedures	Subproblem solution procedure
MSBP	Subproblem TWTmax (Max Total Weighted Tardiness) value	Full re-optimization up to 3 cycles	Branch and Bound or Dispatching rules
ESBP	Machine sequencing number	Full re-optimization up to 3 cycles	Branch and Bound or Dispatching rules
RHP	Machine sequencing number	Full re-optimization up to 3 cycles	Branch and Bound or Dispatching rules with rolling horizon procedure

3.1 Problem description and assumptions

An assembly job shop scheduling problem with component availability constraints is not only addressed on allocation of operations to machines but is also concerned with the allocation of available components to operations which have material requirements originating from an internal (subassembly) or an external (customer) source. The assembly job shop considered in this research consists of various machines. Each machine has different functionality and can only process one job at a time. A final product is fabricated from various items, included assemblies, sub-assemblies and components according the Bill-Of-Materials (BOM) structure as shown in Figure 3.1. The number of assemblies, sub-assemblies and components required to form a final product can be exploded from its BOM. In our assembly job shop model, there is no inventory for components in the beginning. Furthermore, the components can not be manufactured in-house and have to procure from outside vendors. Each shipment from outside vendors delivers single component type and is represented by a purchase order. Moreover, a job can not be processed by machines if there are insufficient quantities of components in an assembly job shop. There are also many jobs in assembly job shop. These jobs are ready to process in the beginning once they allocate enough components. Each job, which represents an order for a final product requested by customers, contains to numerous operations and need to be processed by different machines through assembly shop as stated on the predetermined job routing. Each job has the due date and priority which are known in advance. Failure to deliver final products on time will result in penalty. The objective of the problems is to schedule all jobs to complete on time and also satisfy both capacity and material constraints.

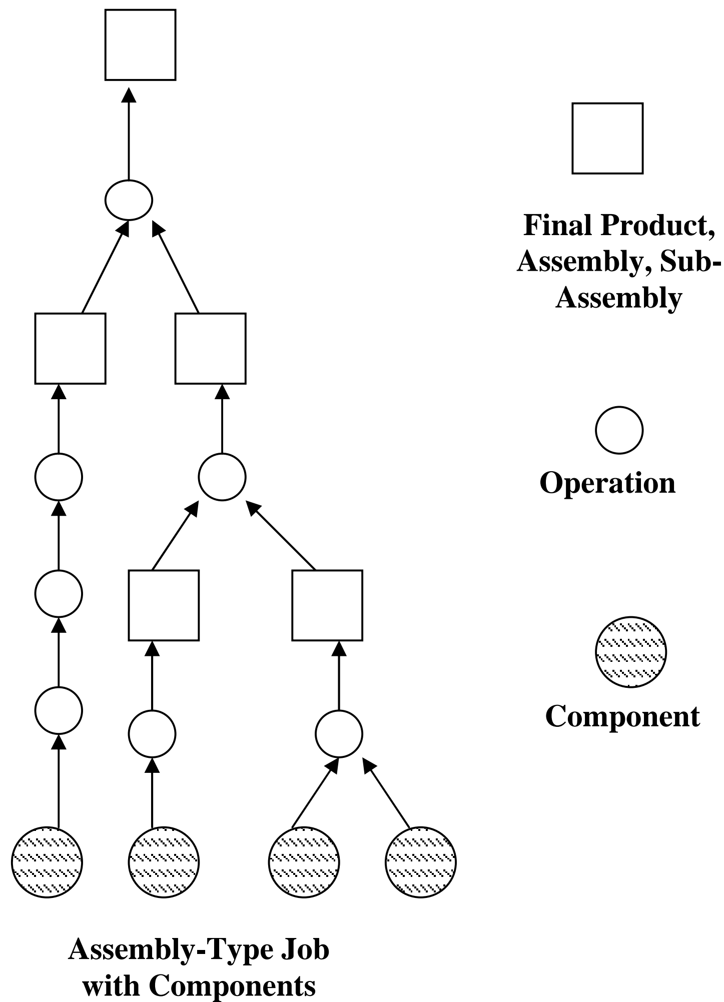


Figure 3.1 Example for product structures of assembly job

However, there are still some assumptions beyond our assembly job shop models. The first assumption is that all jobs are ready to process in the beginning and processing time is deterministic. Second, the purchase order lead time is greater than average customer order lead time. Otherwise, manufacturers could procure the necessary quantities of components after they accepted orders. Additionally, material requirements are assumed to be procured in advance based on forecasts which show as scheduled

receipts for raw materials in the assembly job shop model. If the actual demand is greater than the forecast demand, a material shortage will occur and possibly lead to tardy jobs. Hence, it may result in the loss of customer goodwill. For this reason, the objective of our model is to minimize the total weighted tardiness.

3.2 The extended disjunctive graph and mathematical formulation

The conventional job shop scheduling problem can be represented by using a disjunctive graph. Each node corresponds to an associated operation and each disjunctive arc indicates a possible sequence between two operations which have to be processed in the same type machine. A feasible schedule corresponds to a selection of one disjunctive arc from each pair such that the resulting directed graph is acyclic. This dissertation extends the disjunctive graph to represent the problem of assembly job shop scheduling with components availability. To doing this, a set of nodes representing arrivals of purchase orders (scheduled receipts), called purchase order node (PON), is added into disjunctive graph. Each operation requesting a component is connected to all PONs which carry that type of component using disjunctive (broken) arcs.

It is useful to define the assembly job-shop scheduling problem using a disjunctive graph as shown in Figure 3.2. Let *N* represent the total set of operations with operations *S* and *T* representing the dummy operations "start" and "finish" operations. Let *Z* represent the set of all jobs. Let d_v represent the due date, w_v represent job's priority, and s_v represent the completion time for job v. Let *M* represent the set of all machines, *P* represent the set of all components and *R* represent the set of all purchase orders. Let *A* denote the set of pairs of operations constrained by precedence relations representing the

condition that the sequence of machines is prescribed for each job. Let C denote the set of arcs constrained by precedence relations representing the arrival time for each purchase order. Let E_k represent the set of pairs of operations to be performed on machine k with non-overlapping durations. Hence, each machine can only process one job at any given time. Let p_i , t_i and q_i represent the processing time, start time and requested quantity for component of operation i, respectively. Let B_h represent the set of operations to request component h and F_h represent the set of purchase orders to delver component h. Let a_n and q_n represent the arrival time and delivery quantity of purchase order n. Let Y_h represent the material allocation of components h and $y_{i,n}$ represent the allocation of component from purchase order n to operation i.

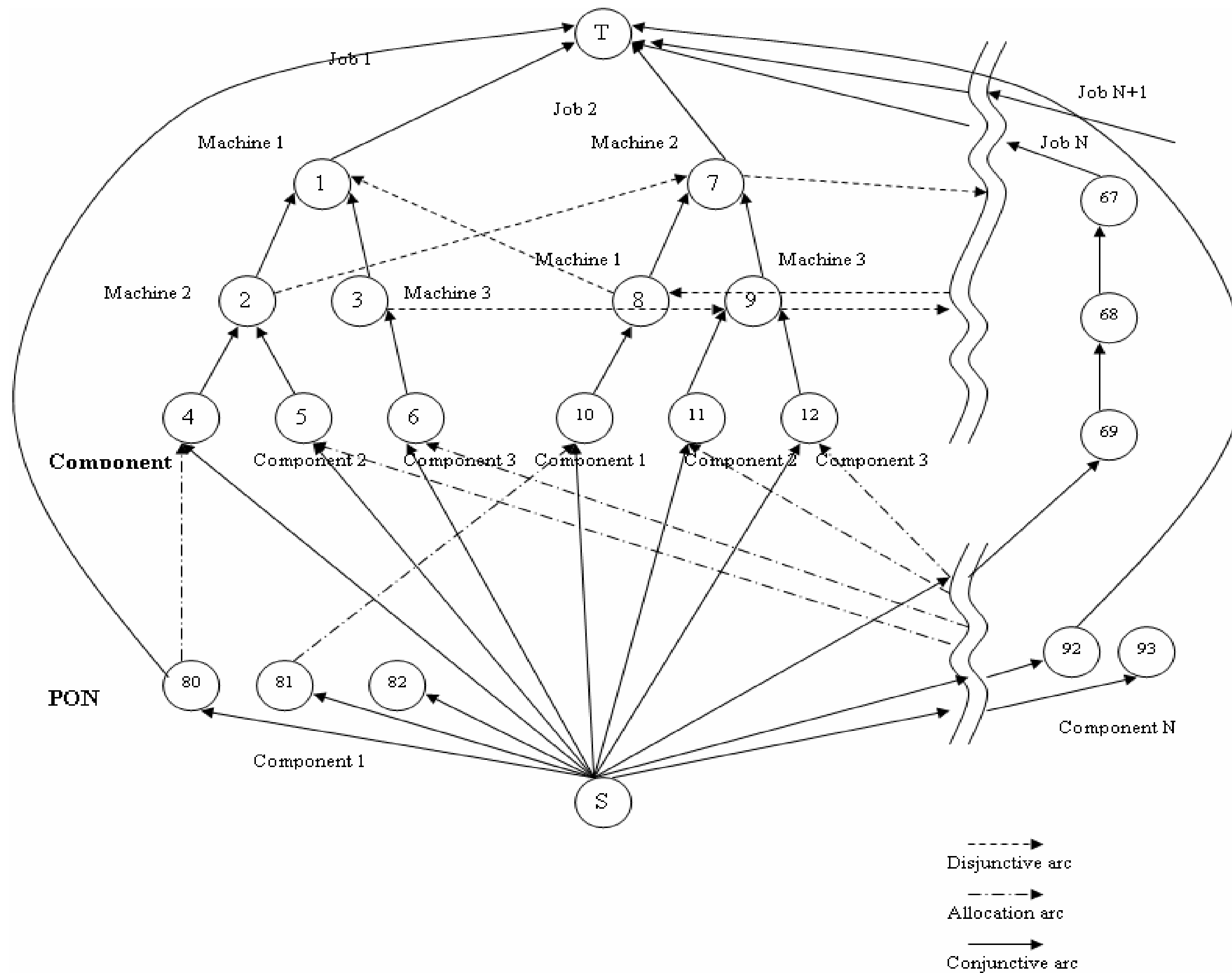


Figure 3.2 A example of a disjunctive graph for assembly job shop scheduling with component availability constraints

A feasible solution to (P^*) is called a schedule. It is helpful to use a disjunctive graph G to represent this problem. Let $G = (N, A, E, R, C, Y)$, where N is the node set for operations, A is the ordinary (conjunctive) arc set for operations, E is the disjunctive arc set for machines, R is the node set for purchase orders, C is the ordinary (conjunctive) arc set for purchase orders and Y is the allocation arc set for components. Each node belonging to set N is an operation. The directed arcs correspond to precedence relations. The pairs of disjunctive arcs correspond to the pairs of operations that are performed on the same machine. Each node belonging to set R is a purchase order. The allocation arcs represent the material allocation of components from purchase orders to operations. The processing time of an operation is shown on an arc. The set of disjunctive arcs, E, consists of cliques E_k, one for each machine k, where $E = \cup(E_k\text{: } k \in M)$. The set of allocation arcs, Y, consists of cliques Y_h, one for each component h, where $Y = \cup(Y_h\text{: } h \in P)$.

The directed graph that results from taking out the disjunctive arcs and the allocation arcs from G is represented by $D = (N, A, R, C)$. S_k is called a machine selection if it includes one and only one member of each disjunctive arc pair of E_k. If a machine selection does not have a directed cycle, it is acyclic. Since each acyclic selection (S_k) corresponds to a particular ordering of the operations associated with machine k, sequencing machine k is equivalent to an acyclic selection in E_k. Similarly, U_h is called a component selection if it includes allocation arcs of Y_h. The union of the machine selection S_k (one in each E_k, $k \in M$) and the component selection U_h (one in each F_h, $h \in P$) is called a complete selection SU, and a partial selection consists of the union taken over a combination of subset (M_0) of M and subset (R_0) of R. An ordinary directed

graph, $D_{SU} = (N, R, A \cup S, C \cup U)$, results from choosing a complete selection SU, that is, putting the machine selection set S in place of the disjunctive arc set E and component selection set U in place of the allocation arcs set Y. The length of the longest path in D_{SU} is equivalent to the makespan of a schedule corresponding to the selection SU. The completion time of job v is equal to the length of the longest path for source node S to top node of jobs v. Lateness of job v can be calculated as the completion time of job v minus the due date of job v. Tardiness of job v is equivalent to the lateness of job v, if the lateness of job v is positive; otherwise, it is zero. Therefore, the goal of this dissertation is to find an acyclic, complete selection $(S \subset E) \cup (U \subset Y)$ that minimizes the weighted tardiness of all orders in the directed graph D_{SU}.

The generic assembly job shop scheduling problem with the objective of minimizing total weighted tardiness can then be formulated as:

(P^*)

$$\min \sum_{v} w_v * \max(s_v - d_v, 0) \quad v \in Z, \text{ ----------------------------(1)}$$

Subject to:

$$t_j - t_i \geq p_i, \qquad (i, j) \in A, \text{ -------------------------(2)}$$

$$t_i \geq 0, \qquad i \in N \cup R \text{ ----------------------(3)}$$

$$t_j - t_i \geq p_i \vee t_i - t_j \geq p_j, \qquad (i, j) \in E_k, k \in M. \text{ --------------(4)}$$

$$\sum_{i} y_{i,n} \leq q_n \qquad i \in B_h, n \in F_h, h \in P \text{ -----------(5)}$$

$$\sum_{n} y_{i,n} = q_i \qquad i \in B_h, n \in F_h, h \in P \text{ -----------(6)}$$

$$t_i \geq a_n \qquad i \in B_h, n \in F_h, h \in P, y_{i,n} = 1 \text{ ---(7)}$$

Table 3.2 Notation of the assembly job shop scheduling problem

N	Set of operations
Z	Set of jobs
d_v	The due date for job v, $v \in Z$
w_v	The priority for job v, $v \in Z$
s_v	The completion time for job v, $v \in Z$
M	The set of all machines
P	The set of all components
A	The set of pairs of operations constrained by precedence relations
C	The set of arcs constrained by the arrival time of purchase orders
E_k	The set of pairs of operations to be performed on machine *k*
p_i	The processing time for operation *i*
t_i	The start time for operation *i*
q_i	The requested quantity for components from operation *i*
B_h	The set of operations to request component h , $h \in P$
F_h	The set of purchase orders to delver component *h,* $h \in P$
R	The set of all purchase orders
a_n	The arrival time for purchase order *n*, $n \in R$
q_n	The delivery quantity for purchase order *n*, $n \in R$
Y_h	The material allocation of components *h*, $h \in P$
$y_{i,n}$	The allocation of component from purchase order *n* to operation *i*.

Equation 1 is the objective function of the problem, which designs to minimize the total weighted tardiness for all jobs. Equation 2 and 3 represent the precedence relation between operations. Equation 4 is the capacity constraint. Equation 5, 6 and 7 are material constraints, which used to allocation components to material requirements.

3.3 Optimization Approach:

In Figure 3.3, the structure of mathematical formulation for multi-level assembly job shop scheduling model consists of three groups of constraints: assembly coordination constraints, material constraints and capacity constraints. The function of the assembly coordination constraints is to assure that the order's precedence constraints are met according to the order's BOM structure. For example, parent's operation PST (Planned Start Time) must be greater than or equal to the child's operation PET (Planned End Time). However, the assembly coordination constraints only consider the assembly coordination between jobs and does not directly provide operation's PST. To obtain the operation's PST accurately, material and capacity constraints are used to compute the operation's PST as described in the following sections. For each job, operation's PST is determined by performing material allocation and capacity schedules with the objective of minimizing total weighted tardiness. Unlike the infinite capacity assumption in the MRP logic, operation's lead time in our model is dynamic and determined by material and capacity constraints. By doing so, operations' PST is not only feasible to either material constraints or capacity constraints, but also is an optimal solution.

In a multi-level assembly job shop scheduling model, orders are collected within a specific time period and processed periodically for scheduling decisions. Scheduling decision involved allocation of limited resources and determination of order PSTs. Consider an assembly job shop scheduling problem over a given planning horizon, T. Assuming that the demand and supply information are known ahead of time for the entire planning horizon. The assembly job shop scheduling problem can be formulated as a mixed integer programming model with the following notations:

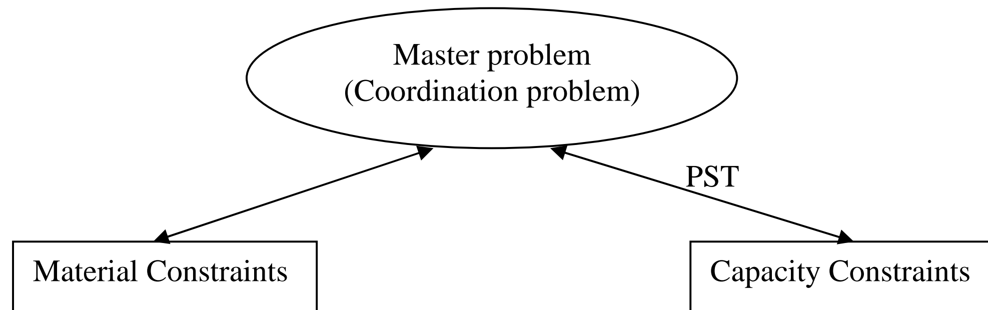


Figure 3.3 Figure 3.3 Mathematical formulations for assembly job shop scheduling problems

Table 3.3 Notation of the assembly job shop scheduling problem

N	Set of operations
Z	Set of jobs
d_v	The due date for job v, $v \in Z$
w_v	The priority for job v, $v \in Z$
s_v	The completion time for job v, $v \in Z$
G	Set of last (Top) operations for job v, $v \in Z$
M	The set of all machines
P	The set of all components
A	The set of pairs of operations constrained by precedence relations
E_k	The set of pairs of operations to be performed on machine k
p_i	The processing time for operation i
t_i	The start time for operation i
q_i	The requested quantity for components from operation i
B_h	The set of operations to request component h , $h \in P$
F_h	The set of purchase orders to delver component h, $h \in P$
R	The set of all purchase orders
a_n	The arrival time for purchase order n, $n \in R$
q_n	The delivery quantity for purchase order n, $n \in R$
Y_h	The material allocation of components h, $h \in P$

$y_{i,n}$ The binary variable for the allocation of component from purchase order n to operation i.

$S_{i,j}$ The binary variable for capacity allocation.

$$(P) \qquad \min \sum_{v} w_v * \max(s_v - d_v, 0) \quad v \in Z, \text{ ----------------------------------(1)}$$

Subject to:

** Assembly coordination constraints*

$$s_v = t_i + p_i \qquad i \in G, , \ v \in Z \text{ ----------------------(2)}$$

$$t_j \geq t_i + p_i \qquad (i, j) \in A, \text{ j is i parent operation ----(3)}$$

$$t_i \geq 0 \qquad i \in N, \text{ ---------------------------------(4)}$$

** Capacity constraints*

$$t_i + p_i \leq t_j + M * S_{i,j} \qquad (i, j) \in E_k, k \in M. \text{ ------------------(5)}$$

$$t_j + p_j \leq t_i + M * (1 - S_{i,j}) \quad (i, j) \in E_k, k \in M. \text{ -------------------(6)}$$

** Material constraints*

$$\sum_i y_{i,n} \leq q_n \qquad i \in B_h, n \in F_h, h \in P \text{ ---------------(7)}$$

$$\sum_n y_{i,n} = q_i \qquad i \in B_h, n \in F_h, h \in P \text{ ---------------(8)}$$

$$t_i \geq a_n * y_{i,n} \qquad i \in B_h, n \in F_h, h \in P \text{ ---------------(9)}$$

In equation 1, the $\sum_v w_v * \max(s_v - d_v, 0)$ shows the sum of weighted tardiness for all jobs as the objective function of the problem. The term $\max(s_v - d_v)$ represents the tardiness of job v. When this term is multiple by job's priority w_v, one obtains the weighted-tardiness for job v. The objective function shown above is to minimize the total weighted tardiness for all jobs within the planning horizon T. Equation 2, 3 and 4 are precedence constraints. They ensure the precedence relations are held between operations. Equation 2 represents the completion time of job v is equal to the top operation of job v. Equation 3 make sure the start time of parent operation is greater than the completion time of child operation. Equation 4 ensures that all start time of operations are positive. Equation 5 and 6 are capacity constraints. They specify that the no operations belonging to the same machine can be processed simultaneously. $S_{i,j}$ is the binary variable. When operation i is the predecessor of operation j, $S_{i,j} = 1$. Equation 7, 8 and 9 are material constraints. They allocate available components to material requirements and guarantee that all operations can not be processed before the arrival of all necessary components. Equation 7 ensures that all material requirements are fulfilled. Equation 8 affirms that the total quantity of components allocates from the specific purchase order to all material requirements is less than the delivered quantity of that purchase order. Equation 9 specifies that the start time of operation is greater than the arrival time of specific purchase order. It will guarantee that operations can only start after the arrival of all necessary components.

3.4 Heuristic approach

The assembly job shop scheduling problem with availability of components is harder to solve than the conventional job shop scheduling problem. In addition to solving a job sequencing problem, one needs to also solve a component allocation problem. In the previous section, we present an optimization approach to solve the multi-level assembly job shop scheduling problem using a mixed-integer programming model. The PST of all jobs will be optimized with respect to material allocation and capacity schedule simultaneously.

Although one can optimally sequence jobs on a single machine within a work-center in a job shop using the branch and bound method, it is not always computationally feasible to optimally sequence jobs when assembly operations are presented. Additionally, the optimization approach requires an excess of computational time and is unable to solve large-scale instances in practice even after relaxation techniques. To overcome these drawbacks, a heuristic is needed to quickly allocate materials to jobs and to determine job sequencing within a machine under precedence relations. To reduce the computational effort, it is necessary to decompose the original problem into numerous subproblems and verify which resource is critical or tightest one. The theory of constraints (Goldratt, 1998) states that critical resources have to be dealt with first in order to improve the overall performance. The concept of TOC had been adopted successfully on the shifting bottleneck algorithm which is one of the popular approaches to solve conventional job shop scheduling problems. This dissertation develops several heuristic approaches, which are based on the concept of the shifting bottleneck algorithm and are capable of solving large-scale assembly job shop scheduling problems, as described in the following section.

Solving both capacity and material constraints optimally are unachievable for large-scale problems in practice. The goal of the heuristic approach is to provide an algorithm which is scalable and provides a solution close to the optimal solution in the desired computational time.

In this chapter, we present three heuristics, which are modified shifting bottleneck procedure (MSBP), efficient shifting bottleneck procedure (ESBP) and rolling horizon procedures (RHP) for the assembly job shop scheduling problem with the availability of components. The idea of modified shifting bottleneck procedure is to decompose the original assembly job shop scheduling problem into several different subproblems which present material plans for components or capacity schedules of machines. The relationship between different jobs is described on the disjunctive graph. Each disjunctive arc represents machine sequences in the appropriate direction and allocation arcs correspond to material allocations. Machine schedules and material allocations are updated to the disjunctive graph iteratively after solving associated subproblems.

The performance of the developed heuristics is also studied in this research with respect to two measures: minimization of the total weighted tardiness and the computational time. The computational results are demonstrated in the next chapter.

3.4.1 Modified shifting bottleneck procedure (MSBP)

A generic modified shifting bottleneck heuristic (MSBP) can be described as follows:

Step 1. Let $M_0 = \varnothing$ and M represent the set of machine types. Let $R_0 = \varnothing$ and R represent the set of component types

Step 2. Remove all the disjunctive arcs and allocation arcs in the graph. Obtain the total weighed tardiness, TWT_{max}, of the graph by finding the longest path from the source node to the top node of every job.

Step 3. For each of the component types in $R - R_0$, solve the material allocation subproblems using the optimization method or dispatching rule.

Step 4. Designate the component type h with the greatest total weighted tardiness $T_{max}(h)$ as the "bottleneck component". If there exists a tie for largest total weighted tardiness, designate the component type h with the average total weighted tardiness $T_{ave}(k)$ as the "bottleneck component". If there is a tie for largest average total weighted tardiness, then designate the first component that is tied for largest average total weighted tardiness the "bottleneck" (an arbitrary rule). Add component type h to the set R_0 and schedule component type h using the solution found in the material allocation subproblem, named MA_h, for component type h.

Step 5. Add the allocation arcs that specify purchase orders to the operations on component type h to the graph. Revise the total weighted tardiness using the equation $TWT_{max}(R_0 \cup h, M_0) = T_{max}(h)$.

Step 6. If $|R_0| = 1$, then local reoptimization is not needed, go to step 3. If $|R_0| > 1$, then resequence the component types in R_0 one at a time until a full cycle goes by without any improvement. After resequencing a component, say component type l, put the component type back into R_0 in the same place that it was removed from and add the allocation arcs that specify the orders to the operations on component type l to the graph. Adjust the total weighed tardiness: $TWT_{max}(R_0 \cup l, M_0) = T_{max}(l)$. If $|R_0| = |R|$, then go to step 7. Otherwise, go to step 3.

Step 7. For each of the machine types in $M - M_0$, solve the machine allocation and sequence subproblems, named CA_m, using the optimization method or dispatching rule.

Step 8. Designate the machine type k with the greatest maximum total weighted tardiness $T_{max}(k)$ as the "bottleneck machine". If there exists a tie for largest maximum total weighted tardiness, designate the machine type k with the greatest average lateness $T_{ave}(k)$ as the "bottleneck machine". If there is a tie for largest average total weighted tardiness, then designate the first machine that is tied for largest average total weighted tardiness the "bottleneck" (an arbitrary rule). Add machine type k to the set M_0 and schedule machine type k using the solution found machine allocation subproblem for machine type k.

Step 9. Add the disjunctive arcs that specify the order of the operations on machine type k to the graph. Revise the total weighted tardiness using the equation $TWT_{max}(R_{0,} M_0 \cup k) = T_{max}(k)$.

Step 10. Resequence the component types in R_0 one at a time until a full cycle goes by without any improvement. After resequencing a component, say component type l, put the component type back into R_0 in the same place that it was removed from and add the allocation arcs that specify the orders to the operations on component type l to the graph. Adjust the total weighed tardiness: $TWT_{max}(R_0 \cup l, M_0) = T_{max}(l)$. Then go to step 11.

Step 11. If $|M_0| = 1$, then local reoptimization is not needed, go to step 7. If $|M_0| > 1$, then resequence the machine types in M_0 one at a time until a full cycle goes by without any improvement. After resequencing a machine, say machine type l, put the machine type back into M_0 in the same place that it was removed from and add the disjunctive arcs that specify the order of the operations on machine type l to the graph. Adjust the total weighted tardiness: $TWT_{max}(R_{0,} M_0 \cup l) = T_{max}(l)$. If $|M_0| = |M|$, then terminate the procedure. Otherwise, go to step 7.

There are four modified shifting bottleneck procedures (MSBP) which are developed to solve assembly job shop scheduling problems. There methods are SB1, SB2, SB3 and SB4. The difference between these MSBP methods is the degree of solution

accuracy obtained from subproblems. The four MSBP methods are summarized in Table 3.4.

Table 3.4 The description of MSBP methods

	SB1	**SB2**	**SB3**	**SB4**
Material allocation subproblem (MA)	Optimization method **(MA1)**	Optimization – longest path implementation **(MA2)**	Dispatching rule	Optimization method **(MA1)**
Capacity allocation subproblem (CA)	Optimization method **(CA1)**	Optimization – longest path implementation **(CA2)**	Optimization method **(CA1)**	Dispatching rule

SB1 uses the optimization method to solve both material and capacity subproblems. The material allocation subproblem and capacity schedule subproblem in SB1 can be formulated as MA1 and CA1. However, the computational time requiring to solve SB1 may still increase exponentially as the size of subproblems increase. To reduce the complexity of SB1, the longest path algorithm is applied to represent the precedence constraints in SB2. By doing so, the total number of constraints is expected to reduce to a reasonable level. The material allocation subproblem in SB2 can be formulated as MA2 and the capacity allocation subproblem in SB2 can be formulated as CA2. The SB3 uses the dispatching rule to solve material allocation subproblems and apply optimization method, CA1, which use the same mathematical formulation as SB1, to solve capacity

subproblems. The SB4 uses the dispatching rule to solve capacity allocation subproblems and apply optimization method, MA1, to solve material allocation subproblems.

Material allocation subproblem (MA1):

For each $h \in R \setminus R_0$

$$\min \sum_{v} w_v * \max(s_v - d_v, 0) \quad \text{v} \in Z, \quad \text{-----------------------(1)}$$

$$t_j - t_i \geq p_i, \quad (i, j) \in \cup(S_p : p \in M_0) \cup A \quad \text{-----(2)}$$

$$t_i \geq 0, \quad i \in N \cup R \quad \text{-----------------------(3)}$$

$$\sum_{i} y_{i,n} \leq q_n \quad (i,n) \in \cup(U_c : c \in R_0) \cup C \quad \text{-------(4)}$$

$$\sum_{n} y_{i,n} = q_i \quad (i,n) \in \cup(U_c : c \in R_0) \cup C \quad \text{--------(5)}$$

$$t_i \geq a_n \quad (i,n) \in \cup(U_c : c \in R_0) \cup C \quad \text{--------(6)}$$

$$\sum_{i} y_{i,n} \leq q_n \quad (i,n) \in Y_h, h \in R \setminus R_0 \quad \text{----------------(7)}$$

$$\sum_{n} y_{i,n} = q_i \quad (i,n) \in Y_h, h \in R \setminus R_0 \quad \text{----------------(8)}$$

$$t_i \geq a_n \vee t_i \geq 0 \quad (i,n) \in Y_h, h \in R \setminus R_0 \quad \text{----------------(9)}$$

Equation 1 is the objective function of the problem, which designs to minimize the total weighted tardiness for all jobs. Equation 2 and 3 represent the operation sequences from the scheduled machines at set M_0. Equation 4, 5 and 6 represent the material allocations from scheduled components at set R_0. Equation 7, 8 and 9 are

material constraints, which used to allocation components to material requirements for the component type h.

Capacity allocation subproblem (CA1):

For each $k \in M \setminus M_0$

$$\min \sum_{v} w_v * \max(s_v - d_v, 0) \quad v \in Z, \quad \text{-----------------------(1)}$$

$$t_j - t_i \geq p_i, \quad (i, j) \in \cup(S_p : p \in M_0) \cup A \quad \text{-----(2)}$$

$$t_i \geq 0, \quad i \in N \cup R \quad \text{----------------------------(3)}$$

$$t_j - t_i \geq p_i \vee t_i - t_j \geq p_j, \quad (i, j) \in E_k,\ k \in M \setminus M_0 \quad \text{-------------(4)}$$

$$\sum_{i} y_{i,n} \leq q_n \quad (i, n) \in \cup(U_c : c \in R_0) \cup C \quad \text{-------(5)}$$

$$\sum_{n} y_{i,n} = q_i \quad (i, n) \in \cup(U_c : c \in R_0) \cup C \quad \text{--------(6)}$$

$$t_i \geq a_n \quad (i, n) \in \cup(U_c : c \in R_0) \cup C \quad \text{----------(7)}$$

Equation 1 is the objective function used to minimize the total weighted tardiness for all jobs. Equation 2 and 3 represent the operation sequences from the scheduled machines at set M_0. Equation 4 are capacity constraints, which used to decide the machine sequence for the specific machine type k. Equation 5, 6 and 7 represent the material allocations from scheduled components at set R_0.

Material allocation subproblem (MA2):

$$\min \sum_{v} w_v * \max(s_v - d_v, 0) \quad v \in Z, \text{ --------------------------------(1)}$$

$$s_v = t_i + l(i,v) \vee s_v = l(S,v) \quad i \in B_h, h \in R \setminus R_0 \text{ --------------------(2)}$$

$$\sum_i y_{i,n} \le q_n \quad (i,n) \in \cup(U_c : c \in R_0) \cup C \text{ ---------(3)}$$

$$\sum_n y_{i,n} = q_i \quad (i,n) \in \cup(U_c : c \in R_0) \cup C \text{ ---------(4)}$$

$$t_i \ge a_n \quad (i,n) \in \cup(U_c : c \in R_0) \cup C \text{ ---------(5)}$$

$$\sum_i y_{i,n} \le q_n \quad (i,n) \in Y_h, h \in R \setminus R_0 \text{ ----------------(6)}$$

$$\sum_n y_{i,n} = q_i \quad (i,n) \in Y_h, h \in R \setminus R_0 \text{ ----------------(7)}$$

$$t_i \ge a_n \vee t_i \ge 0 \quad (i,n) \in Y_h, h \in R \setminus R_0 \text{ ----------------(8)}$$

Where $l(i, j)$ is the value of longest path from operation *i* to operation *j*.

Equation 1 is the objective function to minimize the total weighted tardiness for all jobs. Equation 2 represents the completion time of job *v*, which is calculated from the start time of operation *i* plus the value of longest path from operation *i* to job *v* or the value of longest path from source node to job *v*. Equation 3, 4 and 5 represent the material allocations from scheduled components at set R_0. Equation 6, 7 and 8 are material constraints, which used to allocation components to material requirements for the component type *h*. In MA2, there are no capacity constraint equations because the delay from capacity conflict is added in the longest path calculation.

Capacity allocation subproblem (CA2):

$$\min \sum_{v} w_v * \max(s_v - d_v, 0) \quad v \in Z, \text{-------------------------------}(1)$$

$$s_v = t_i + l(i,v) \vee s_v = l(S,v) \quad i \in N^*, k \in M \setminus M_0 \quad \text{-------------}(2)$$

$$t_j - t_i \geq p_i, \quad (i, j) \in \cup(S_p : p \in M_0) \cup A \quad \text{----}(3)$$

$$t_i \geq l(S,i), \quad i \in N^*, k \in M \setminus M_0 \text{----------------}(4)$$

$$t_j - t_i \geq p_i \vee t_i - t_j \geq p_j, \quad (i, j) \in E_k,\ k \in M \setminus M_0 \text{------------}(5)$$

Where N* is the set of operations to be processed on machine k and $l(i, j)$ is the value of longest path from operation i to operation j.

Equation 1 is the objective function designs to minimize the total weighted tardiness for all jobs. Equation 2 represents the completion time of job v, which is calculated from the start time of operation i plus the value of longest path from operation i to job v or the value of longest path from source node to job v. Equation 3 represents the machines sequences from scheduled machines at set M_0. Equation 4 ensures that all start time of operations are greater than the value of longest path from the source node to itself. Equation 5 represents the capacity constraints used to decide the machine sequence for machine type k.

3.4.2 Efficient shifting bottleneck procedure (ESBP)

However, it may be still hard to solve subproblems optimally for large instances even after decomposition procedures. Uzsoy and Wang (2000) discovered that most of the computational effect in a shifting bottleneck procedure is dedicated to solving subproblems during re-optimization procedure. In general, the complexity increases as number of jobs increases and the number of machine types increases, as there are more subproblems to solve. By reducing the number of subproblems or denying efficient procedures to solve subproblems, the computational time of the shifting bottleneck procedure can be reduced significantly. Demirkol et al. (1997) concluded that unlike subproblems and re-optimization procedures, bottleneck machine selection criteria didn't have any significant effect on solution quality and computational time.

Based on the finding from Demirkol et al. (1997), the shifting bottleneck procedure with a limited re-optimization procedure in the last iteration reduces computational time without a significant sacrifice in solution quality. If a reasonable material plan or job sequence on machines can produce by using a dispatching rule, then a good solution can be obtained and improved using a re-optimization procedure. We propose an efficient shifting bottleneck procedure (ESBP) for the assembly job shop scheduling problem with the availability of components, where bottleneck machine or component selection procedure is avoided and re-optimization procedure is constrained to few iterations. Hence, the total number of subproblems is expected to decrease in ESBP.

We introduce the efficient shifting bottleneck procedure which aims to reduce the total number of subproblems to be solved in a shifting bottleneck procedure, while keeping the solution quality at an acceptable level. Unlike modified shifting bottleneck

procedure, efficient shifting bottleneck procedure does not solve the all subproblems, it designs to obtain feasible solutions of the limited number of subproblems within reasonable computational time by using efficient algorithms or dispatching rules which are suited for assembly job shop scheduling problems in practice.

A generic efficient shifting bottleneck procedure (ESBP) can be described as follows:

Step 1. Choose an efficient dispatching rule or any other assembly job shop scheduling algorithm to obtain material plan of each component, job sequence and machine schedule of each machine type as the initial solution.

Step 2. Use re-optimization procedure to improve the initial solution. The material subproblems and capacity subproblems can be solved by using a dispatching rule which is more efficient to obtain solutions.

There are three efficient shifting bottleneck procedures (ESBP) introduced to solve assembly job shop scheduling problems, which are ESBP1, ESBP2 and ESBP3. The difference between these ESBP methods is the degree of solution accuracy. ESBP1 uses the optimization method to solve both material and capacity subproblems. The ESBP2 uses the dispatching rule to solve material allocation subproblems and apply optimization method, which has the same formulation as SB1, to solve capacity subproblems. The ESBP3 uses the dispatching rule to solve capacity subproblems and

apply optimization method to solve material subproblems. The three ESBP methods are summarized in Table 3.5.

Table 3.5 The description of ESBP methods

	ESBP1	**ESBP2**	**ESBP3**
Initial Solution	Dispatching rule	Dispatching rule	Dispatching rule
Material allocation subproblem (MA)	Optimization method **(MA1)**	Dispatching rule	Optimization method **(MA1)**
Capacity allocation subproblem (CA)	Optimization method **(CA1)**	Optimization method **(CA1)**	Dispatching rule

3.4.3 Rolling horizon procedure (RHP)

Although ESBP reduces the total number of subproblems requiring to solve the assembly job shop scheduling problem, these subproblems are still NP-hard problems in nature. It is necessary to reduce the complex of subproblems to capable of solving large size problems within reasonable computational times for practice uses. We introduce the ESBP with implementation of the rolling horizon procedure to solve subproblems, so-called RHP. The procedures of RHP is similar to ESBP, the only difference between them is the way to solve subproblems. In RHP, subproblems are solved by the rolling

horizon procedure. The rolling horizon procedure decomposes the subproblem into numerous time windows that can be optimized independently within a reasonable computational effort. There are several algorithms to divide the problems, which included the number of operations and the length of time window. In RHP, the subproblem is divided into time windows by the fixed number of operations, which is similar to the rule developed by Ovacik and Uzsoy (1995). In their works, operations are chosen based on earliest due date rule. However, RHP uses the dispatching rule to obtain initial solutions of the subproblem. Then, operations are separated into different time windows based on their starting time from the initial solutions. Afterward, the sequence of operations belonging to the first time window is optimized and the partial sequence is frozen. The operations belonging to second time window are added into the problem. The sequence of operations is optimized and the partial sequences are scheduled according to solutions. This procedure will repeat until all operations are scheduled. By restricting the size of subproblems, it does not only decrease the computational effort of subproblems, but also reduces the complexity of the overall problem as well. Furthermore, the complexity of this approach is expected to be a function of the time window's size.

A rolling horizon procedure (RHP) can be described as follows:

Step 1. Choose an efficient dispatching rule or any other assembly job shop scheduling algorithm to obtain material plan of each component, job sequence and machine allocation of each machine as the initial solution.

Step 2. Use re-optimization procedure to improve the initial solution. The material subproblems and capacity subproblems can be solved by using a dispatching rule or an optimization method with the implementation of rolling horizon procedure to reduce the complexity of test problems.

The procedure of RHP is described as follows. Let n denote the number of operations per time window and denote p the number of overlapping operations per time window. Let V(t) presents the set of unscheduled operations, S(t) presents the set of scheduled operations and K(t) presents the set of candidate operations for current time window.

Step 1. Choose first O=min(n-|K(t)|, |V(t)|) operations from the set of unscheduled operations V(t) for machine k and add these operations to the set of candidate operations K(t), which is belonging to current subproblem. Let V(t) = V(t) – O(t) and K(t) = K(t) U O(t).

Step 2. Optimize the sequence of operations in K(t). Select first L= min(n-p, |K(t)|) operations from K(t) and schedule them according to the partial sequence. Mark the status of these operations to "Fixed" and add these operations to S(t). Let S(t)= S(t)U L(t). For the last O=min(p, n-L) operations in K(t). Mark the status of operations in O(t) to "Open".

Step 3. If K(t) = empty set , Stop RHP; otherwise, go to Step1.

3.5 Dispatching rules

The assembly job shop scheduling problem is NP-hard problem in both theoretical research and real world practice. A simple way to schedule all jobs on the machines in practice is dispatching rules which received great attention from researchers in past decades. Many different dispatching rules have been developed in both research and practice. In general, dispatching rules sort all jobs according to specific priority indexes when they enter the queue of machines or materials. A priority index is based on job characteristics, machine attributes or current shop conditions, such as job processing time and the order of arrival time at a machine. Because dispatching rules only rely on current shop status and local information, they are myopic in nature. Without considering the future jobs in the global view, we do not expect dispatching rules to work well in the long run. However, scheduling problems are complex and the excess computational time on optimization methods is not suitable in realistic production systems. Additionally the options for scheduling realistic systems are still very limited, dispatching rules continues to find extensive use in industry (Hopp and Spearman, 2000).

Although dispatching rules can not provide the superior solutions, they may perform better on a specific objective. The simplest dispatching rule is first come first served (FCFS). The FCFS rule sorts the jobs in increasing order which they arrive at a machine disregarding the priority and the processing time of jobs. To improve due date performance, earliest due date first (EDD) is wild used. The EDD rule processes the job in the increasing order of job's due date which could be set internally or externally. To reduce congestion or total completion time, shortest processing time first (SPT) may be a

good candidate. The SPT rule chooses the job to process in the increasing order of the processing time of the jobs.

In an assembly job shop, jobs consist of several levels in the BOM. Assembly of components may occur all the time when jobs are being processed through the shop. Failure to provide sufficient amount of components at the time of assembly will result in delays of shipments. In MRP systems, assembly due dates are provided for components which are being assembled and can be used as the reference for progress of jobs towards completion. Baker and Kanet (1983) found that the use of operation due dates as progress milestones improves the due date-oriented sequencing rules using only job due dates in a single stage job shop.

In this dissertation, six dispatching rules are used to evaluate the performance of MSBP and ESBP. Because the objective of these models is to minimize the total weighted tardiness for the assembly job shop scheduling problem, EDD rule is chosen as a candidate for performance comparison. To consider the impact of the priority of the jobs on the objective, WEDD is developed as a function of due dates and priorities. The WEDD rule chooses the job to process in the increasing order of priority and break ties by due date. The objective of minimizing total weighted tardiness is more difficult to solve than the one of minimizing maximum tardiness. The Apparent Tardiness Cost (ATC) rule is a function of WSPT and MS. It is designed to minimize total weighted tardiness on a single machine with consideration of release dates and due dates. The dispatching rules used in this dissertation are described as follows:

1. EDD (Earliest due date first): The EDD rule will select a job with the earliest due date to be processed next when machines are idle. It is designed to minimize the maximum lateness among jobs for a single machine.
2. WEDD (Weighted Earliest due date first): When machines are idle, the WEDD rule will choose a job with the highest priority to be processed first and break ties by earliest due date. It is designed to minimize the total weighted tardiness among jobs for a single machine.
3. WTC (Weighted tardiness cost): When a machine becomes free, the WTC will choose the jobs with highest ranking index to be scheduled next. The index is described as follows: $A_j(t) = TNOW + L(j,T) - d_j$. If $A_j(t)$ is negative, $I_j(t) = A_j(t)$, else, $I_j(t) = A_j(t) * w_j$. It is designed to minimize total weighted tardiness on a single machine with consideration of job's priority and due dates.
4. ATC (Apparent tardiness cost): When a machine becomes free, the ATC will choose the jobs with highest ranking index to be scheduled next. The index is described as following: $I_j(t) = \frac{w_j}{p_j} \exp(-\frac{\max(d_j - p_j - t, 0)}{K\overline{p}})$, where K is the scaling parameter, which can be determined empirically, and p is the average of the processing times of the remaining jobs. It is designed to minimize total weighted tardiness on a single machine with consideration of release dates and due dates. Additionally, if K is very large, the ATC rule reduces to the WSPT rule. If K is very small and there are no overdue jobs, the rule reduces to the MS rule.

5. MWTC (Modified weighted tardiness cost): When a machine becomes free, the MWTC will choose the jobs with highest ranking index to be scheduled next. The index is described as follows: $A_j(t) = TNOW + L(j,T) - d_j$. For the material plans, if $A_j(t)$ is negative, $I_j(t) = A_j(t)$, else, $I_j(t) = A_j(t) * w_j$. For the capacity schedules, if $A_j(t)$ is negative, $I_j(t) = A_j(t)$, else, $I_j(t) = A_j(t) / p_j$. It is designed to minimize total weighted tardiness on a single machine with consideration of release dates and due dates.
6. WSPT (Weighted shortest processing time first): When a machine becomes free, the WSPT will choose the jobs with highest priority over processing time to be scheduled next. It tends to minimize the total weighted completion times among jobs for a single machine.

Table 3.6 The summary of dispatching rules

Rule	Capacity Allocation Index	Material Allocation Index
EDD (Earliest due date first)	Earliest d_j	Earliest d_j
WEDD (Weighted Earliest due date first)	Smallest $w_j * d_j$	Smallest $w_j * d_j$
WTC (Weighted tardiness cost):	If $A_j(t) \le 0, I_j(t) = A_j(t)$, else, $I_j(t) = A_j(t) * w_j$	If $A_j(t) \le 0, I_j(t) = A_j(t)$, else, $I_j(t) = A_j(t) * w_j$
ATC (Apparent tardiness cost)	$I_j(t) = \frac{w_j}{p_j} \exp(-\frac{\max(d_j - p_j - t, 0)}{K\bar{p}})$	Earliest d_j
MWTC (Modified weighted tardiness cost)	If $A_j(t) \le 0$, $I_j(t) = A_j(t)$, else, $I_j(t) = A_j(t) * w_j$.	If $A_j(t) \le 0$, $I_j(t) = A_j(t)$, else, $I_j(t) = A_j(t) / p_j$
WSPT (Weighted shortest processing time first)	Highest $\frac{w_j}{p_j}$	Earliest d_j

* $A_j(t) = TNOW + L(j, T) - d_j$

Chapter 4. Computational Results

In this dissertation, several modified shifting bottleneck procedures (MSBP), efficient shifting bottleneck procedures (ESBP) and rolling horizon procedure (RHP) are developed to solve assembly job shop scheduling problems with the availability of components in the make-to-order (MTO) environment. We also investigated the performance of all proposed methods and their impact on the performance of a production system. There are several dispatching rules developed in this research and are used as performance comparisons in our experiments. The optimization approach, which is MIP model, is first evaluated for the performance. The computational results show how the complexity of the MIP model based on the different number of jobs with limited maximum CPU time setting. Second, the performance of heuristic approaches, which are based on shifting bottleneck procedures, is studied with respect to two measures: minimization of the total weighted tardiness and the computational time. The goal is to identify an approach which is scalable and provides solutions close to solutions obtained from optimization methods with the objective of the minimization of the total weighted tardiness measure. Third, we study the difference of the performance between dispatching rules and heuristics. Fourth, the effect of the implementation of rolling horizon procedure to solve subproblems is also investigated. The results demonstrate how different shifting bottleneck algorithms can help manufacturers improve the performance of an assembly job shop.

4.1 Design of Experiments

The manufacturing models used in this experiment acts as a real-world job-shop production environment. There are ten components consumed by ten assembly items and ten end items. There are also ten machines in this manufacturing model and the processing time of an operation is selected from a uniform distribution between 5 and 30. Each job has maximum levels of BOM up to 3 and the total number of assembly items per BOM is equal to 3. For each operation, there is a 40% probability to consume components which have to be procured from outside vendors in advance. Our experiment consists of several factors and varying number of levels for each factor. The number of jobs is set at 5, 10, 15, 20, 25, 30 and 35. Tightness of supply levels is assumed to be medium. A medium level tightness of supply means that the inter-arrival time for each components follows a uniform distribution with mean equals to 10 and standard deviation equals to 5. Tightness of supply (TOS) value is a major measure for material constraints. It will determine by demand and supply patterns. In our experiments, we control the TOS value by changing the variability of supply, instead of supply mean. For example, two supply patterns may have the same inter-arrival rate, but different variability. The supply pattern with lower variability will generate less tardy orders than the one with higher variability. We can simply increase or decrease the variability of supply inter-arrival time until reaching the desirable level of TOS and still maintain the same inter-arrival rate for both demand and supply.

Each job is randomly given a priority value set at 1, 2 and 4 with probability equal to 70%, 20% and 10%. A job is classified as a high priority order if the priority value is 4. A medium priority job if the priority value is 2 and a low priority order otherwise. A 10%

high priority ratio in our experiments means that 10% of orders are high priority jobs. We define "the job penalty ratio" as the ratio of penalty factor of high priority jobs to the penalty factor of low priority jobs. We tested job penalty ratios of 2, referring them as medium in our experiments. A job's due date is calculated as a function of total processing time and a due date factor. The job's due-date is quoted as job's arrival time plus the estimated flow time as the following equation.

Due Date = Job arrival time + Estimated flow time

Job arrival time = 0

*Estimated flow time = Uniform{ (2.5*Total Processing Time)*(1-Due date factor), 0.3*Total Processing time}*

It is noted that our due-date setting method treats all orders in the same way with no regard to weather they are high priority or low priority jobs. The only difference between a high priority job and a low priority job is job penalty ratio. This means that late high priority jobs will generate more penalty than late low priority jobs. A similar logic is used by Malhotra et al (1994).

The test problems were generated randomly by varying a set of parameters as described above. For each instance, the performance measures are computed from 5 different random seeds in order to increase the accuracy of the experiments. More than 5 optimization method settings are tested in selected test problems and over 30 different heuristics settings and dispatching rules are evaluated for each test problem. Hence, there are 1105 randomly test problems in our experiments. The description of parameters used in test problems is illustrated as shown in Table 4.1. To make fair comparisons between all proposed methods, MIP methods, SB heuristics and dispatching rules are all coded

using Xpress-MP (Mosel). The experiments are performed under the Windows platform with Pentium 4 2.8 GHz processor (single core processor).

Table 4.1 Experimental design for randomly generated test problems

Factors	Levels	Number of levels
Number of jobs	5, 10, 15, 20, 25, 30, 35	7
Number of machines	10	1
Number of items	10	1
Order Priority (Order penalty ratio)	Low(70%, 1), Medium(20%, 2), High(10%, 4)	1
Due Date Tightness	Medium (0.3)	1
Tightness of supply	Medium	1
Number of BOM level	Medium (3)	1
Number of items per assembly level	Medium (3)	1
Methods	MIP, SB1, SB2, SB3, SB4, ESBP1, ESBP2, ESBP3	8
Total test problems		

4.2 Performance Measures and Computational Results

Two measures, total weighted tardiness cost and computational time, are used to study the performance of the optimization methods and heuristic approaches. The

primary performance measure is the total weighted tardiness, which distinguishes between high and low priority jobs. It measures customer service levels. For measure of scalability, the computational time is an important indicator.

We have adopted a relative measure scale instead of an absolute one while discussing weighted tardiness cost and computational time. A value of 1.0 in computational time means that the corresponding approach had the least computational time compared to the other approaches tested in the study. Similarly, a value of 1.3 in weighted tardiness cost for an approach means that the weighted tardiness cost using that approach had a value 30% above the best cost value.

We used performance measures similar to Demirkol et al. (1997). Since the subproblems can not generally be solved to optimality by dispatch rules, we replaced LB(I), denoted a lower bound on the optimal total weighted tardiness value of instance I, by TWT(SB, I) and CPU(SB, I).

In this experiment, TWT(H) and CPU(H) are used to evaluate the performance of heuristics H. For a given problem class S, let TWT(H) be the total weighted tardiness value obtained by heuristic H divided by total weighted tardiness value of best heuristic in that class.

$$\text{TWT (H)} = \frac{\sum_{I \in S} TWT(H, I)}{\sum_{I \in S} TWT(SB, I)}$$

There are two performance measures of CPU time. CPU(H) shows the performance of heuristic H compared to heuristic SB in a given problem class S. For a given problem class S, let CPU(H) as the CPU time obtained by heuristic H divided by CPU time of the best heuristic SB in that class.

$$\text{CPU (H)} = \frac{\sum_{I \in S} CPU(H, I)}{\sum_{I \in S} CPU(SB, I)}$$

.

The value of TWT (H) and CPU(H) indicated the performance of specific heuristic H on a given problem class S compared to best heuristic SB in that class. If TWT(H) or CPU(H) is equal to 1, it means the specific heuristic H provides best solutions or computational time for all test problems.

4.3 Computational results for optimization methods

In this section, we present the one optimization method to solve assembly job shop scheduling problems, which are the mixed integer programming model (MIP), and two heuristics for comparison purposes, which are modified shifting bottleneck procedure (SB1) and the modified shifting bottleneck procedure with longest path implementation (SB2). In the MIP model, production constraints are formulated as a mixed-integer programming model without any relaxation constraints. In this case, the MIP model requires excess computing power to obtain the optimal solution and can not solve large-scale assembly job shop scheduling problems in practice. To overcome this drawback, we developed an SB1 method by decomposing the original assembly job shop schedule problem to numerous subproblems and solving them iteratively. Hence, it requires less computational time and provides the solutions closed to the solutions of MIP. Additionally, SB2 is introduced to save the computational time in each subproblem by using the longest path implementation to represent the precedence constraints. In this

section, the experiments are designed to assess the performance of the MIP, the SB1 and the SB2 as shown in Table 4.2. The goal is to evaluate the difference of solutions between the MIP, the SB1 and SB2 based on computational time and solution quality.

Table 4.2 Experimental design for randomly generated test problems – Optimization methods

Factors	Levels	Number of levels
Number of jobs	5, 10, 15	5
Number of machines	10	1
Number of items	10	1
Methods	MIP (100), MIP (500), MIP (1000), MIP (3000), MIP (5000), SB1, SB2	7
Total test problems		175

* MIP(100): MIP model with maximum CPU time setting equals to 100 seconds.
* MIP(500): MIP model with maximum CPU time setting equals to 500 seconds.
* MIP(1000): MIP model with maximum CPU time setting equals to 1000 seconds.
* MIP(3000): MIP model with maximum CPU time setting equals to 3000 seconds.
* MIP(5000): MIP model with maximum CPU time setting equals to 5000 seconds.
* SB1: Shifting bottleneck algorithm with maximum CPU time setting equals to 20 seconds.
* SB2: Shifting bottleneck algorithm with longest path implementation and maximum CPU time setting equals to 20 seconds.

In Table 4.3, CPU(H) ratio of MIP methods increase dramatically as the MaxCPUTime increases. It is due to the nature of mixing-integer programming. However, TWT(H) ratio of MIP (5000) only improve by 17% when CPU(H) ratio raises more than 16 times compared to MIP (100). Moreover, MIP (5000), which has the MaxCPUTime setting to 5000 seconds, only found three more integer solutions than MIP (100) does on

average. Out of 15 test problems, it provides only two more better solutions than MIP (100) does. It indicates that increasing MaxCPUTime setting for MIP method does not seem to improve the solution quality that much and only increases the computation effects as shown in Figure 4.1. Thus, an algorithm which can respond to both material plans and capacity schedules faster than the MIP method is needed to solve assembly job shop scheduling problems efficiently. In this experiment, two SB heuristics, SB1 and SB2, have been used to evaluate the performance of decomposition methodology on the solution quality and computational time. SB1 is based on the concept of shifting bottleneck procedures. It breakdowns the assembly job shop scheduling problems into numerous subproblems, which represent different material plans and capacity schedules, and solve each separately. SB2 used the same theory to solve assembly job shop scheduling problems but tries to reduce more computational efforts by the implementation of longest path algorithm to represent the precedence relation between operations in each subproblem. The TWT(H) ratio of SB1 and SB2 is better than all MIP methods in test problems consistently as shown in Table 4.3. In addition, SB1 and SB2 require less computational time than MIP methods by 93% on average and provide better solutions than MIP method by 17% in overall test problems. It shows that the benefit of decomposition methodology does not only reduce the computational time but also improves the solution quality in the case that optimal solutions can not be obtained from MIP method in reasonable computational times. Moreover, SB2 requires less computational time than SB1 because of the reduction on the total number of variables and constraints in each subproblem. But it also sacrifices the solution quality by 9% compared to SB1, as shown in Table 4.3. In Figure 4.1, it easily can see that the two

shifting bottleneck procedures are superior to all MIP methods on all performance measures.

Table 4.3 Computational results for optimization methods

Method	MIP	MIP	MIP	MIP	MIP	SB1	SB2
MaxCPUTime*	100	500	1000	3000	5000	20	20
CPU(H) –Mean	4.510	19.739	35.904	100.17	164.34	5.592	2.839
TWT(H)-Mean	1.656	1.449	1.404	1.388	1.380	1.158	1.261
TWT(H)-SD	0.315	0.398	0.365	0.352	0.348	0.226	0.225
Number of integer found	7.33	8.73	9.40	9.86	10.00	-	-
Number of best solutions	4	5	5	6	6	11	4
Percentage of best solutions	16%	20%	20%	24%	24%	44%	16%

*** MaxCPUTime: The maximum cpu time allowed for each subproblem. The parameter is controlled by "XPRS_MAXTIME" parameter in Xpress-MP.**

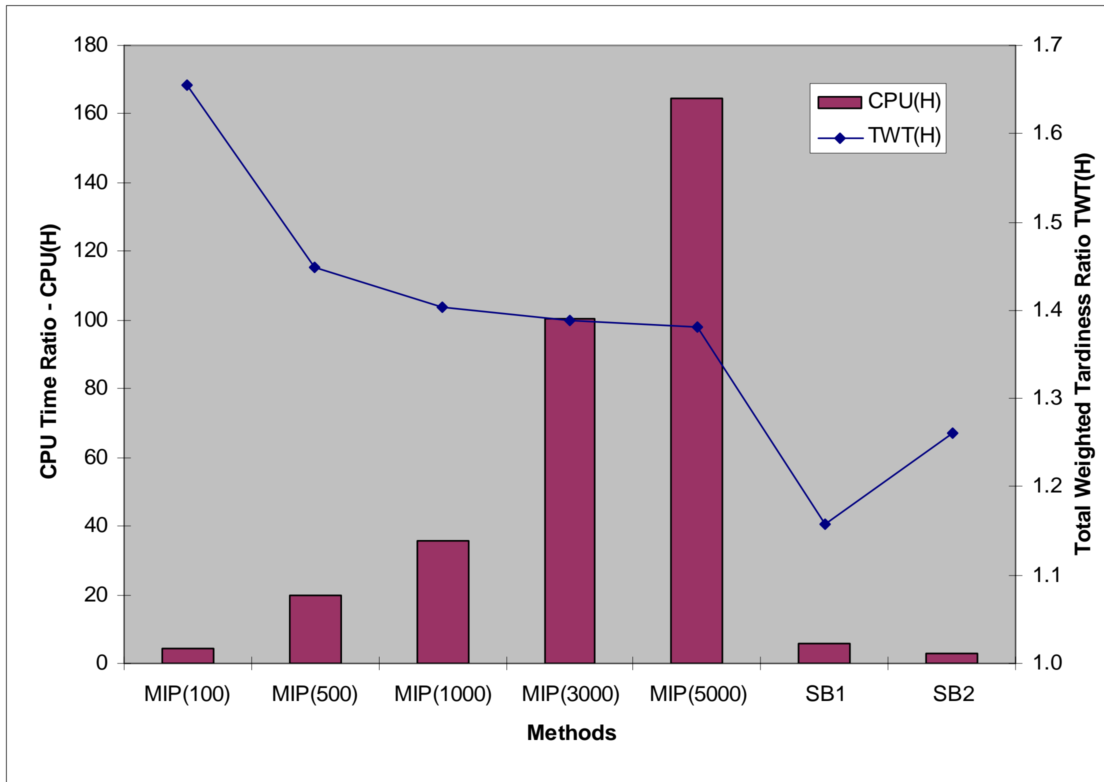


Figure 4.1 CPU(H) and TWT(H) for optimization methods

4.4 Computational results for dispatching rules:

There are six dispatching rules which are used to evaluate the performance of heuristic approaches in this experiment as shown in Table 4.4. These dispatching rules are EDD, WEDD, ATC, WSPT, WTC and MWTC. EDD and WEDD are used to minimize the maximum lateness. WSPT are designed to minimize the total weighted completion time. ATC, WTC and MWTC are introduced to minimize the total weighted tardiness.

Table 4.4 Experimental designs for randomly generated test problems – Dispatching rules

Factors	Levels	Number of levels
Number of jobs	15, 20, 25, 30, 35	5
Number of machines	10	1
Number of items	10	1
Methods	EDD, WEDD, WAC, ACT, MWAC, WSPT	6
Total test problems		180

* EDD: earliest due date first.
*WEDD: Weighted earliest due date first
* WTC: Weighted tardiness cost
* ATC: Apparent tardiness cost
* MWTC: Modified weighted tardiness cost
* WSPT: Weighted shortest processing time first.

Table 4.5 Computational results for dispatching rules

Method	MWTC	ATC	EDD	WEDD	WTC	WSPT
CPU(H) –Mean	1.094	1.085	1.122	1.086	1.065	1.079
TWT(H) -Mean	1.079	1.076	1.204	1.136	1.136	1.288
TWT(H)-SD	0.091	0.080	0.201	0.218	0.086	0.415
Number of best solutions	8	9	1	10	0	1
Percentage of best solutions	32%	36%	4%	40%	0%	4%

The ratio of CPU(H) for all six dispatching rules are closed as shown in Table 4.5. However, the TWT(H) ratio of MWTC and ATC are better than other dispatching rules from 5% to 21% because they consider more information on status of jobs and the shop than others. The TWT(H) ratio of ATC is 0.3% less than the TWT(H) ratio of MWTC. However, the performance of ATC rule relies on the scaling parameter K, which needs to be determined empirically. The value of scaling parameter K used in this experiment is 5. MWTC does not need any parameters and provide similar performance as ATC. WEDD provides better TWT(H) value than EDD by 7%. It indicates the performance can be improved if dispatching rules include job priority in their index. WSPT is commonly used on conventional job shop scheduling problems with string-type jobs which have linear routings. But it does not perform well on assembly job shop according to the performance measure TWT(H) as shown in Table 4.5. In Figure 4.2, it shows that MWTC, ATC and WTC not only have low value on TWT(H) ratio in average, but also provide low variability on TWT(H) ratio as well. It indicates these three dispatching rules produce the solutions which are more reliable than other dispatching rules.

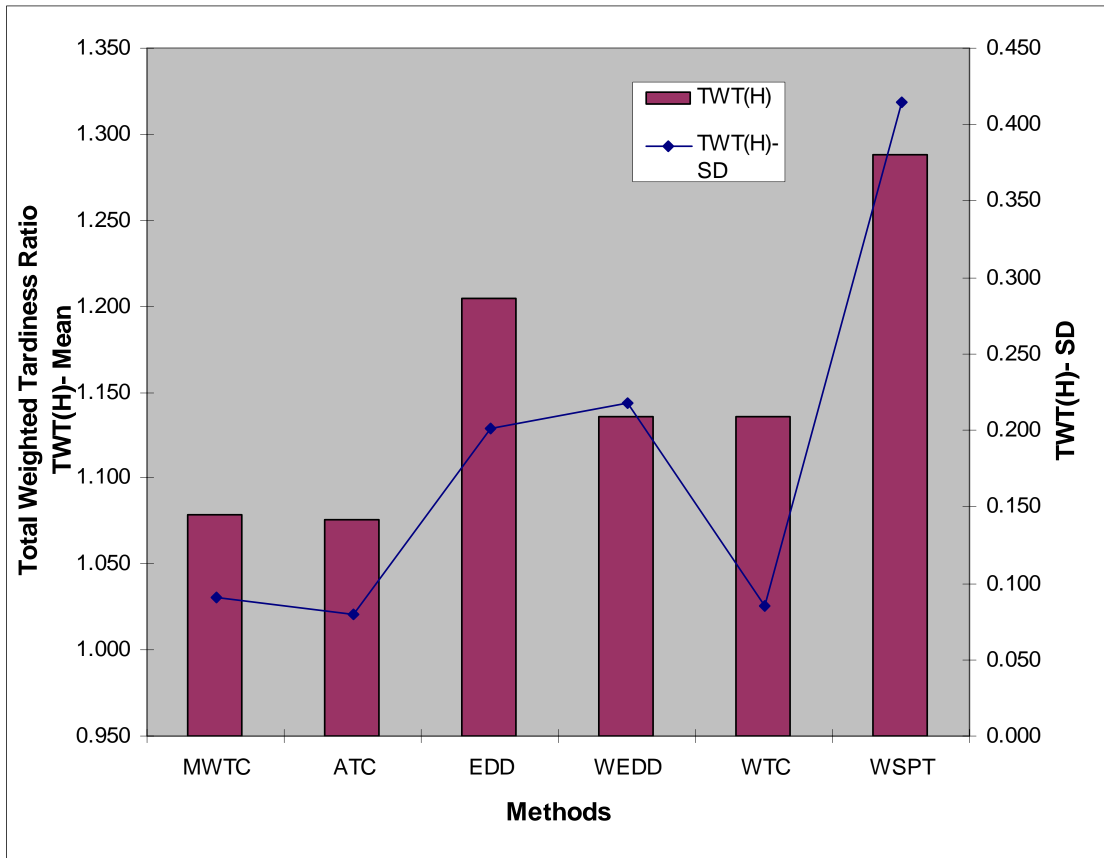


Figure 4.2 TWT(H) – Mean and SD for dispatching rules

4.5 Computational results for heuristic approach 1 – Modified shifting bottleneck procedure (MSBP) :

We present the optimization approach to solve assembly job shop scheduling problems. However, the MIP methods are not scalable to run real-world problems due to excessive computational time. To be able to solve large-sized problems, several heuristic approaches are developed. These heuristics provide solutions close to optimal solutions within reasonable computational times. In this experiment, we are interested in the performance of heuristic approaches with different order sizes from 15 to 35. Hence, the performance of these heuristics is compared to the best solutions obtained form the

dispatching rules. By doing so, we demonstrate the trade-off between optimization approaches, heuristic approaches and dispatching rules in term of computational time and solution quality.

Table 4.6 Experimental design for randomly generated test problems – MSBP

Factors	Levels	Number of levels
Number of jobs	15, 20, 25, 30, 35	5
Number of machines	10	1
Number of items	10	1
Methods	SB1, SB2, SB3-Dispatch, SB4-Dispatch	14
Total test problems		350

*SB1: Shifting bottleneck procedure
*SB2: Shifting bottleneck algorithm (Using longest path to present precedence constraints)

SB3-Dispatch uses the same shifting bottleneck procedure to solve assembly job shop scheduling problems. It still applies optimization methods to solve capacity subproblems but it uses dispatching rules to solve material subproblems to reduce the computational time. There are six different SB3-Dispatch methods tested in this experiment, which included SB3-ATC, SB3-MWTC, SB3-EDD, SB3-WEDD, SB3-WTC and SB3-WSPT as described in Table 4.7. The results of all test problems are shown in Table 4.8 and it can easily be seen that SB3-ATC, SB3-MWTC and SB3-WTC has better TWT(H) ratio than others. Moreover, the TWT(H) ratio of SB3-WEDD is better than SB3-EDD rule by 28%, it indicates that WEDD rule could produce better

solutions than EDD rule in material subproblems because of the consideration of job priority. Additionally, SB3-ATC and SB3-MWTC are chosen to present the SB3-Dispatch compared to other heuristic approaches in later experiments.

Table 4.7 The summary of SB3 methods and dispatching rule used in this experiment

	SB3-ATC	**SB3-MWTC**	**SB3-EDD**	**SB3-WEDD**	**SB3-WTC**	**SB3-WSPT**
Material allocation subproblem	ATC Dispatching rule	MWTC Dispatching rule	EDD Dispatching rule	WEDD Dispatching rule	WTC Dispatching rule	WSPT Dispatching rule
Capacity allocation subproblem	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**

Table 4.8 Computational results for SB3 methods

Method	SB3-ATC	SB3-MWTC	SB3-EDD	SB3-WEDD	SB3-WTC	SB3-WSPT	ATC
CPU(H) -Mean	1333	1402	1593	1371	1396	2581	1
TWT(H) – Mean	1.033	1.038	1.527	1.107	1.039	1.308	1.202
TWT(H) – SD	0.046	0.052	0.595	0.175	0.051	0.314	0.110
Number of best solutions	12	12	0	5	11	0	0
Percentage of best solutions	48%	48%	0%	20%	44%	0%	0%

SB4-Dispatch follows the same shifting bottleneck procedure as SB1 to solve assembly job shop scheduling problems. It applies optimization methods to solve material subproblems but uses dispatching rules to solve capacity subproblems to reduce the computational time. There are six different SB4-Dispatch methods tested in this experiment, which include SB4-ATC, SB4-MWTC, SB4-EDD, SB4-WEDD, SB4-WTC and SB4-WSPT as summarized in Table 4.9. In addition, test problems used in this experiment are material-intensive because of assembly operations and insufficient amounts of components. The results of all test problems are shown in Table 4.10 and it can easily be seen that all SB4-Dispatch rules produce worse TWT(H) ratio than ATC dispatching rule. This indicates that using optimization methods to solve material constraints incompletely may produce worse solutions than using dispatching rules, such as ATC rule. Furthermore, SB4-ATC and SB4-WEDD are chosen to present the SB4-Dispatch compared to other heuristics approaches in later experiments.

Table 4.9 The summary of SB4 methods and dispatching rule used in this experiment

	SB4-ATC	SB4-MWTC	SB4-EDD	SB4-WEDD	SB4-WTC	SB4-WSPT
Material allocation subproblem	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**
Capacity allocation subproblem	ATC Dispatching rule	MWTC Dispatching rule	EDD Dispatching rule	WEDD Dispatching rule	WTC Dispatching rule	WSPT Dispatching rule

Table 4.10 Computational results for SB4 methods

Method	SB4-ATC	SB4-MWTC	SB4-EDD	SB4-WEDD	SB4-WTC	SB4-WSPT	ATC
CPU(H) - Mean	10621	10840	10753	10653	10672	10373	1
TWT(H) – Mean	1.241	1.232	1.418	1.137	1.186	1.294	1.095
TWT(H) – SD	0.281	0.274	0.448	0.121	0.138	0.339	0.186
Number of best solutions	4	3	1	5	1	3	11
Percentage of best solutions	16%	12%	4%	20%	4%	12%	44%

The comparison of MSBP method is summarized in Table 4.11. There are six MSBP methods tested in this experiment, which included SB1, SB2, SB3-ATC, SB3-MWTC, SB4-WEDD and SB4-WTC. It can easily be seen that SB1 has the best THT(H) ratio among all MSBP methods shown in Table 4.11. The TWT(H) ratio of SB3-ATC and SB3-MWTC is worse than TWT(H) of SB1 by 5%, but these two methods could reduce computational times by 91% compared to SB1. Moreover, the two SB3 methods also provide better solutions than ATC rule by 14% in term of TWT(H) ratio. However, two SB4 methods produce worse solutions than any methods used in this experiment. The difference between SB3 and SB4 is different degrees of solutions obtained from material subproblems and capacity subproblems. SB3 uses dispatching rules to solve material subproblems and apply optimization methods to obtain solutions from capacity subproblems. SB4 solves capacity subproblems by using dispatching rules and solves material subproblems by employing optimization methods. It indicates that dispatching rules, ATC and MWTC, used in this experiment can provide solutions closed to SB1 in material subproblems. Second, the dispatching rules could produce better solutions in material subproblems than capacity subproblems compared to optimization methods with limited computational time settings. In addition, SB2 does not perform well in both TWT(H) and CPU(H) ratios because of the technical problems of programming. The longest path algorithm used in SB2 is written using MOSEL language, which is designed to solve mathematical problems. It is very inefficient to execute “IF statement” in terms of computational times compared to C language, particularly in large-sized instances. It caused SB2 to become the worst among all heuristics approaches. In Figure 4.3, it can

easily be seen that SB3 methods are best choice to solve the problems efficiently in terms of computational times and solution quality among all MSBP methods.

Table 4.11 Computational results for MSBP methods

Method	SB1	SB2	SB3-ATC	SB3-MWTC	SB4-WEDD	SB4-WTC	ATC
CPU(H) - Mean	16429	61753	1333	1402	10653	10672	1
TWT(H) – Mean	1.034	1.433	1.078	1.084	1.309	1.367	1.260
TWT(H) – SD	0.047	0.339	0.106	0.117	0.152	0.171	0.205
Number of best solutions	13	0	7	7	0	0	0
Percentage of best solutions	52%	0%	28%	28%	0%	0%	0%

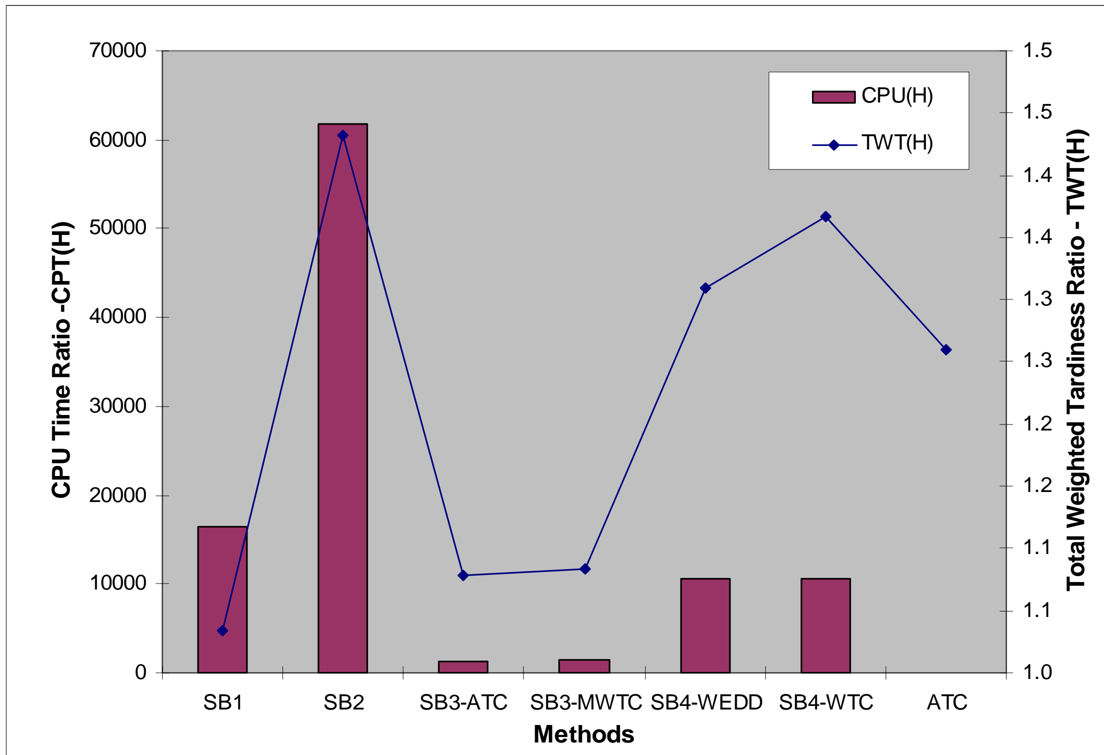


Figure 4.3 CPU(H) and TWT(H) for MSBP methods

4.6 Computational results for heuristic approach 2 – Efficient shifting bottleneck procedure (ESBP) :

Although MSBP produces better solutions than the MIP method, it is still a time-consumed approach due to numerous subproblems needing to be solved during the re-optimization procedure. According to Chen et al. (2006), the complexity increases as the number of jobs increases and the number of machine types increases, as there are more subproblems to solve. By reducing the number of subproblems or applying efficient procedures to solve subproblems, one can reduce the computational time of the shifting bottleneck procedure significantly. Additionally, Demirkol et al. (1997) concluded that

bottleneck machine selection criteria had no significant effect on solution quality and computational time but subproblems and re-optimization procedures did.

It is reasonable to obtain an initial solution by using a dispatching rule, then improve it using a re-optimization procedure in order to reduce the number of subproblems which required to solve assembly job shop scheduling problems by using MSBP methods. Chen et al. (2006) propose an efficient shifting bottleneck heuristic for parallel machine work-centers, where bottleneck machine selection procedure is avoided and re-optimization procedure is constrained to few iterations. Hence, the total number of subproblems is expected to decrease. In this section, the performance of ESBP is evaluated and the effect of the re-optimization procedure on TWT(H) and CPU(H) are investigated in this experiment.

Table 4.12 Experimental design for randomly generated test problems - ESBP

Factors	Levels	Number of levels
Number of jobs	15, 20, 25, 30, 35	5
Number of machines	10	1
Number of items	10	1
Methods	ESBP1, ESBP2-Dispatch, ESBP3-Dispatch	13
Total test problems		325

In ESBP1, the initial solution is obtained by dispatching rules. Then, improve it by using re-optimization procedure with optimization methods to solve both material subproblems and capacity subproblems. ESBP1 avoids the bottleneck machine selection procedure and limits the re-optimization procedure to few iterations. By doing so, ESBP1 is able to reduce the excess of computational times which is a major complaint of MSBP as the size of problems increase. The total number of subproblems requiring to be solved by ESBP1 is expected to decrease as compared to MSBP methods. There are six different ESBP1 methods tested in this experiment, which include ESBP1-ATC, ESBP1-MWTC, ESBP1-EDD, ESBP1-WEDD, ESBP1-WTC and ESBP1-WSPT as seen in Table 4.13. The results of all test problems are shown in Table 4.14 and it can see easily that ESBP1-ATC, ESBP1-MWTC and ESBP1-WTC provide better TWT(H) ratio than the others. Moreover, ESBP1 methods that produce better initial solutions do perform better on TWT(H) ratio. The only exception is ESBP1-WTC. This indicates that the better initial solution could help ESBP1 methods perform better during re-optimization procedures. Furthermore, ESBP1-ATC and ESBP1-MWTC are chosen to present the ESBP1 compared to other heuristic approaches in later experiments.

Table 4.13 The summary of ESBP1 methods used in this experiment

	ESBP1-ATC	**ESBP1-MWTC**	**ESBP1-EDD**	**ESBP1-WEDD**	**ESBP1-WTC**	**ESBP1-WSPT**
Initial Solution	ATC Dispatching rule	MWTC Dispatching rule	EDD Dispatching rule	WEDD Dispatching rule	WTC Dispatching rule	WSPT Dispatching rule
Material allocation subproblem	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**
Capacity allocation subproblem	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**

Table 4.14 Computational results for ESBP1 methods

Method	ESBP1-ATC	ESBP1-MWTC	ESBP1-EDD	ESBP1-WEDD	ESBP1-WTC	ESBP1-WSPT	ATC
CPU(H) - Mean	1178	1260	1206	1102	1219	1088	1
TWT(H) – Mean	1.057	1.070	1.162	1.128	1.089	1.274	1.257
TWT(H) – SD	0.060	0.084	0.151	0.144	0.073	0.334	0.141
Number of best solutions	9	7	3	8	0	1	0
Percentage of best solutions	36%	28%	12%	32%	0%	4%	0%

ESBP2 follows the same procedure as ESBP1 to solve assembly job shop scheduling problems. But it solves the material subproblems by using dispatching rules and obtains the solutions from capacity subproblems by optimization methods during the re-optimization procedure. There are six different ESBP2 methods tested in this experiment, which include ESBP2-ATC, ESBP2-MWTC, ESBP2-EDD, ESBP2-WEDD, ESBP2-WTC and ESBP2-WSPT as summarized in Table 4.15. The results of all test problems are shown in Table 4.16 and it can easily be seen that ESBP2-ATC, ESBP2-MWTC and ESBP2-WTC has best TWT(H) ratio than the others. It indicates that these three ESBP2 methods can generate similar solutions which are close to the solutions from optimization methods with limited computational times restrictions in material subproblems. But the TWT(H) ratio of ATC rule also indicates that ESBP2 methods produce the solutions which are worst than the ones obtained from the ESBP1. However, the CPU(H) ratio of ESBP2 shows a significant reduction on computational time by 80% compared to ESBP1. Moreover, the ESBP2 methods that produce better initial solutions do perform better on the TWT(H) ratio. It is the same finding as seen in the results from the ESBP1. Furthermore, ESBP2-ATC and ESBP2-MWTC are chosen to present the ESBP1 compare to other heuristic approaches in later experiments.

Table 4.15 The summary of ESBP2 methods used in this experiment

	ESBP2-ATC	**ESBP2-MWTC**	**ESBP2-EDD**	**ESBP2-WEDD**	**ESBP2-WTC**	**ESBP2-WSPT**
Initial Solution	ATC Dispatching rule	MWTC Dispatching rule	EDD Dispatching rule	WEDD Dispatching rule	WTC Dispatching rule	WSPT Dispatching rule
Material allocation subproblem	ATC Dispatching rule	MWTC Dispatching rule	EDD Dispatching rule	WEDD Dispatching rule	WTC Dispatching rule	WSPT Dispatching rule
Capacity allocation subproblem	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**	Optimization method **(CA1)**

Table 4.16 Computational results for ESBP2 methods

Method	ESBP2-ATC	ESBP2-MWTC	ESBP2-EDD	ESBP2-WEDD	ESBP2-WTC	ESBP2-WSPT	ATC
CPU(H) - Mean	198	209	193	218	234	230	1
TWT(H) – Mean	1.051	1.065	1.215	1.131	1.075	1.323	1.197
TWT(H) – SD	0.058	0.078	0.206	0.204	0.077	0.413	0.102
Number of best solutions	9	8	1	7	6	0	0
Percentage of best solutions	36%	32%	4%	28%	24%	0%	0%

ESBP3 follows the same procedure as ESBP1 to solve assembly job shop scheduling problems. But it solves the material subproblems by using the optimization method and obtains the solutions from capacity subproblems by dispatching rules during the re-optimization procedure. As seen in Table 4.17, there are six different ESBP3 methods tested in this experiment, which include ESBP3-ATC, ESBP3-MWTC, ESBP3-EDD, ESBP3-WEDD, ESBP3-WTC and ESBP3-WSPT. The results of all test problems are seen in Table 4.18 and it can easily be seen that only ESBP3-MWTC, ESBP3-WEDD and ESBP3-WTC have better TWT(H) ratio than the ATC rule by 3% on average. It indicates that all ESBP3 methods does not perform well in capacity subproblems by using dispatching rules compared to the optimization method with limited computational times. In addition, ESBP3 methods only reduce computational times by 25% compared to ESBP1. Furthermore, ESBP3-WEDD and ESBP3-MWTC are chosen to present the ESBP3 compared to other heuristic approaches in later experiments.

Table 4.17 The description of ESBP3 methods used in this experiment

	ESBP3-ATC	**ESBP3-MWTC**	**ESBP3-EDD**	**ESBP3-WEDD**	**ESBP3-WTC**	**ESBP3-WSPT**
Initial Solution	ATC Dispatching rule	MWTC Dispatching rule	EDD Dispatching rule	WEDD Dispatching rule	WTC Dispatching rule	WSPT Dispatching rule
Material allocation subproblem	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**	Optimization method **(MA1)**
Capacity allocation subproblem	ATC Dispatching rule	MWTC Dispatching rule	EDD Dispatching rule	WEDD Dispatching rule	WTC Dispatching rule	WSPT Dispatching rule

Table 4.18 Computational results for ESBP3 methods

Method	ESBP3-ATC	ESBP3-MWTC	ESBP3-EDD	ESBP3-WEDD	ESBP3-WTC	ESBP3-WSPT	ATC
CPU(H) - Mean	914	919	1023	809	1000	846	1
TWT(H) - Mean	1.085	1.094	1.191	1.118	1.136	1.289	1.145
TWT(H) - SD	0.086	0.109	0.169	0.152	0.075	0.398	0.111
Number of best solutions	7	9	2	10	0	1	0
Percentage of best solutions	28%	36%	8%	40%	0%	4%	0%

The comparison of all ESBP methods is given in Table 4.11. There are six ESBP methods tested in this experiment, which are ESBP1-ATC, ESBP1-MWTC, ESBP2-ATC, ESBP2-MWTC, ESBP3-MATC and ESBP3-WEDD. It can be seen that ESBP1 has better THT(H) ratio among all ESBP methods as shown in Table 4.19. The TWT(H) ratio of ESBP2-ATC and ESBP2-MWTC is worse than TWT(H) of ESBP1 around 5%, but they could reduce computational times by 83%. In addition, ESBP2 methods are superior to the ATC rule in terms of the solution quality. Two ESBP3 methods only provide better solutions than ATC rule by 4% on TWT(H) ratio. The difference between ESBP2 and ESBP3 is a different degree of solutions obtained from material subproblems and capacity subproblems. ESBP2 uses dispatching rules to solve material subproblems and applying optimization methods to obtain solutions from capacity subproblems. ESBP3 solves capacity subproblems by using dispatching rules and solves material subproblems by employing optimization methods. This indicates that dispatching rules, ATC and MWTC, used in this experiment could provide solutions closed to ESBP1 in material subproblems. Second, the dispatching rules could produce better solutions in material subproblems than capacity subproblems compared to optimization methods with limited computational time settings.

Table 4.19 Computational results for ESBP methods

Method	ESBP1-ATC	ESBP2-MWTC	ESBP3-MATC	ESBP1-MWTC	ESBP2-ATC	ESBP3-WEDD	ATC
CPU(H) - Mean	1273	209	919	1260	231	809	1
TWT(H) – Mean	1.049	1.107	1.190	1.061	1.117	1.215	1.245
TWT(H) – SD	0.051	0.135	0.155	0.129	0.134	0.183	0.166
Number of best solutions	6	4	1	10	3	4	0
Percentage of best solutions	24%	16%	4%	40%	12%	16%	0%

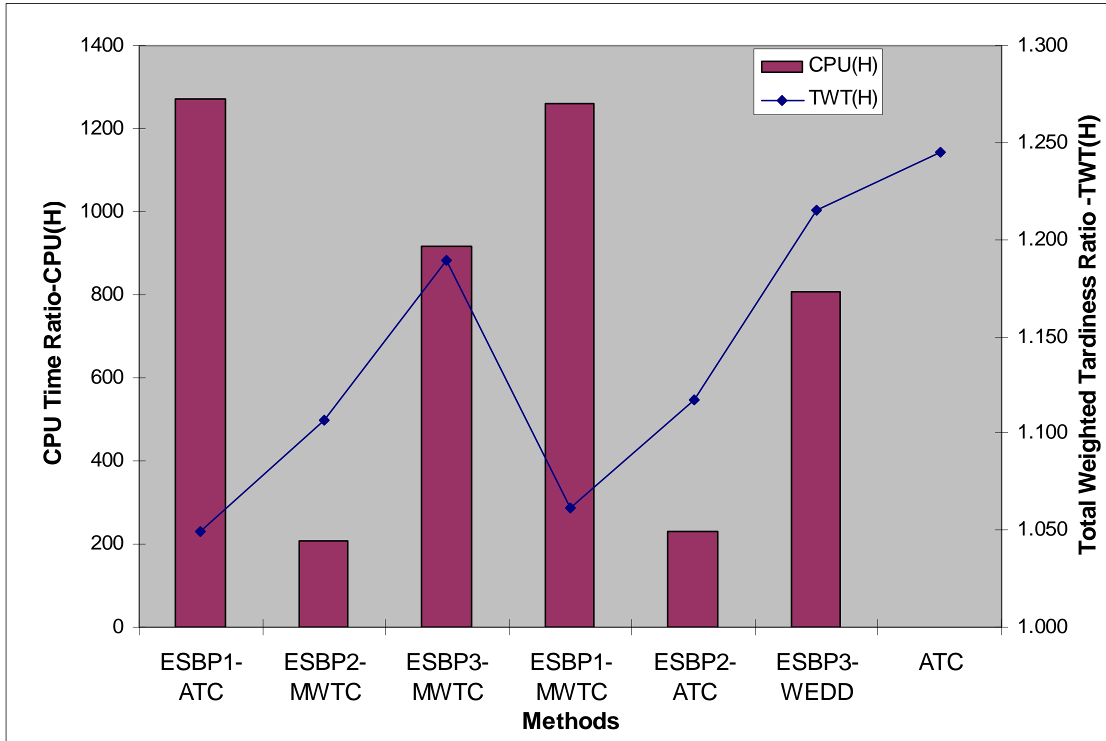


Figure 4.4 CPU(H) and TWT(H) for ESBP methods

4.7 Computational results: ESBP vs. MSBP vs. Dispatching rules

In this section, the performance of seven methods from MSBP, ESBP and dispatching rules is investigated in terms of TWT(H) ratio and CPU(H) ratio. These methods are SB1, SB3-ATC, SB4-EDDP, ESBP1-ATC, ESBP2-MWTC, ESBP3-MWTC and ATC rule. As be seen in Table 4.20, SB1 has the best TWT(H) ratio among all methods and is followed by ESBP1-ATC, SB3-ATC and ESBP2-ATC. But SB1 also have the worst performance on CPU(H) ratio among all methods. SB1 produces a better TWT(H) ratio than ESBP2-MWTC by 8.2%, but it has a CPU(H) ratio 78 times more than ESBP2-MWTC. In addition, SB1 has a better TWT(H) ratio than ESBP1-ATC by 2.5%, but it has a CPU(H) ratio 13 times more than ESBP1-ATC. Hence, ESBP1-ATC

and ESBP2-MWTC both perform better than the ATC rule on TWT(H) ratio by 16% and 11%. These results indicate that ESBP could dramatically reduce the computational times of MSBP by decreasing the total number of subproblems which are required during re-optimization procedures. Second, avoiding the knowledge on the rank of bottleneck resources selection only has a small effect on the solution quality around 3% as shown on a comparison of TWT(H) ratio of MSBP over TWT(H) of ESBP. Third, the reduction of computational times on SB3-ATC is due to the reduction of computational times to solve material subproblems by using dispatching rules. The reduction of computational times on ESBP2-MWTC is due to the decrease of total number of subproblems and implementation of dispatching rules on material subproblems. Fourth, using dispatching rules to solve capacity subproblems does not provide the acceptable solutions which are close to the ones obtained from optimization methods with limited computational times. However, using dispatching rules to solve material subproblems does produce the acceptable solutions which are close to the ones obtained from optimization methods with limited computational times. It made the performance of SB3-ATC and ESBP2-MWTC better than SB4-WEDD and ESBP3-MWTC on both TWT(H) and CPU(H).

Table 4.20 Computational results for ESBP vs. MSBP

Method	SB1	SB3-ATC	SB4-WEDD	ESBP1-ATC	ESBP2-MWTC	ESBP3-MWTC	ATC
CPU(H) - Mean	16429	1345	10653	1273	209	919	1
TWT(H) – Mean	1.044	1.090	1.323	1.070	1.130	1.212	1.274
TWT(H) – SD	0.056	0.122	0.164	0.094	0.162	0.167	0.221
Number of best solutions	10	7	1	5	4	0	0
Percentage of best solutions	40%	28%	4%	20%	16%	0%	0%
Number of subproblems	230.8	230.8	232	28.8	26.4	21.6	-

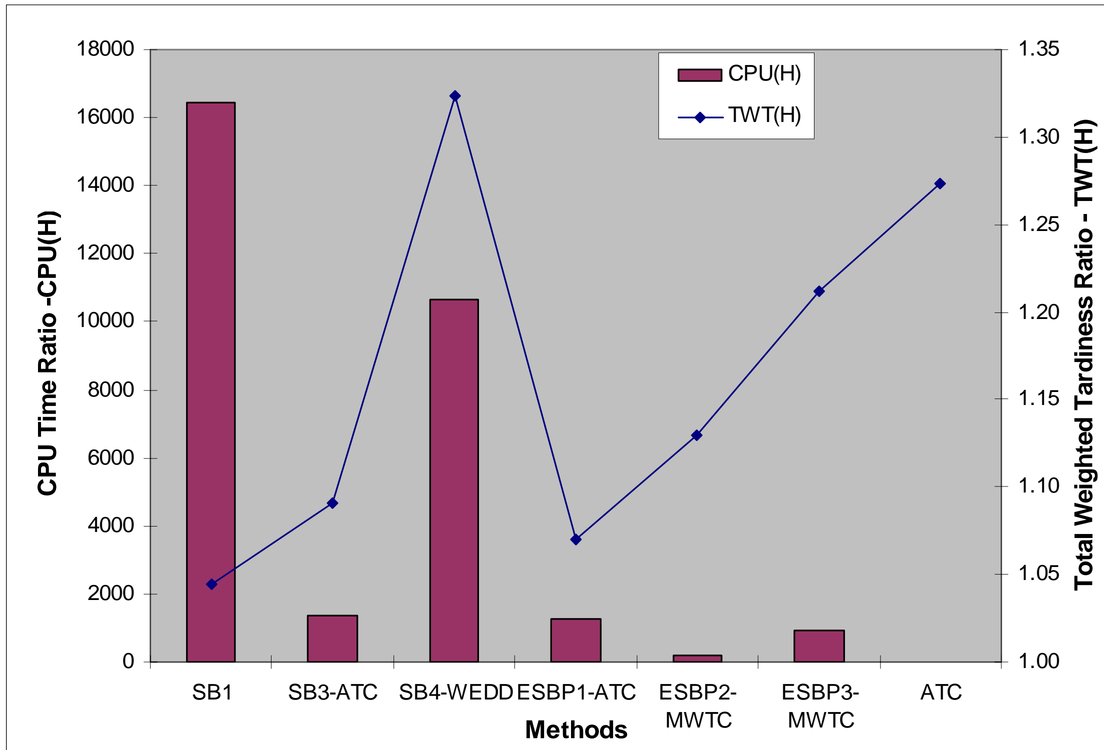


Figure 4.5 CPU(H) and TWT(H) for MSBP vs. ESBP

4.8 Computational results for RHP :

In this section, RHP, ESBP2-ATC and ATC rule are selected as candidates to evaluate the performance of RHP in this experiment as shown in Table 4.22. The only difference between RHP and ESBP2-ATC is to use different methods to solve capacity subproblems. In RHP, capacity subproblems are solved by using optimization methods combined with rolling horizon procedures. For each capacity subproblem, operations are broken down into different time windows and solved them iteratively. However, in

ESBP2-ATC, each capacity subproblem is solved by using optimization methods in single time window. By doing this, RHP could reduce the complexity of capacity subproblems in case of the number of jobs increases.

Table 4.21 Experimental design for randomly generated test problems –RHP

Factors	Levels	Number of levels
Number of jobs	40, 50, 60, 70, 80	5
Number of machines	10	1
Number of items	10	1
Methods	ESBP2-ATC, RHP, ATC	3
Total test problems		75

In Table 4.23, it can be seen that RHP has a TWT(H) ratio which is 0.9% worse than ESBP2-ATC. But it reduces the CPU(H) ratio by 55% compared to ESBP2-ATC. This indicates that using rolling horizon procedures can reduce the complexity of assembly job shop scheduling problems. However, CPU(H) ratio of RHP raises when the number of jobs increases from 40 to 80. It is because the number of jobs in each time window also increases and it requires more computing power to solve the problems associated with each time window. Moreover, the computational time for each subproblem in ESBP2-ATC already reaches the maximum CPU setting and would not increase anymore. Additionally, TWT(H) ratio of RHP is 6% better than the ATC rule. It can be improved by increasing the maximum CPU time setting for each subproblem in Xpress-MP.

Table 4.22 Computational results for RHP – CPU(H)

Number of jobs	RHP	ESBP2-ATC	ATC
40	184.6	511.8	1
50	234.6	597.3	1
60	277.0	583.4	1
70	252.7	520.6	1
80	260.8	458.4	1
Average	241.9	534.3	1

Table 4.23 Computational results for RHP – TWT(H)

Number of jobs	RHP	ESBP2-ATC	ATC
40	1.015	1.002	1.104
50	1.021	1.005	1.076
60	1.015	1.014	1.053
70	1.026	1.016	1.057
80	1.023	1.016	1.074
Average	1.020	1.011	1.082
SD	0.005	0.007	0.017

4.9 Summary of computational results:

Based on findings from computational results, we can draw the decision tree for choosing appropriate methods to solve assembly job shop scheduling problem. According to number of jobs and maximum computation time allowance, the best method to use to maximize total weighted tardiness can be shown as the decision tree below in Figure 4.6.

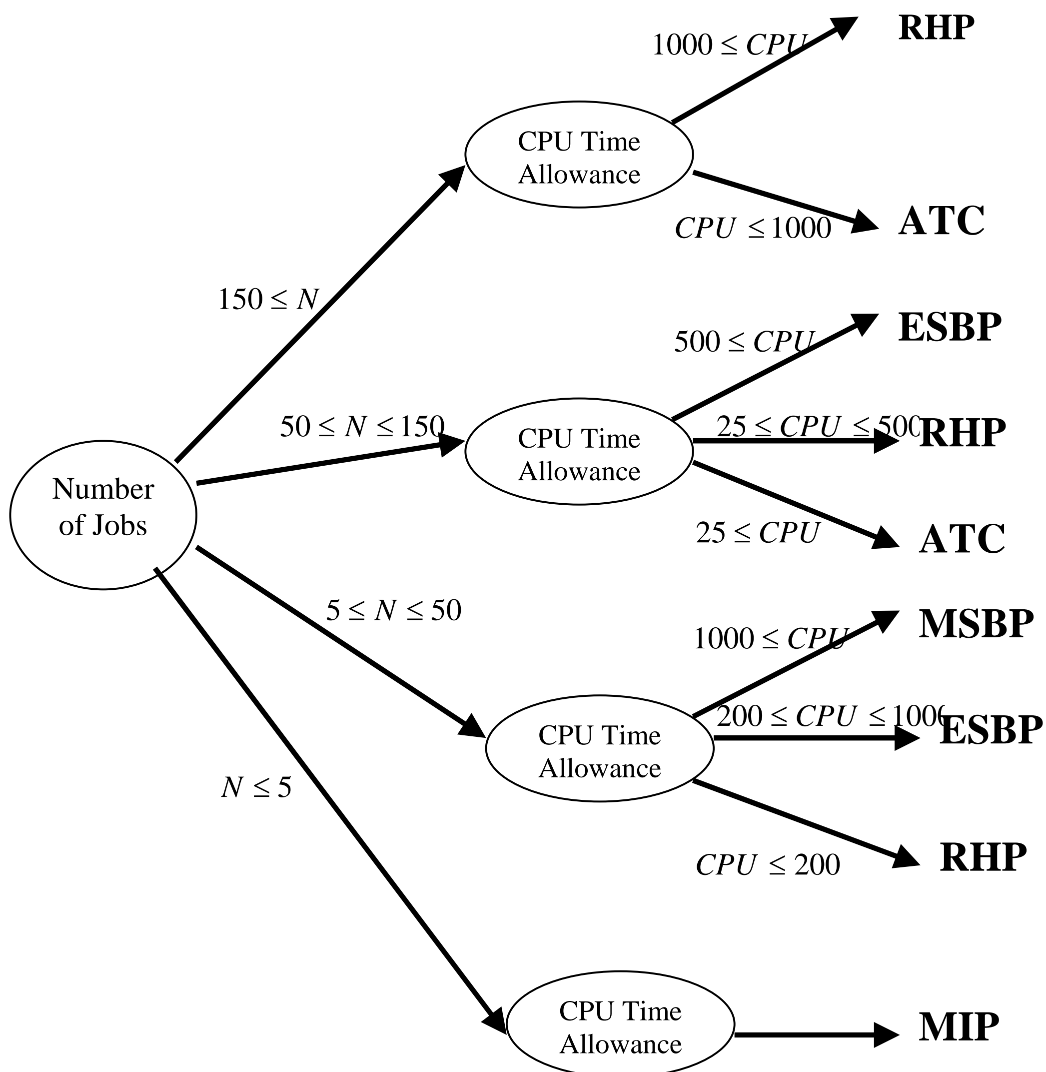


Figure 4.6 Summary of computational results

Chapter 5. Future research directions

5.1 Dynamic assembly job shop scheduling problems with components availability constraints

In this research, several assumptions have been made to simplify the assembly job shop scheduling problems. One of the assumptions is the static assembly job shop scheduling problem, which means all jobs are ready to process in the beginning. However, real life scheduling problems are dynamic and stochastic in nature. The arrival times of jobs are not always the same and new jobs can enter the system after processing start in a random manner. Some operations could still be in progress at the point that scheduling decisions are made. These problems are so-called dynamic assembly job shop scheduling problems. In dynamic assembly job shop scheduling problems, problems can classified into two classes as deterministic and stochastic assembly job shop problems based on the information of problem parameters, such as processing time and release times. For example, release times play a critical rule for the performance of job shop because they control work-in-process (WIP) directly. If release times of jobs are fixed, these problems are deterministic assembly job shop scheduling problems. If release times of jobs are varied and can not be known in advance, these problems become stochastic assembly job shop scheduling problems.

It is interesting to study dynamic assembly job shop scheduling problems with component availability constraints, because these problems are much more complicated than the static assembly job shop scheduling problems and are close to realistic problems. However, there are more constraints to be considered in dynamic assembly job shop problems than static ones. First, some operations may be still in progress on specific

machines at the time of scheduling. These operations have to be the earliest ones on the machine sequences, otherwise, the solution would not be feasible. Second, jobs can not be processed before their release times. Third, previous schedules may not be valid and rescheduling actions are needed because of new arrival jobs and completed jobs. A mathematical formulation of dynamic assembly job shop scheduling problems with deterministic release times and processing time is described as follows.

(P) $$\min \sum_{v} w_v * \max(s_v - d_v, 0) \quad v \in Z, \quad \text{------------------------------(1)}$$

Subject to:

** Assembly coordination constraints*

$$s_v = t_i + p_i \qquad i \in G, \, v \in Z \quad \text{----------------------(2)}$$

$$t_j \geq t_i + p_i \qquad (i, j) \in A, \text{ j is i parent operation ---(3)}$$

$$t_i \geq r_i \vee t_i \geq T \qquad i \in N, \quad \text{--------------------------------(4)}$$

** Capacity constraints*

$$t_j - t_i \geq p_i \vee t_i - t_j \geq p_j, \qquad (i, j) \in E_k, k \in M. \quad \text{------------------(5)}$$

$$t_j - t_i \geq u_i + T \qquad i \in U_k, \; k \in M. \text{ and j is i parent operation---(6)}$$

** Material constraints*

$$\sum_{i} y_{i,n} \leq q_n \qquad i \in B_h, n \in F_h, h \in P \quad \text{---------------(7)}$$

$$\sum_{n} y_{i,n} = q_i \qquad i \in B_h, n \in F_h, h \in P \quad \text{---------------(8)}$$

$$t_i \geq a_n \vee t_i \geq r_i \qquad i \in B_h, n \in F_h, h \in P \quad \text{---------------(9)}$$

Table 5.1 Notation of dynamic assembly job shop scheduling problems

N	Set of operations
Z	Set of jobs
d_v	The due date for job v, $v \in Z$
w_v	The priority for job v, $v \in Z$
s_v	The completion time for job v, $v \in Z$
M	The set of all machines
T	The current time
U_k,	The set of in progress operation for machine k, $k \in M$
u_i	Remaining operation time for operation i
r_i	Release time for operation i
P	The set of all components
A	The set of pairs of operations constrained by precedence relations
E_k	The set of pairs of operations to be performed on machine *k*
p_i	The processing time for operation *i*
t_i	The start time for operation *i*
q_i	The requested quantity for components from operation *i*
B_h	The set of operations to request component h , $h \in P$
F_h	The set of purchase orders to delver component *h*, $h \in P$
R	The set of all purchase orders
a_n	The arrival time for purchase order *n*, $n \in R$
q_n	The delivery quantity for purchase order *n*, $n \in R$
Y_h	The material allocation of components *h*, $h \in P$
$y_{i,n}$	The allocation of component from purchase order *n* to operation *i*.

Equation 1 is the objective function of the problem. It tends to minimize the total weighted tardiness for all jobs. Equation 2 and 3 represent the precedence relation

between operations. . Equation 4 ensures that no operations are starting before its release time. Equation 5 specifies that the no two operations belonging to the same machine can be processed at the same time. Equation 6 ensures that unfinished operation is the first one to be processed on the specific machine. Equation 7, 8 and 9 are material constraints. They guarantee that all operations can not be started before the arrival of necessary components and all material requirements are fulfilled. Equation 9 specifies that the operation will not start before the release time or arrival time of specific purchase order.

5.2 The effect of different approaches on the overall performance of a production system

It is interesting to investigate the effect of the different approaches with varying degrees of solution accuracy on the overall performance of a production system, including due-date, order release, shop floor scheduling and machine sequence.

Several researchers have studied due-date setting, material planning and order release and review problems over the past few decades. Moses et al. (2003) proposed the real-time order promising methods with two different order release policies. Their results indicate that order release policy can significantly improve the overall due-date performance. They also conclude that good due-date assignment methods lead to superior due-date and shop floor performance. However, poor due-date assignment methods will restrict the overall performance despite of an efficient order release policy. It is motivating to investigate different scheduling methods on the overall performance of a production system, which included total weight tardiness, total tardiness, tardy ratio, service level, median absolute lateness, estimated flowtime, actual flowtime, shop flow time, total work

in process, released work in process and utilization. By doing so, we are able to define how the different scheduling methods can help manufacturers achieve superior performance and determine what the actual contributions of these methods are to the performance of a production system. By doing this, we are able to answer that whether better scheduling approaches can improve overall performance with or without good quality due-date setting methods, order release technique and sequencing rules.

5.3 Other future research and directions

A good direction of further research is to consider improving the solution quality of subproblems in the efficient SB heuristics. It will be interesting to apply other efficient algorithms to solve material plan and machine schedule subproblems. EDD-LI, proposed by Ovacik and Uzsoy (1997) considered adjacent pair exchange to find local optimum, is a good algorithm to improve solution quality of the efficient SB procedures. RHP demonstrates how to breakdown the subproblems and solve them efficiently can reduce the complexity of problems. The algorithm to determine which operations put into these time windows is essential for the solution quality and computational effects. It may become an interesting subject to study in the future.

Components substitution is also another interesting topic to improve due date and shop floor performance. Chen et al. (2001, 2002) incorporate two dimensions of "customer flexibility" – quantity and customer raw material configuration. The flexibility allows customers to select preferred suppliers for raw materials. For a specific raw material type (component), if customers choose more than one supplier source, manufacturers will have a better chance to accept orders by switching from one raw

material to another based on each customer's preference list.

Computing power has been improved dramatically in past few years and processors with multiple cores are available to PC users. With multiple execution cores, the processor can manage processor-intensive and multi-threads tasks easily. Another interesting topic for further research is to distribute the subproblems to different threads and use multiple-cores processor to execute these threads simultaneously. This work can be done by using the specific software and import the settings into an optimization software, such as Xpress-MP. By doing this, we expect that the computational time can be reduced even the total number of subproblems remain the same as it is in this research.

Chapter 6. Conclusions

The focus of most studies in production scheduling is on convenient job shop scheduling problems. Only few of these research considers job shop scheduling problems involving assembly operations. This dissertation reviews the literature related to assembly job shop scheduling problems. It also considers component availability constraints and addresses scalability issues. The functionality of assembly job shop scheduling in this dissertation is not only to allocate jobs to machines, but also to consider material allocation and the availability of components. Moreover, the disjunctive graph is extended to represent the assembly job shop scheduling problems and the mathematical model is also developed. Several heuristic approaches are also introduced to reduce the computational effort and are able to solve large-scale problems in practice. This research also investigates the performance of optimization approaches, heuristics methods and dispatching rules. The computational results demonstrate the trade-off between optimization and heuristic approaches in terms of solution quality and computational time.

The shifting bottleneck procedure is one of the most popular heuristics used to solve job-shop problems. In this dissertation, modified shifting bottleneck procedures (MSBP) adopt the concept of shifting bottleneck procedures to reduce computation time and maintain solution quality in assembly job shop scheduling problems with component availability constraints. An efficient shifting bottleneck procedure (ESBP) is also developed to reduce the complexity of problems and the trade-off between computation time and solution quality is also studied. The results of experiments indicate that the number of subproblems involved in shifting bottleneck procedures has a significant effect

on the computational time. Moreover, decreasing the total number of subproblems does not only reduce most of the computational time significantly but also increases robustness in large-scale instances. The computational results also suggest using rolling horizon procedure (RHP) to restrain the size of subproblems can reduce most of computational times and keep solution quality at an accept level.

Reference

Adam, N. R., Bertrand, J. M., and Surkis, J., (1987), "Priority Assignnet Procedures in Multi-Level Assembly Job Shops", *IIE Transaction*, 19, 317-328.

Adams, J., Balas, E., and Zawack, D., (1988), "The shifting bottleneck procedure for job shop scheduling", *Management Science*, 34, 391-401.

Baker, K. R., and Kanet, J. J., (1983), "Job Shop Scheduling with modified due dates", *Journal of Operations Management,* 4, 11-12.

Balakrishnan, A. and Geunes, J., (2000), "Requirements Planning with Substitutions: Exploiting Bill-of-Materials Flexibility in Production Planning", *Manufacturing & Service Operations Management*, 2, 2, 166-185.

Bragg, D., Duplaga, E. and Watts, C., (1999), "The effects of partial order release and component reservation on inventory and customer service performance in an MRP environment", *International Journal of Production Research*, 37, 3, 523-538.

Chen, C.-Y., Zhao, Z, and Ball, M., (2002), "A Model for Batch Advanced Available-To-Promise", *Production and Operations Management*, 11, 4.

Chen, C.-Y., Zhao, Z, and Ball, M., (2001), "Quantity and Due Date Quoting Available to Promise"*, Information Systems Frontiers*, 3, 4, 477-488.

Chen, K., Lee, M., Pulat, S. and Moses, S., (2006), "The shifting bottleneck procedure for job shops with parallel machines", *International Journal of Industrial and Systems Engineering,* 1, 244-262.

Conway, R. W., and Maxwell, W. L., (1962), "Network dispatching by shortest operation discipline", *Operation Research*, 10, 51-73.

.

Demirkol, E., Mehta, S. and Uzsoy, R., (1997), "A computational study of shifting bottleneck procedures for shop scheduling problems", *Journal of Heuristic,* 3, 111-137.

Elvers, D. A., and Treleven, M. D., (1985), "An investigation of labor assignment rules in a dual-constrained job shop", *Journal of Operations Management*, 6, 51-68.

Fry, T. D., Oliff, M. D., Minor, E. D., and Leong, G. K., (1989), "The effects of production structure and sequencing rules on assembly-job shop performance", *International Journal of Production Research*, 27, 671-686.

Goldratt, E. M. and Cox, J., (1986), *The Goal: A Process of Ongoing Improvement*, North River Press, New York, NY.

Guide, J. V., and Srivastava, R., (2000), "A review of techniques for buffering against uncertainty with MRP systems", *Production Planning and Control*, 11, 3, 223-233.

Holtsclaw, H. H. and Uzsoy, R., (1996), “Machine criticality measures and subproblem solution procedures in shifting bottleneck methods: A computational study”, *Journal of the Operational Research Society*, 47, 666-677.

Hopp, W. J. and Spearman, M. L., (2000), *Factory Physics*, McGraw-Hill, New York, NY.

Johnson, S. M., (1954), “Optimal two- and three-stage production schedules with set-up times included”, *Naval Research Logistics Quarterly*, 1, 61-68.

Karmarkar, U. and Nambimadom, R., (1996), “Material allocation in MRP with Tardiness Penalties”, *Journal of Global Optimization*, 9, 453-482.

Kempf, K., Uzsoy, R., Smith, S., and Gary, K., (2000), “Evaluation and comparison of production schedules”, *Journal of Industry*, 42, 2-3, 203-20.

Klein, R., Luss, H. and Rothblum, U., (1994), “Relaxation-based algorithms for minimax optimization problems with resource allocation applications”, *Mathematical Programming*, 64, 337-363.

Klein, R., Luss, H. and Rothblum, U., (1995), “Multiperiod allocation of substitutable resources”, *European Journal of Operational Research*, 85, 488-503.

Kolisch, R., (2000), “Integrated scheduling, assembly area and part-assignment for large-scale, make-to-order assemble”, *International Journal of Production Economics*, 64, 127-141.

Lawler E. L., Lenstra J. K., Rinnooy Kan A. H. G. and Shmoys D.B. , (1993), *Sequencing and scheduling: algorithms and complexity*, Handbooks in Operations Research and Management Sciences: Logistics of Production and Inventory, North-Holland, Amsterdam, 445–522.

Luss, H. and Smith, D., (1988), “Multiperiod Application of Limited ResourcesL A Minimax Approach”, *Naval Research Logistics*, 35, 490-501.

Maxwell, W. L., and Mehra, M., (1968), “Multiple-Factor Rules for Sequencing with Assembly Constraints”, *Naval Research Logistics Quarterly*, 15, 241-254.

Moses, S., Grant, H., Gruenwald, L. and Pulat, S., (2003), “Real-time due-date promising by build-to-order environments”, *International Journal of Production Research,* 42, 20, 4353-4375.

Musselman, K and Uzsoy, R., (2001), “Advanced Planning and Scheduling for Manufacturing”, *Handbook of Industrial Engineering*, 2033-2053.

Muth, J. F. and Thompson, G. L., (1963), *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, NJ.

Ovacik, I. M. and Uzsoy, R., (1995), "Rolling horizon procedures for dynamic parallel machine scheduling with sequence-dependent setup times", *International Journal of Production Research,* 33, 3173-3192.

Ovacik, I. M. and Uzsoy, R., (1997), *Decomposition Methods for Complex Factory Scheduling Problems*, Kluwer Academic Publishers, Noewell, MA.

Park, M., and Kim, Y., (2000), "A branch and bound algorithm for a production scheduling problem in an assembly system under due date constraints", *European Journal of Operational Research*, 123, 504-518.

Park, M., and Kim, Y., (1999), "A heuristic algorithm for a production planning problem in an assembly system", *Journal of Operational Research Society*, 50, 138-147.

Pathumnakul, S., and Egbelu, P., (2006), "An algorithm for minimizing weighted earliness penalty in assembly job shops", *International Journal of Production Economics*, 103, 203-245.

Philipoom, R. P., Russell, S. R., and Fry, D. T., (1991), “A preliminary investigation of multi-attribute based sequencing rules for assembly shops”. *International Journal of Production Research*, 29, 739-753.

Pinedo, M., (1995), *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, Englewood Cliffs, NJ.

Pinedo, M., (2005), *Planning and Scheduling in Manufacturing and Services*, Springer, New York, NY.

Reeja, M. K., and Rajendran, C., (2000), “Dispatching rules for scheduling in assembly jobshops – Part 1”. *International Journal of Production Research*, 38, 9, 2051-2066.

Russell, R. S., and Taylor III, B. W., (1985), “An evaluation of sequencing rules for an assembly shop”. *Decision Science*, 16, 196-212.

Tang, S., (1988), “A Max-Min Application problem: its Solution and Applications”, *Operations Research,* 36, 2, 359-367.

Uzsoy, R. and Wang, C. S., (2000), “Performance of decomposition procedures for job shop scheduling problems with bottleneck machines”, *International Journal of production Research*, 38, 1271-1286.

Appendix A. Example of Input data format – 5 jobs, 5 machines and 5 components

CO: [1 2 3 4 5]

orderNumber: [1 2 3 4 5]

orderRequestedItem: ["2" "2" "1" "2" "1"]

orderQuantity: [1 1 1 1 1]

orderDueDate: [95 79 59 51 101]

orderPriority: [1 1 1 2 1]

SO: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44]

SONumber: ["1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13" "14" "15" "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29" "30" "31" "32" "33" "34" "35" "36" "37" "38" "39" "40" "41" "42" "43" "44"]

parentCO: [1 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 4 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0]

parentSO: [0 1 2 2 2 1 6 6 6 1 0 11 12 12 12 11 11 17 17 17 0 21 21 23 23 23 21 0 28 28 30 30 30 28 0 35 36 36 36 35 35 41 41 41]

SOItem: ["2" "5" "8" "8" "6" "3" "10" "7" "6" "6" "2" "5" "7" "7" "10" "10" "4" "7" "9" "10" "1" "6" "5" "10" "8" "10" "8" "2" "9" "3" "6" "8" "9" "9" "1" "4" "8" "9" "7" "8" "4" "7" "8" "9"]

SOQuantity: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

SOPriority: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1]

requestedMachine: ["1" "3" "0" "0" "0" "5" "0" "0" "0" "0" "3" "5" "0" "0" "0" "0" "2" "0" "0" "0" "4" "0" "2" "0" "0" "0" "0" "4" "0" "3" "0" "0" "0" "0" "1" "1" "0" "0" "0" "0" "2" "0" "0" "0"]

OP: [10 17 0 0 0 24 0 0 0 0 7 14 0 0 0 0 21 0 0 0 8 0 26 0 0 0 0 13 0 17 0 0 0 0 15 25 0 0 0 0 11 0 0 0]

parentCO1: [1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5]

assemblyRequired: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Part: [1 2 3 4 5 6 7 8 9 10]

partNumber: [1 2 3 4 5 6 7 8 9 10]

partID: ["1" "2" "3" "4" "5" "6" "7" "8" "9" "10"]

RWType: [0 0 0 0 0 1 1 1 1 1]

ASType: [0 0 1 1 1 0 0 0 0 0]

EDType: [1 1 0 0 0 0 0 0 0 0]

PO: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]

PONumber: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]

POItem: ["10" "9" "10" "8" "10" "6" "10" "9" "6" "10" "6" "10" "6" "10" "10" "7" "7" "8" "8"]

POTime: [13 8 27 17 40 7 55 15 30 61 51 70 74 75 80 23 34 38 43]

POQuantity: [3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3]

MC: [1 2 3 4 5]

MachineNumber: [1 2 3 4 5]

MachineID: ["1" "2" "3" "4" "5"]

Appendix B. Source Code of Xpress-MP Model – MIP

```
model MA
uses "mmxprs", "mmsystem"   !Xpress-Optimizer

parameters
   inputFileName = 'MA.dat'
   outputFileName = 'out.dat'
   maxCPUTime = 10
   maxCPUTimeSelection = 5
   maxCPUTimeReOpt = 10
   maxCPUTimeFullReOpt = 10
   rollingInterval = 20
   overlapInterval = 10
end-parameters

forward procedure printResult

declarations
  CO:set of integer         !Customer Orders
  SO: set of integer        !Shop Order
  Part: set of integer      !Parts
  PO: set of integer        !Purchase Orders
  MC: set of integer         !Machines
end-declarations


!initializations from '5CO_25PO_2M_MOSES.dat'
!initializations from '10CO_50PO_2M_MOSES.dat'
!initializations from '20CO_100PO_2M_MOSES.dat'
!initializations from '40CO_200PO_2M_MOSES.dat'
initializations from inputFileName
  CO SO Part PO MC
end-initializations

finalize(CO)
finalize(SO)
finalize(Part)
finalize(PO)
finalize(MC)


declarations
  !Customer Orders
  orderNumber: array(CO) of integer
  orderRequestedItem: array(CO) of string
  orderQuantity: array(CO) of integer
  orderDueDate: array(CO) of integer
  orderPET: array(CO) of mpvar
  orderPriority: array(CO) of integer
  orderTardiness: array(CO) of mpvar

  !Shop Order
  SONumber: array(SO) of string
```

```
parentCO: array(SO) of integer
parentSO: array(SO) of integer
SOItem: array(SO) of string
SOQuantity: array(SO) of integer
SOPriority: array(SO) of integer
requestedMachine: array(SO) of string
OP: array(SO) of integer
PST: array(SO) of mpvar    !Desicion variables
PET: array(SO) of mpvar       !Desicion variables
LL: array(SO) of integer
parentCO1: array(SO) of integer

!Parts
partNumber: array(Part) of integer
partID: array(Part) of string
RWType: array(Part) of integer
ASType: array(Part) of integer
EDType: array(Part) of integer

!Purchase Orders
PONumber: array(PO) of integer
POItem: array(PO) of string
POTime: array(PO) of integer
POQuantity: array(PO) of integer

!Machines
MachineNumber: array(MC) of integer
MachineID: array(MC) of string

TM: array(SO, PO) of mpvar    !Desicion variables
AM: array(SO, PO) of mpvar    !Desicion variables
AC: array(SO, SO) of mpvar    !Desicion variables

BigM: real
starttime: real

MaterialST1: array(PO) of linctr
MaterialST2: array(SO) of linctr
MaterialST3: array(SO, PO) of linctr
MaterialST4: array(SO, PO) of linctr
MaterialST5: array(SO, PO) of linctr
CapacityST1: array(SO, SO) of linctr
CapacityST2: array(SO, SO) of linctr
CapacityST3: array(SO, SO) of linctr
PrecST1: array(SO) of linctr
PrecST2: array(SO, SO) of linctr
PrecST3: array(CO, SO) of linctr

NumberOfSM: real
TotalCUPTimeSM: real
TotalGapSM: real
NumberOfSP: real
TotalCUPTimeSP: real
TotalGapSP: real

NumberOfSR: real
```

```
  TotalCUPTimeSR: real
  TotalGapSR: real

  initialSolDispatch: real
  initialSolRHP: real
  initialSolSB: real

end-declarations


!initializations from '5CO_25PO_2M_MOSES.dat'
!initializations from '10CO_50PO_2M_MOSES.dat'
!initializations from '20CO_100PO_2M_MOSES.dat'
!initializations from '40CO_200PO_2M_MOSES.dat'
initializations from inputFileName

  ! CO (Customer Orders) data
  orderNumber orderRequestedItem orderQuantity orderDueDate orderPriority

  ! SO (Shop Order)
  SONumber parentCO parentSO SOItem SOQuantity SOPriority requestedMachine OP parentCO1

  !Part (Parts)
  partNumber partID RWType ASType EDType

  !PO (Purchase Orders)
  PONumber POItem POTime POQuantity

  !MC (Machines)
  MachineNumber MachineID
end-initializations


!-----------------------------------------------------------------

starttime:= gettime
BigM:= 2000000

!Objective
PrimalObj:= sum(k in CO) orderPriority(k)*orderQuantity(k)*orderTardiness(k)
!PrimalObj:= sum(k in CO) orderQuantity(k)*orderTardiness(k)


forall(k in CO) orderTardiness(k) >= orderPET(k)-orderDueDate(k)
forall(k in CO) orderTardiness(k) >= 0


!--------------Material constraints

!Limits on PO Quantity
forall(j in Part | RWType(j) = 1) do
   forall(s in PO | POItem(s) = partID(j) )
     MaterialST1(s):= sum(i in SO | SOItem(i) =  partID(j) ) TM(i,s) <= POQuantity(s)
end-do

!Satisfy SO Requirements
```

```
forall(j in Part | RWType(j) = 1) do
  forall(i in SO | SOItem(i) = partID(j) )
    MaterialST2(i):= sum(s in PO |POItem(s) = partID(j) ) TM(i,s) = SOQuantity(i)
end-do


forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST3(i,s):=  PST(i) >= POTime(s)*AM(i,s)
  end-do
end-do

! if TW(s,i) > 0, AW = 1, else, AW = 0

forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST4(i,s):= TM(i,s) >= AM(i,s)
  end-do
end-do


forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST5(i,s):= TM(i,s)<= 1000*AM(i,s)
  end-do
end-do



!-----------Capacity constraints
!if PST(i) >= PET(j), then AC(i,j) =1 , else, AC(i,j) = 0

forall(m in MC) do
          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
       CapacityST1(i,ii):= PET(i) <= PST(ii) + BigM*AC(i, ii)
   end-do
end-do

forall(m in MC) do
          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
       CapacityST3(i,ii):= AC(i,ii) + AC(ii,i) = 1
   end-do
end-do


!forall(i in SO, s in PO) AM(i,s) <= 1
!forall(i in SO, s in PO) AM(i,s) >= 0
!forall(i in SO, ii in SO) AC(i,ii) <=1
!forall(i in SO, ii in SO) AC(i,ii) >=0
```

```
!------------Precedence constraints

forall(i in SO) PrecST1(i):= PET(i) = PST(i) + OP(i)

!forall(i in SO) PST(i) <= 1100000000
forall(i in SO) PST(i) <= 1000000

forall(i in SO) do
  forall(ii in SO | parentSO(ii) = i) PrecST2(i, ii):= PST(i) >= PET(ii)
end-do

forall(k in CO) do
   forall(i in SO | parentCO(i) = k) PrecST3(k, i):= orderPET(k) = PET(i)
end-do



forall(i in SO) PST(i) is_integer
forall(i in SO) PET(i) is_integer
forall(i in SO, s in PO) AM(i,s) is_binary
forall(i in SO, s in PO) TM(i,s) is_integer
forall(i in SO, ii in SO) AC(i,ii) is_binary



declarations
  status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
end-declarations

starttime:= gettime


setparam("XPRS_SCALING",0)
setparam("XPRS_FEASTOL",1.0E-10)
setparam("XPRS_MIPTOL",1.0E-9)

setparam("XPRS_MAXTIME",maxCPUTime)
setparam("XPRS_MAXMIPSOL",0)
minimize(PrimalObj)

status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
writeln('Primal is ', status(getprobstat), ",Objective: ",getobjval )
writeln(" CPU time: " , gettime-starttime)
writeln('best bound is ', getparam("XPRS_BESTBOUND"))
writeln('Number of integer found is ', getparam("XPRS_MIPSOLS"))


forall(i in SO) writeln("SO--PST(", i,"):", getsol(PST(i)), ";PET(",i,"):", getsol(PET(i)), ";OP(",i,"):", getsol(OP(i)) )
writeln('Primal is ', status(getprobstat), ",Objective: ",getobjval )
writeln(" CPU time: " , gettime-starttime)
writeln(" BigM: " , getsol(BigM))
forall(k in CO) writeln("CO--orderTardiness(", k,"):", getsol(orderTardiness(k)))
forall(k in CO) writeln("CO--PET(", k,"):", getsol(orderPET(k)))
forall(k in CO) writeln("CO--DueDate(", k,"):", getsol(orderDueDate(k)))
forall(k in CO) writeln("CO--PET(", k,"):", getsol(orderPET(k)))
```

```
!forall(i in SO) writeln("SO--PST(", i,"):", getsol(PST(i)), ";PET(",i,"):", getsol(PET(i)), ";OP(",i,"):", getsol(OP(i)) )

End-model
```

Appendix C. Source Code of Xpress-MP Model – MSBP

```
model MA
uses "mmxprs", "mmsystem"   !Xpress-Optimizer

parameters
  inputFileName = 'MA.dat'
  outputFileName = 'out.dat'
  maxCPUTime = 10
  maxCPUTimeSelection = 5
  maxCPUTimeReOpt = 10
  maxCPUTimeFullReOpt = 10
  rollingInterval = 20
  overlapInterval = 10

  machineSubMethod = 0 !*** 0-Opt, 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT, 10-OPT(withlongestPath)
  partSubMethod = 0    !*** 0-Opt, 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT, 10-OPT(withlongestPath)


end-parameters


forward procedure SB

forward procedure partSubproblem(currentPartID:string)

forward procedure machineSubproblem(currentMachineID:string)

forward procedure partSubproblemOpt(currentPartID:string)

forward procedure machineSubproblemOpt(currentMachineID:string)

forward procedure partSubproblemDsp(currentPartID:string)

forward procedure machineSubproblemDsp(currentMachineID:string)

forward procedure partSubproblemOptSP(currentPartID:string)

forward procedure machineSubproblemOptSP(currentMachineID:string)

forward procedure partSubproblemDspSP(currentPartID:string)

forward procedure machineSubproblemDspSP(currentMachineID:string)

forward function getInfoToCO(i: integer):integer

forward function getNumberOfOP(i, j: integer):integer

forward function getInfoSlack(i: integer):integer


forward procedure checkSolution
```

```
forward function getLongestPath(i, j: integer):integer

forward function getSOParnetCO(i: integer):integer

forward procedure push(i, j: integer)

forward procedure pull(i, j: integer)

forward procedure pushcycle(i, j, k: integer)

forward function getLongestPath1(i, j: integer):integer

forward procedure push1(i, j: integer, nodePush: boolean)

forward procedure printResult


declarations
  CO:set of integer       !Customer Orders
  SO: set of integer      !Shop Order
  Part: set of integer    !Parts
  PO: set of integer      !Purchase Orders
  MC: set of integer       !Machines
end-declarations


initializations from inputFileName
  CO SO Part PO MC
end-initializations

finalize(CO)
finalize(SO)
finalize(Part)
finalize(PO)
finalize(MC)


declarations
  !Customer Orders
  orderNumber: array(CO) of integer
  orderRequestedItem: array(CO) of string
  orderQuantity: array(CO) of integer
  orderDueDate: array(CO) of integer
  orderPET: array(CO) of mpvar
  orderPriority: array(CO) of integer
  orderTardiness: array(CO) of mpvar

  !Shop Order
  SONumber: array(SO) of string
  parentCO: array(SO) of integer
  parentSO: array(SO) of integer
  SOItem: array(SO) of string
  SOQuantity: array(SO) of integer
  SOPriority: array(SO) of integer
```

```
requestedMachine: array(SO) of string
OP: array(SO) of integer
PST: array(SO) of mpvar    !Desicion variables
PET: array(SO) of mpvar       !Desicion variables
LL: array(SO) of integer
parentCO1: array(SO) of integer
numberOfIncomingArc: array(SO) of integer
numberOfToken: array(SO) of integer
dueDate: array(SO) of integer

!Parts
partNumber: array(Part) of integer
partID: array(Part) of string
RWType: array(Part) of integer
ASType: array(Part) of integer
EDType: array(Part) of integer
PartCV: array(Part) of real
PartDone: array(Part) of boolean

!Purchase Orders
PONumber: array(PO) of integer
POItem: array(PO) of string
POTime: array(PO) of integer
POQuantity: array(PO) of integer

!Machines
MachineNumber: array(MC) of integer
MachineID: array(MC) of string
MachineCV: array(MC) of real
MachineDone: array(MC) of boolean

TM: array(SO, PO) of mpvar    !Desicion variables
AM: array(SO, PO) of mpvar    !Desicion variables
AC: array(SO, SO) of mpvar    !Desicion variables

BigM: real
starttime: real
maxCV: real
allDone: boolean


MaterialST1: array(PO) of linctr
MaterialST2: array(SO) of linctr
MaterialST3: array(SO, PO) of linctr
MaterialST4: array(SO, PO) of linctr
MaterialST5: array(SO, PO) of linctr
CapacityST1: array(SO, SO) of linctr
CapacityST2: array(SO, SO) of linctr
CapacityST3: array(SO, SO) of linctr
CapacityST4: array(SO, SO) of linctr
PrecST1: array(SO) of linctr
PrecST2: array(SO, SO) of linctr
PrecST3: array(CO, SO) of linctr
PrecST4: array(SO) of linctr
PrecST5: array(SO, CO) of linctr
Tard1: array(CO) of linctr
```

```
Tard2: array(CO) of linctr
fixedAC: array(SO,SO) of linctr
fixedAM: array(SO,PO) of linctr
tempAC: array(SO, SO) of real
tempAM: array(SO, SO) of real
DelayPrecST: array(SO,SO) of linctr

UE: array(SO) of real
UE_st: array(SO) of real
UE_temp: array(SO) of real
UE1: array(SO, CO) of real
UE1_st: array(SO, CO) of real
UE1_temp: array(SO, CO) of real

tk: real
iteration: real
BestObjPrime: real

PathFrom: array(SO) of integer
Flow: array(SO) of integer
UpdatePathFrom: boolean
UpdatePathFrom1: array(SO) of boolean
counter: integer
sumAC: real
sumAC1: real

PathToSO: array(SO, SO) of integer
UpdatePathToSO: array(SO) of boolean
nodePush: array(SO) of boolean

 NumberOfSM: real
TotalCUPTimeSM: real
TotalGapSM: real
NumberOfSP: real
TotalCUPTimeSP: real
TotalGapSP: real

NumberOfSR: real
TotalCUPTimeSR: real
TotalGapSR: real

initialSolDispatch: real
initialSolRHP: real
initialSolSB: real

TotalROWS: real
TotalCOLS: real
TotalELEMS: real
TotalMIPENTS: real

SIndex: array(SO) of real
readySeq: array(SO) of integer
POQuantityS: array(PO) of integer

numberOfOPS: array(SO) of integer
pathToCO: array(SO) of integer
```

```
  numberOfOPSSlack: array(SO) of integer
  pathToCOSlack: array(SO) of integer

end-declarations


initializations from inputFileName
  ! CO (Customer Orders) data
  orderNumber orderRequestedItem orderQuantity orderDueDate orderPriority

  ! SO (Shop Order)
  SONumber parentCO parentSO SOItem SOQuantity SOPriority requestedMachine OP parentCO1

  !Part (Parts)
  partNumber partID RWType ASType EDType

  !PO (Purchase Orders)
  PONumber POItem POTime POQuantity

  !MC (Machines)
  MachineNumber MachineID
end-initializations


!------------------------------------------------------------------

starttime:= gettime
BigM := 10000
!Objective
PrimalObj:= sum(k in CO) orderPriority(k)*orderQuantity(k)*orderTardiness(k)

forall(k in CO) Tard1(k) := orderTardiness(k) >= orderPET(k)-orderDueDate(k)
forall(k in CO) Tard2(k) := orderTardiness(k) >= 0


!--------------Material constraints

!Limits on PO Quantity
forall(j in Part | RWType(j) = 1) do
   forall(s in PO | POItem(s) = partID(j) )
     MaterialST1(s):= sum(i in SO | SOItem(i) =  partID(j) ) TM(i,s) <= POQuantity(s)
end-do

!Satisfy SO Requirements
forall(j in Part | RWType(j) = 1) do
   forall(i in SO | SOItem(i) = partID(j) )
     MaterialST2(i):= sum(s in PO |POItem(s) = partID(j) ) TM(i,s) = SOQuantity(i)
end-do


forall(j in Part | RWType(j) = 1) do
   forall(i in SO| SOItem(i) = partID(j)) do
      forall(s in PO | POItem(s) = partID(j) )
        MaterialST3(i,s):=  PST(i) >= POTime(s)*AM(i,s)
   end-do
```

```
end-do

! if TW(s,i) > 0, AW = 1, else, AW = 0

forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST4(i,s):= TM(i,s) >= AM(i,s)
  end-do
end-do


forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST5(i,s):= TM(i,s) <= 1000* AM(i,s)
  end-do
end-do



!-----------Capacity constraints
!if PST(i) >= PET(j), then AC(i,j) =1 , else, AC(i,j) = 0

forall(m in MC) do
          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
       CapacityST1(i,ii):= PET(i) <= PST(ii) + BigM*AC(i, ii)
   end-do
end-do

(!
forall(m in MC) do
   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
       CapacityST2(i,ii):= PET(ii) <= PST(i) + BigM*(1-AC(i, ii))
   end-do
end-do
!)

forall(m in MC) do
          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
       CapacityST3(i,ii):= AC(i,ii) + AC(ii,i) = 1
   end-do
end-do


!------------Precedence constraints
forall(i in SO) PrecST1(i):= PET(i) = PST(i) + OP(i)

forall(i in SO) do
 forall(ii in SO | parentSO(ii) = i) PrecST2(i, ii):= PST(i) >= PET(ii)
end-do

forall(k in CO) do
```

```
  forall(i in SO | parentCO(i) = k) PrecST3(k, i):= orderPET(k) = PET(i)
end-do

!forall(i in SO) PST(i) <= 8000000
forall(i in SO) PST(i) <= 10000
forall(i in SO) PST(i) is_integer
forall(i in SO) PET(i) is_integer
forall(i in SO, s in PO) AM(i,s) is_binary
forall(i in SO, s in PO) TM(i,s) is_integer
forall(i in SO, ii in SO) AC(i,ii) is_binary


!********* perform main program

SB

checkSolution

printResult

!********* End of main program


procedure SB

declarations

status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string

end-declarations

starttime := gettime


! Relax material constraints
forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) ) do
      sethidden(MaterialST3(i,s), true)
    end-do
  end-do
end-do


forall(m in MC) do
         forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
       sethidden(CapacityST1(i,ii), true)
       sethidden(CapacityST3(i,ii), true)
       tempAC(i,ii):= 0
       fixedAC(i,ii):= AC(i,ii) = tempAC(i,ii)
     end-do
   end-do
end-do
```

```
!Reset all values
forall(m in MC) do
        machineDone(m) := false
end-do

forall(j in Part | RWType(j) = 1) do
        partDone(j) := false
end-do

allDone := false

setparam("XPRS_MAXTIME",maxCPUTimeSelection)



!start SB --Part
while( allDone = false) do
  maxCV := -1
  finishSubproblem := false


  forall(j in Part | RWType(j) = 1 and PartDone(j) = false) do
    partSubproblem( partID(j) )

    maxCV := maxlist(maxCV, getsol(PartCV(j)) )
  end-do
 writeln("maxCV(endSelection):", maxCV)

 forall(j in Part | RWType(j) = 1 and PartDone(j) = false) do
   if(maxCV = getsol(PartCV(j)) and finishSubproblem = false) then
     setparam("XPRS_MAXTIME",maxCPUTime)
     partSubproblem( partID(j) )
     setparam("XPRS_MAXTIME",maxCPUTimeSelection)

     forall(i in SO| SOItem(i) = partID(j)) do
             forall(s in PO | POItem(s) = partID(j) )
             sethidden(MaterialST3(i,s), false)
           end-do
     !fixed TM(i,s)
                 forall(i in SO| SOItem(i) = partID(j)) do
                   forall(s in PO | POItem(s) = partID(j) ) do
                     tempAM(i,s):= getsol(AM(i,s))
                     fixedAM(i,s):= AM(i,s) = tempAM(i,s)

                    end-do
                 end-do

     finishSubproblem := true
     PartDone(j) := true
   end-if
 end-do

 writeln("maxCV(endSub):", maxCV)

!---part---- reopt
```

```
objective := getobjval
 writeln("CurrentObj(BegReopt):", getobjval)

 forall(j in Part | RWType(j) = 1 and PartDone(j) = true) do
      forall(i in SO| SOItem(i) = partID(j)) do
                 forall(s in PO | POItem(s) = partID(j) ) do
                   sethidden(MaterialST3(i,s), true)
                   sethidden(fixedAM(i,s), true)
                 end-do
              end-do
              setparam("XPRS_MAXTIME",maxCPUTimeReOpt)
      partSubproblem( partID(j) )
      setparam("XPRS_MAXTIME",maxCPUTimeSelection)
      forall(i in SO| SOItem(i) = partID(j)) do
                 forall(s in PO | POItem(s) = partID(j) )
                 sethidden(MaterialST3(i,s), false)
              end-do
      !fixed TM(i,s)
      if(getobjval < objective) then
                                forall(i in SO| SOItem(i) = partID(j)) do
                                   forall(s in PO | POItem(s) = partID(j) ) do
                                      tempAM(i,s):= getsol(AM(i,s))
                                      fixedAM(i,s):= AM(i,s) = tempAM(i,s)

                                    end-do
                                end-do
                                objective := getobjval
                                writeln("BestObj:", getobjval)
                      end-if

                      forall(i in SO| SOItem(i) = partID(j)) do
                         forall(s in PO | POItem(s) = partID(j) ) do
                            sethidden(fixedAM(i,s), false)
                          end-do
                      end-do

 end-do


 !check allDone Value
 allDone := true


 forall(j in Part | RWType(j) = 1 and PartDone(j) = false) do
    allDone := false
 end-do

end-do


!Reset all values
forall(m in MC) do
           machineDone(m) := false
end-do

forall(j in Part | RWType(j) = 1) do
```

```
          partDone(j) := false
end-do

allDone := false



!start SB  ---machine

while( allDone = false) do
  maxCV := -1
  finishSubproblem := false

  forall(m in MC| MachineDone(m) = false) do
    machineSubproblem( MachineID(m) )

    maxCV := maxlist(maxCV, getsol(MachineCV(m)) )
  end-do



 forall(m in MC| MachineDone(m) = false) do
   if(maxCV = getsol(MachineCV(m)) and finishSubproblem = false) then
     setparam("XPRS_MAXTIME",maxCPUTime)
     machineSubproblem( MachineID(m) )
     setparam("XPRS_MAXTIME",maxCPUTimeSelection)
    forall(i in SO|requestedMachine(i) = MachineID(m) ) do
            forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
              sethidden(CapacityST1(i,ii), false)
              sethidden(CapacityST3(i,ii), false)
            end-do
          end-do

          writeln("maxCV(endSelection):", maxCV)

                !fixed AC(i,ii)
          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
            forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
              tempAC(i,ii):= getsol(AC(i,ii))
              fixedAC(i,ii):= AC(i,ii) = tempAC(i,ii)
            end-do
          end-do

     finishSubproblem := true
     MachineDone(m) := true
   end-if
 end-do


 !reoptimization


objective := getobjval

writeln("CurrentObj-(Before Reopt-Part):", getobjval)
```

```
setparam("XPRS_MAXTIME",maxCPUTimeReOpt)

forall(j in Part | RWType(j) = 1 and PartDone(j) = true) do
    forall(i in SO| SOItem(i) = partID(j)) do
            forall(s in PO | POItem(s) = partID(j) ) do
              sethidden(MaterialST3(i,s), true)
              sethidden(fixedAM(i,s), true)
            end-do
          end-do
          !setparam("XPRS_MAXTIME",maxCPUTimeReOpt)
    partSubproblem( partID(j) )
    !setparam("XPRS_MAXTIME",maxCPUTimeSelection)
    forall(i in SO| SOItem(i) = partID(j)) do
            forall(s in PO | POItem(s) = partID(j) )
            sethidden(MaterialST3(i,s), false)
          end-do
    !fixed TM(i,s)
    if(getobjval < objective) then
                          forall(i in SO| SOItem(i) = partID(j)) do
                            forall(s in PO | POItem(s) = partID(j) ) do
                              tempAM(i,s):= getsol(AM(i,s))
                              fixedAM(i,s):= AM(i,s) = tempAM(i,s)
                             end-do
                          end-do
                          objective := getobjval
                          writeln("BestObj:", getobjval)
                  end-if

                  forall(i in SO| SOItem(i) = partID(j)) do
                    forall(s in PO | POItem(s) = partID(j) ) do
                      sethidden(fixedAM(i,s), false)
                     end-do
                  end-do

end-do

setparam("XPRS_MAXTIME",maxCPUTimeFullReOpt)

writeln("maxCV:", maxCV)
writeln("CurrentObj:(Before Reopt-Machine):", objective)

forall(m in MC| MachineDone(m) = true) do
    forall(i in SO|requestedMachine(i) = MachineID(m) ) do
            forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
              sethidden(CapacityST1(i,ii), true)
              sethidden(CapacityST3(i,ii), true)
              sethidden(fixedAC(i,ii), true)
            end-do
          end-do

    machineSubproblem( MachineID(m) )

   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
            forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
              sethidden(CapacityST1(i,ii), false)
              sethidden(CapacityST3(i,ii), false)
```

```
              end-do
            end-do

      if(getobjval < objective) then
                              !fixed AC(i,ii)
                       forall(i in SO|requestedMachine(i) = MachineID(m) ) do
                         forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
                           tempAC(i,ii):= getsol(AC(i,ii))
                           fixedAC(i,ii):= AC(i,ii) = tempAC(i,ii)
                         end-do
                       end-do
                     objective := getobjval
                              writeln("BestObj:", getobjval)
                     end-if

                     forall(i in SO|requestedMachine(i) = MachineID(m) ) do
              forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
                sethidden(fixedAC(i,ii), false)
              end-do
            end-do
  end-do

  !check allDone Value
  allDone := true

  forall(m in MC| MachineDone(m) = false) do
    allDone := false
  end-do

end-do


initialSolSB := getobjval


count:= 1
maxCount := 3
currentObj := getobjval
bestObj := getobjval + 1

while( count <= maxCount and currentObj < bestObj) do
  bestObj := getobjval

  writeln("CurrentObj(Full-Opt):", getobjval, ";Count:", count )


  writeln("CurrentObj(Full-Opt):", getobjval)


  forall(j in Part | RWType(j) = 1 and PartDone(j) = true) do
      forall(i in SO| SOItem(i) = partID(j)) do
              forall(s in PO | POItem(s) = partID(j) ) do
                sethidden(MaterialST3(i,s), true)
                sethidden(fixedAM(i,s), true)
              end-do
            end-do
```

```
      partSubproblem( partID(j) )

      forall(i in SO| SOItem(i) = partID(j)) do
               forall(s in PO | POItem(s) = partID(j) )
               sethidden(MaterialST3(i,s), false)
             end-do
      !fixed TM(i,s)
      if(getobjval < objective) then
                              forall(i in SO| SOItem(i) = partID(j)) do
                                forall(s in PO | POItem(s) = partID(j) ) do
                                  tempAM(i,s):= getsol(AM(i,s))
                                  fixedAM(i,s):= AM(i,s) = tempAM(i,s)
                                 end-do
                              end-do
                              objective := getobjval
                              writeln("BestObj:", getobjval)
                    end-if

                    forall(i in SO| SOItem(i) = partID(j)) do
                      forall(s in PO | POItem(s) = partID(j) ) do
                        sethidden(fixedAM(i,s), false)
                       end-do
                    end-do

  end-do

 !objective := getobjval
  writeln("CurrentObj:", getobjval)

  !reoptimization
  forall(m in MC| MachineDone(m) = true) do
      forall(i in SO|requestedMachine(i) = MachineID(m) ) do
               forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
                 sethidden(CapacityST1(i,ii), true)
                 sethidden(CapacityST3(i,ii), true)
                 sethidden(fixedAC(i,ii), true)
               end-do
             end-do

      machineSubproblem( MachineID(m) )

     forall(i in SO|requestedMachine(i) = MachineID(m) ) do
               forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
                 sethidden(CapacityST1(i,ii), false)
                 sethidden(CapacityST3(i,ii), false)
               end-do
             end-do

      if(getobjval < objective) then
                              !fixed AC(i,ii)
                        forall(i in SO|requestedMachine(i) = MachineID(m) ) do
                          forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
                            tempAC(i,ii):= getsol(AC(i,ii))
                            fixedAC(i,ii):= AC(i,ii) = tempAC(i,ii)
                          end-do
```

```
                end-do
              objective := getobjval
                      writeln("BestObj:", getobjval)
              end-if

              forall(i in SO|requestedMachine(i) = MachineID(m) ) do
          forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
            sethidden(fixedAC(i,ii), false)
          end-do
        end-do


  end-do

count:= count +1
  currentObj := getobjval
end-do


! unHidden all material constraints
forall(j in Part | RWType(j) = 1) do
    forall(s in PO | POItem(s) = partID(j) ) do
      sethidden(MaterialST1(s), false)
  end-do
end-do


forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    sethidden(MaterialST2(i), false)
  end-do
end-do

forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) ) do
      sethidden(MaterialST3(i,s), false)
      sethidden(MaterialST4(i,s), false)
      sethidden(MaterialST5(i,s), false)
    end-do
  end-do
end-do

!unHidden all capacity constraints
forall(m in MC) do
        forall(i in SO|requestedMachine(i) = MachineID(m) ) do
    forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
      sethidden(CapacityST1(i,ii), false)
      sethidden(CapacityST3(i,ii), false)
    end-do
  end-do
end-do

!unHidden precedence constaint
forall(i in SO) do
  forall(ii in SO | parentSO(ii) = i) sethidden(PrecST2(i, ii), false)
```

```
end-do

forall(k in CO) do
  forall(i in SO | parentCO(i) = k) sethidden(PrecST3(k, i),false)
end-do


 !Add precedence constraints
 forall(m in MC) do
   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     sethidden(PrecST4(i),true)
   end-do
 end-do



forall(m in MC) do
  forall(i in SO|requestedMachine(i) = MachineID(m) ) do
   forall(k in CO) do
       sethidden(PrecST5(i, k),true)
   end-do
  end-do
end-do

end-procedure


procedure partSubproblem(currentPartID:string)

(!
  machineSubMethod = 0 !*** 0-Opt, 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT, 10-OPT(withLongestPath)
  partSubMethod = 0    !*** 0-Opt, 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT, 10-OPT(withLongestPath)
!)

case partSubMethod of
  0: partSubproblemOpt(currentPartID)
  10: partSubproblemOptSP(currentPartID)
  1..9: partSubproblemDsp(currentPartID)
  !11: partSubproblemDspSP(currentPartID)

end-case


end-procedure



procedure machineSubproblem(currentMachineID:string)

case machineSubMethod of
  0: machineSubproblemOpt(currentMachineID)
  10: machineSubproblemOptSP(currentMachineID)
  1..9: machineSubproblemDsp(currentMachineID)
  !11: machineSubproblemDspSP(currentMachineID)
```

```
end-case

end-procedure

!***** part-subproblem optimization **********

procedure partSubproblemOpt(currentPartID:string)

declarations
originalPrimalObj: real
primalObjective: real
bestPrimalObjective: real
tempBestPrimalObj: real
dualObj: real
objective: real
dual_cap: real
lamda: real
countLamda: real
fixedPST: array(SO) of linctr    !Desicion variables
status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
fixedTM: array(SO, PO) of linctr    !Desicion variables
fixedAM: array(SO, PO) of linctr    !Desicion variables
fixedAC: array(SO, SO) of linctr    !Desicion variables
tempTM: array(SO, PO) of real
tempAM: array(SO, PO) of real
tempAC: array(SO, SO) of real
firstrun: boolean

end-declarations

startTimeSP := gettime


!Start from raw materials
forall(j in Part | RWType(j) = 1 and partID(j) = currentPartID) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      sethidden(MaterialST3(i,s), false)
  end-do
  setparam("XPRS_SCALING",0)
  setparam("XPRS_MAXMIPSOL",0)
  !setparam("XPRS_MAXTIME",10)

  minimize(PrimalObj)
  status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
  writeln('Solution after part-', j ,' is ', status(getprobstat), ",Objective: ",getobjval )
  !writeln("CPU time: " , gettime-starttime)

  !Record CV
  PartCV(j) := getobjval


  forall(i in SO| SOItem(i) = partID(j)) do
```

```
    forall(s in PO | POItem(s) = partID(j) )
      sethidden(MaterialST3(i,s), true)
  end-do

end-do

NumberOfSP := NumberOfSP + 1
TotalCUPTimeSP := TotalCUPTimeSP + (gettime - startTimeSP)
if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSP := TotalGapSP + 0
else
  TotalGapSP := TotalGapSP + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure




!***** machine-subproblem optimization **********


procedure machineSubproblemOpt(currentMachineID:string)

declarations
originalPrimalObj: real
primalObjective: real
bestPrimalObjective: real
tempBestPrimalObj: real
dualObj: real
objective: real
dual_cap: real
lamda: real
countLamda: real
fixedPST: array(SO) of linctr    !Desicion variables
status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
fixedTM: array(SO, PO) of linctr    !Desicion variables
fixedAM: array(SO, PO) of linctr    !Desicion variables

!fixedAC: array(SO, SO) of linctr    !Desicion variables
tempTM: array(SO, PO) of real
tempAM: array(SO, PO) of real
!tempAC: array(SO, SO) of real
firstrun: boolean

end-declarations

startTimeSM := gettime
```

```
!Start from machines
forall(m in MC | MachineID(m) = currentMachineID) do
            forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
        sethidden(CapacityST1(i,ii), false)
        sethidden(CapacityST3(i,ii), false)
        sethidden(fixedAC(i,ii), true)
       end-do
   end-do
   setparam("XPRS_SCALING",0)
   setparam("XPRS_FEASTOL",1.0E-10)
            setparam("XPRS_MIPTOL",1.0E-9)
   setparam("XPRS_MAXMIPSOL",0)
   !setparam("XPRS_MAXTIME",10)
   minimize(PrimalObj)
   status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
   writeln('Solution after machine-', m ,' is ', status(getprobstat), ",Objective: ",getobjval )

   !Record CV
   MachineCV(m) := getobjval


   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
       sethidden(CapacityST1(i,ii), true)
       sethidden(CapacityST3(i,ii), true)
       sethidden(fixedAC(i,ii), false)
     end-do
   end-do


end-do

forall(m in MC) do
            forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
        sethidden(fixedAC(i,ii),false)
      end-do
   end-do
end-do

NumberOfSM := NumberOfSM + 1
TotalCUPTimeSM := TotalCUPTimeSM + (gettime - startTimeSM)
if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSM := TotalGapSM + 0
else
  TotalGapSM := TotalGapSM + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")
```

```
end-procedure




!***** part-subproblem optimization longest path implmentation **********


procedure partSubproblemOptSP(currentPartID:string)

declarations

status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
fixedAM1: array(SO,PO) of linctr
!tempAM: array(SO,PO) of real

end-declarations

startTimeSP := gettime


!Start from raw materials
forall(j in Part | RWType(j) = 1 and partID(j) = currentPartID) do


  forall(jj in Part | RWType(jj) = 1) do
    forall(i in SO| SOItem(i) = partID(jj)) do
      forall(s in PO | POItem(s) = partID(jj) ) do
        sethidden(MaterialST3(i,s), true)
      end-do
    end-do
  end-do



  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) ) do
     sethidden(MaterialST3(i,s), false)
    end-do
  end-do


  UpdatePathFrom := false

           forall(i in SO)
             UpdatePathFrom1(i):= false

           forall(i in SO) PathFrom(i) := 0

           forall(i in SO) UpdatePathToSO(i) := false
           forall(i in SO, jj in SO) PathToSO(i,jj) := 0


   forall(m in MC ) do
```

```
   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
       sethidden(CapacityST1(i,ii), true)
       sethidden(CapacityST3(i,ii), true)
     end-do
   end-do
 end-do



forall(i in SO) do
          forall(ii in SO | parentSO(ii) = i) sethidden(PrecST2(i, ii), true)
        end-do

        forall(k in CO) do
          forall(i in SO | parentCO(i) = k) sethidden(PrecST3(k, i),true)
        end-do



forall(i in SO| SOItem(i) = partID(j)) do
   forall(k in CO) do
     forall(jj in SO|parentCO(jj) = k) do
               if(getLongestPath1(i,jj) > 0) then
                 PrecST5(i, k):= orderPET(k) >= PST(i) + getLongestPath1(i,jj)+OP(jj)
               end-if
               if(i=jj) then
                 PrecST5(i, k):= orderPET(k) >= PST(i) + OP(i)
               end-if
             end-do
   end-do
 end-do


 setparam("XPRS_SCALING",0)
 !setparam("XPRS_HEURDEPTH",5)
 setparam("XPRS_FEASTOL",1.0E-10)
        !setparam("XPRS_BIGMMETHOD",1)
        !setparam("XPRS_OPTIMALITYTOL",0)
        !setparam("XPRS_PRESOLVE",0)
        setparam("XPRS_MIPTOL",1.0E-9)
 setparam("XPRS_MAXMIPSOL",0)
 !setparam("XPRS_MAXTIME",10)

minimize(PrimalObj)
status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
writeln('Solution after part-', j ,' is ', status(getprobstat), ",Objective: ",getobjval )
!writeln("CPU time: " , gettime-starttime)

!Record CV



forall(i in SO| SOItem(i) = partID(j)) do
   forall(k in CO) do
     forall(jj in SO|parentCO(j) = k) do
```

```
                    if(getLongestPath1(i,jj) > 0) then
                      sethidden(PrecST5(i, k), true)
                    end-if
                    if(i=j) then
                      sethidden(PrecST5(i, k), true)
                    end-if
                  end-do
       end-do
     end-do


   forall(i in SO) do
              forall(ii in SO | parentSO(ii) = i) sethidden(PrecST2(i, ii), false)
            end-do

            forall(k in CO) do
              forall(i in SO | parentCO(i) = k) sethidden(PrecST3(k, i),false)
            end-do


     forall(m in MC | MachineDone(m) = true) do
       forall(i in SO|requestedMachine(i) = MachineID(m) ) do
         forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
           sethidden(CapacityST1(i,ii), false)
           sethidden(CapacityST3(i,ii), false)
         end-do
       end-do
     end-do



forall(i in SO| SOItem(i) = partID(j)) do
   forall(s in PO | POItem(s) = partID(j) ) do
     sethidden( fixedAM1(i,s) , false)
     tempAM(i,s):= getsol(AM(i,s))
     fixedAM1(i,s):= AM(i,s) = tempAM(i,s)

   end-do
end-do

forall(jj in Part | RWType(jj) = 1 and PartDone(jj) = true) do
  forall(i in SO| SOItem(i) = partID(jj)) do
     forall(s in PO | POItem(s) = partID(jj) )
       sethidden(MaterialST3(i,s), false)
   end-do
end-do



  setparam("XPRS_SCALING",0)
   !setparam("XPRS_HEURDEPTH",5)
   setparam("XPRS_FEASTOL",1.0E-10)
            !setparam("XPRS_BIGMMETHOD",1)
            !setparam("XPRS_OPTIMALITYTOL",0)
            !setparam("XPRS_PRESOLVE",0)
            setparam("XPRS_MIPTOL",1.0E-9)
```

```
   setparam("XPRS_MAXMIPSOL",0)
   !setparam("XPRS_MAXTIME",10)

   minimize(PrimalObj)
   status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
   writeln('Solution (Actual) after part-', j ,' is ', status(getprobstat), ",Objective: ",getobjval )
   !writeln("CPU time: " , gettime-starttime2)
   !writeln('best bound is ', getparam("XPRS_BESTBOUND"))
   !writeln('Number of integer found is ', getparam("XPRS_MIPSOLS"))

  PartCV(j) := getobjval



forall(i in SO| SOItem(i) = partID(j)) do
  forall(s in PO | POItem(s) = partID(j) ) do

     sethidden( fixedAM1(i,s) , true)
   end-do
end-do


end-do

NumberOfSP := NumberOfSP + 1
TotalCUPTimeSP := TotalCUPTimeSP + (gettime - startTimeSP)

if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSP := TotalGapSP + 0
else
  TotalGapSP := TotalGapSP + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure




!***** machine-subproblem optimization longest path implmentation **********

procedure machineSubproblemOptSP(currentMachineID:string)

declarations

status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
fixedAC1: array(SO,SO) of linctr
!tempAC1: array(SO, SO) of real


end-declarations
```

```
startTimeSM := gettime

!Start from machines
forall(m in MC | MachineID(m) = currentMachineID) do


  forall(j in Part | RWType(j) = 1) do
          forall(s in PO | POItem(s) = partID(j) ) do
            sethidden(MaterialST1(s), true)
          end-do
        end-do


        forall(j in Part | RWType(j) = 1) do
          forall(i in SO| SOItem(i) = partID(j)) do
            sethidden(MaterialST2(i), true)
          end-do
        end-do

        forall(j in Part | RWType(j) = 1) do
          forall(i in SO| SOItem(i) = partID(j)) do
            forall(s in PO | POItem(s) = partID(j) ) do
              sethidden(MaterialST3(i,s), true)
              sethidden(MaterialST4(i,s), true)
              sethidden(MaterialST5(i,s), true)
            end-do
          end-do
        end-do


  forall(i in SO) do
         forall(ii in SO | parentSO(ii) = i) sethidden(PrecST2(i, ii), true)
        end-do

        forall(k in CO) do
          forall(i in SO | parentCO(i) = k) sethidden(PrecST3(k, i),true)
        end-do

 !---- only need machine m constraints, hide all capacity constraints, then unhidden machine m constraints

  forall(mm in MC ) do
    forall(i in SO|requestedMachine(i) = MachineID(mm) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(mm) and i <> ii) do
        sethidden(CapacityST1(i,ii), true)
        sethidden(CapacityST3(i,ii), true)
      end-do
    end-do
  end-do



        forall(i in SO|requestedMachine(i) = MachineID(m) ) do
    forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
      sethidden(CapacityST1(i,ii), false)
      sethidden(CapacityST3(i,ii), false)
```

```
    sethidden(fixedAC(i,ii), true)
  end-do
end-do

!--------------------



writeln("Machine:", m)
UpdatePathFrom := false

forall(i in SO)
         UpdatePathFrom1(i):= false

forall(i in SO) PathFrom(i) := 0

forall(i in SO) UpdatePathToSO(i) := false
forall(i in SO, j in SO) PathToSO(i,j) := 0


!calculate longest path for (0,i) and (i,k)


!Add precedence constraints
cputime:= gettime

forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  PrecST4(i):=  PST(i) >= getLongestPath(0,i)
end-do

writeln("cpu time:", gettime-cputime)

cputime:= gettime

forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  forall(k in CO) do
    forall(j in SO|parentCO(j) = k) do
              if(getLongestPath1(i,j) > 0) then
                PrecST5(i, k):= orderPET(k) >= PST(i) + getLongestPath1(i,j)+OP(j)
              end-if
              if(i=j) then
                PrecST5(i, k):= orderPET(k) >= PST(i) + OP(i)
              end-if
           end-do
  end-do
end-do
writeln("cpu time:", gettime-cputime)

cputime:= gettime
forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  forall(j in SO|requestedMachine(j) = MachineID(m) and i<>j) do
    if(getLongestPath1(i,j) > 0) then
      DelayPrecST(i,j):= PST(j) >= PST(i) + getLongestPath1(i,j)
    end-if
  end-do
end-do
```

```
writeln("cpu time:", gettime-cputime)


setparam("XPRS_SCALING",0)
!setparam("XPRS_HEURDEPTH",5)
setparam("XPRS_FEASTOL",1.0E-10)
        !setparam("XPRS_BIGMMETHOD",1)
        !setparam("XPRS_OPTIMALITYTOL",0)
        !setparam("XPRS_PRESOLVE",0)
        setparam("XPRS_MIPTOL",1.0E-9)
setparam("XPRS_MAXMIPSOL",0)
!setparam("XPRS_MAXTIME",10)

minimize(PrimalObj)
status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
writeln('Solution after machine-', m ,' is ', status(getprobstat), ",Objective: ",getobjval )


writeln("cycle-CPU time: " , gettime-starttime)


 forall(i in SO|requestedMachine(i) = MachineID(m) ) do
   sethidden(PrecST4(i),true)
 end-do

 forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  forall(k in CO) do
    forall(j in SO|parentCO(j) = k) do
             if(getLongestPath1(i,j) > 0) then
               sethidden(PrecST5(i, k), true)
             end-if
             if(i=j) then
               sethidden(PrecST5(i, k), true)
             end-if
           end-do
  end-do
end-do



 forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  forall(j in SO|requestedMachine(j) = MachineID(m) and i<>j) do
   sethidden(DelayPrecST(i,j),true)
  end-do
end-do



forall(j in Part | RWType(j) = 1) do
          forall(s in PO | POItem(s) = partID(j) ) do
            sethidden(MaterialST1(s), false)
          end-do
        end-do


        forall(j in Part | RWType(j) = 1) do
```

```
            forall(i in SO| SOItem(i) = partID(j)) do
              sethidden(MaterialST2(i), false)
            end-do
          end-do

          forall(j in Part | RWType(j) = 1) do
            forall(i in SO| SOItem(i) = partID(j)) do
              forall(s in PO | POItem(s) = partID(j) ) do
                !sethidden(MaterialST3(i,s), false)
                sethidden(MaterialST4(i,s), false)
                sethidden(MaterialST5(i,s), false)
              end-do
            end-do
          end-do

          forall(j in Part | RWType(j) = 1 and PartDone(j) = true) do
           forall(i in SO| SOItem(i) = partID(j)) do
              forall(s in PO | POItem(s) = partID(j) )
               sethidden(MaterialST3(i,s), false)
            end-do
          end-do

          forall(i in SO) do
           forall(ii in SO | parentSO(ii) = i) sethidden(PrecST2(i, ii), false)
          end-do

          forall(k in CO) do
            forall(i in SO | parentCO(i) = k) sethidden(PrecST3(k, i),false)
          end-do



          !---- unhide all capacity constraints to find actual solution

forall(mm in MC | MachineDone(mm) = true) do
  forall(i in SO|requestedMachine(i) = MachineID(mm) ) do
    forall( ii in SO |requestedMachine(ii) = MachineID(mm) and i <> ii) do
      sethidden(CapacityST1(i,ii), false)
      sethidden(CapacityST3(i,ii), false)

    end-do
  end-do
end-do

          !--------------------



          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do

    tempAC1(i,ii):= round(getsol(AC(i,ii)))
    fixedAC1(i,ii):= AC(i,ii) = tempAC1(i,ii)
    sethidden(fixedAC1(i,ii), false)
    !writeln("AC(",i,",",ii,"):",strfmt(getsol(AC(i,ii)),13),",tempAC:",tempAC(i,ii))
  end-do
```

```
   end-do



   setparam("XPRS_SCALING",0)
   !setparam("XPRS_HEURDEPTH",5)
   setparam("XPRS_FEASTOL",1.0E-10)
          !setparam("XPRS_BIGMMETHOD",1)
          !setparam("XPRS_OPTIMALITYTOL",0)
          !setparam("XPRS_PRESOLVE",0)
          setparam("XPRS_MIPTOL",1.0E-9)
   setparam("XPRS_MAXMIPSOL",0)
   !setparam("XPRS_MAXTIME",10)

   minimize(PrimalObj)
   status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
   writeln('Solution (Actual) after machine-', m ,' is ', status(getprobstat), ",Objective: ",getobjval )
   !writeln("CPU time: " , gettime-starttime)
   !writeln('best bound is ', getparam("XPRS_BESTBOUND"))
   !writeln('Number of integer found is ', getparam("XPRS_MIPSOLS"))


   !Record CV
   MachineCV(m) := getobjval


          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
       sethidden(fixedAC1(i,ii), true)
       !sethidden(fixedAC(i,ii), false)
       !writeln("AC(",i,",",ii,"):",strfmt(getsol(AC(i,ii)),13),",tempAC:",tempAC(i,ii))
     end-do
   end-do


end-do

NumberOfSM := NumberOfSM + 1
TotalCUPTimeSM := TotalCUPTimeSM + (gettime - startTimeSM)
if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSM := TotalGapSM + 0
else
  TotalGapSM := TotalGapSM + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure
```

```
!***** part-subproblem dispatch **********


procedure partSubproblemDsp(currentPartID:string)

declarations

            status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
            fixedAM1: array(SO,PO) of linctr
            tempAM1: array(SO,PO) of real

end-declarations

startTimeSP := gettime

!Start from raw materials
forall(j in Part | RWType(j) = 1 and partID(j) = currentPartID) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      sethidden(MaterialST3(i,s), false)
  end-do


  cOrder := 0
  simulatorClock := 100000
  onhand := 0
  currentPO := 0


  forall(s in PO | POItem(s) = partID(j) ) do
    POQuantityS(s) := POQuantity(s)
  end-do

  forall(i in SO | SOItem(i) = partID(j) ) do
    readySeq(i) := 1
  end-do

   AllPODone := false

            while( AllPODone = false) do
                simulatorClock := 100000

                forall(s in PO | POItem(s) = partID(j) ) do
                  if( POQuantityS(s) > 0 ) then
                    simulatorClock := minlist( simulatorClock, POTime(s) )
                  end-if
                end-do

                forall(s in PO | POItem(s) = partID(j) ) do

           if( POQuantityS(s) > 0 and POTime(s) <= simulatorClock ) then
                    onhand := onhand + POQuantityS(s)
                    POQuantityS(s) := 0
                    currentPO := s
                  end-if
```

```
end-do

 !writeln("nextArrivaltime:" + simulatorClock + ",s:" + currentPO)

forall(i in SO | SOItem(i) = partID(j) ) do
            forall(jj in SO| parentCO(jj) = parentCO1(i) ) do
               SIndex(i) := 0!getLongestPath1(i,jj)
            end-do
          end-do



          forall(i in SO | SOItem(i) = partID(j) and readySeq(i) = 1) do
            if(readySeq(i) = 1 ) then

              case partSubMethod of
                1: do !ATC
                     NOO := getInfoToCO(i)
             temp_exp := -1*maxlist( 0,orderDueDate(parentCO1(i)) - pathToCO(i) -
simulatorClock) /5*17.5
             SIndex(i) :=  SOPriority(i) * exp( temp_exp )
           end-do

         2: do !ATC1
             NOO := getInfoToCO(i)
             temp_SIndex :=  simulatorClock + pathToCO(i) - orderDueDate(parentCO1(i))
                          if( temp_SIndex > 0) then
                            SIndex(i) := temp_SIndex * SOPriority(i)
                          else
                            SIndex(i) := temp_SIndex
                          end-if
           end-do

         5: do  !ODDT
             NOO := getInfoToCO(i)
             temp_SIndex :=  simulatorClock + pathToCO(i) - orderDueDate(parentCO1(i))
                          if( temp_SIndex > 0) then
                            SIndex(i) := temp_SIndex * SOPriority(i)
                          else
                            SIndex(i) := temp_SIndex
                          end-if
           end-do

         7: do  !TWK
             NOOS := getInfoSlack(i)
               SIndex(i) := pathToCOSlack(i)
           end-do

         8: do  !SOPN
             NOOS := getInfoSlack(i)
               SIndex(i) :=  (orderDueDate(parentCO1(i)) - pathToCOSlack(i) - simulatorClock)/
numberOfOPSSlack(i)
           end-do


        end-case
```

```
         end-if
       end-do



!assign raw materals
cOrder := 0
noMorePart := false
while( onhand > 0 and noMorePart = false) do

  forall(i in SO | SOItem(i) = partID(j) ) do
    if(readySeq(i) = 1 ) then
      case partSubMethod of
             1: do !ATC
          if(cOrder = 0) then
                          cOrder := i
                        elif( SIndex(i) > SIndex(cOrder) ) then
                          cOrder := i
                        end-if
        end-do

      2: do  !ATC1
          if(cOrder = 0) then
                          cOrder := i
                        elif( SIndex(i) > SIndex(cOrder) ) then
                          cOrder := i
                        end-if
        end-do

      3: do !EDD
          if(cOrder = 0) then
                          cOrder := i
                        elif( dueDate(i) < dueDate(cOrder) ) then
                          cOrder := i
                        end-if
        end-do

      4: do !EDDP
                 if(cOrder = 0) then
                   cOrder := i
                 elif( SOPriority(i) > SOPriority(cOrder) ) then
                   cOrder := i
                 elif( SOPriority(i) = SOPriority(cOrder) ) then
                   if( orderDueDate(parentCO1(i)) < orderDueDate(parentCO1(cOrder)) ) then
                                 cOrder := i
                   end-if
                 end-if
        end-do

      5: do  !ODDT
          if(cOrder = 0) then
                          cOrder := i
                        elif( SIndex(i) > SIndex(cOrder) ) then
                          cOrder := i
                        end-if
```

```
            end-do

          6: do  !WSPT
                    if(cOrder = 0) then
                                  cOrder := i
                                elif( SOPriority(i) > SOPriority(cOrder) ) then
                                  cOrder := i
                                end-if
            end-do



                         7: do  !TWK
              if(cOrder = 0) then
                              cOrder := i
                            elif( SIndex(i) < SIndex(cOrder) ) then
                              cOrder := i
                            end-if
            end-do

          8: do  !SOPN
              if(cOrder = 0) then
                              cOrder := i
                            elif( SIndex(i) < SIndex(cOrder) ) then
                              cOrder := i
                            end-if
            end-do

          9: do  !SPT
              if(cOrder = 0) then
                              cOrder := i
                            elif( OP(i) < OP(cOrder) ) then
                              cOrder := i
                            end-if
            end-do

          end-case
                       end-if
      end-do

      if( cOrder > 0) then

        onhand := onhand-1
        readySeq(cOrder) := 2
        !writeln("fixedAM(" + cOrder + "," + currentPO + "," + onhand)
        fixedAM1( cOrder, currentPO):= AM(cOrder, currentPO) = 1
sethidden(fixedAM1(cOrder, currentPO), false)

        cOrder := 0
      else
        noMorePart := true
      end-if
    end-do
```

```
    AllPODone := true
    forall(i in SO | SOItem(i) = partID(j) ) do
      if(readySeq(i) = 1 ) then
        AllPODone := false
      end-if
    end-do

  end-do



  setparam("XPRS_SCALING",0)
  setparam("XPRS_MAXMIPSOL",0)
  !setparam("XPRS_MAXTIME",10)

  minimize(PrimalObj)
  status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
  writeln('Solution after part-', j ,' is ', status(getprobstat), ",Objective: ",getobjval )
  !writeln("CPU time: " , gettime-starttime)

  !Record CV
  PartCV(j) := getobjval

  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) ) do
     sethidden(MaterialST3(i,s), true)
     sethidden(fixedAM1(i,s), true)
    end-do
  end-do

end-do

NumberOfSP := NumberOfSP + 1
TotalCUPTimeSP := TotalCUPTimeSP + (gettime - startTimeSP)

if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSP := TotalGapSP + 0
else
  TotalGapSP := TotalGapSP + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure
```

```
!***** machine-subproblem dispatch **********

procedure machineSubproblemDsp(currentMachineID:string)

declarations

status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
fixedAC1: array(SO,SO) of linctr
tempAC1: array(SO, SO) of real


end-declarations

startTimeSM := gettime

!Start from machines
forall(m in MC | MachineID(m) = currentMachineID) do

          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
       sethidden(CapacityST1(i,ii), false)
       sethidden(CapacityST3(i,ii), false)
       sethidden(fixedAC(i,ii), true)
     end-do
   end-do


   writeln("Machine:", m)
   UpdatePathFrom := false

   forall(i in SO)
            UpdatePathFrom1(i):= false

   forall(i in SO) PathFrom(i) := 0

   forall(i in SO) UpdatePathToSO(i) := false
   forall(i in SO, j in SO) PathToSO(i,j) := 0


   !calculate longest path for (0,i) and (i,k)

   cputime:= gettime


   progressJob := 0
   previousJob := 0
   simulationClock := 10000000

   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     simulationClock := minlist(getLongestPath(0,i), simulationClock)
   end-do

   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     readySeq(i) := 0
   end-do
```

```
AllOpsDone := false

while( AllOpsDone = false) do

NoReadyOrder := true
forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 1 ) do
  NoReadyOrder := false
end-do

if(NoReadyOrder = true) then
  minRelease := 100000
  forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 0) do
    minRelease := minlist(getLongestPath(0,i), minRelease)
  end-do
  simulationClock := maxlist(minRelease, simulationClock)
end-if



forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 0 ) do
  !writeln("readySeq(" + i + "): "+readySeq(i) + "," + getLongestPath(0,i) )

  if( getLongestPath(0,i) <= simulationClock ) then
    readySeq(i) := 1
  end-if

  forall(j in SO|requestedMachine(j) = MachineID(m) and i<>j) do
    if(getLongestPath1(j,i) > 0 and readySeq(j) < 2) then
      readySeq(i) := 0
    end-if
  end-do

end-do

forall(i in SO|requestedMachine(i) = MachineID(m) ) do
 forall(j in SO|parentCO(j) = parentCO1(i)) do
     SIndex(i) := 0!getLongestPath1(i,j)
   end-do
end-do



forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 1) do
  if(readySeq(i) = 1 ) then

    case machineSubMethod of
      1: do !ATC
          NOO := getInfoToCO(i)
          temp_exp := -1*maxlist( 0,orderDueDate(parentCO1(i)) - pathToCO(i) - simulationClock)
/5*17.5
          SIndex(i) :=  SOPriority(i) * exp( temp_exp )
        end-do

      2: do !ATC1
          NOO := getInfoToCO(i)
```

```
            temp_SIndex :=  simulationClock + pathToCO(i) - orderDueDate(parentCO1(i))
                          if( temp_SIndex > 0) then
                            SIndex(i) := SOPriority(i) /OP(i)
                          else
                            SIndex(i) := temp_SIndex
                          end-if
          end-do

        5: do  !ODDT
            NOO := getInfoToCO(i)
            temp_SIndex :=  simulationClock + pathToCO(i) - orderDueDate(parentCO1(i))
                          if( temp_SIndex > 0) then
                            SIndex(i) := temp_SIndex * SOPriority(i)
                          else
                            SIndex(i) := temp_SIndex
                          end-if
          end-do

        7: do  !TWK
            NOOS := getInfoSlack(i)
                  SIndex(i) := pathToCOSlack(i)
          end-do

        8: do  !SOPN
            NOOS := getInfoSlack(i)
                  SIndex(i) :=  (orderDueDate(parentCO1(i)) - pathToCOSlack(i) - simulationClock)/
numberOfOPSSlack(i)
          end-do


      end-case


    end-if
  end-do



  progressJob := 0

  forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 1) do
    !writeln("Job(" + i + "): OP:" + OP(i) )
   if(readySeq(i) = 1 ) then

      case machineSubMethod of
        1: do !ATC
            if(progressJob = 0) then
                            progressJob := i
                          elif( SIndex(i) > SIndex(progressJob) ) then
              progressJob := i
            end-if
          end-do

        2: do  !ATC1
            if(progressJob = 0) then
                            progressJob := i
```

```
                    elif( SIndex(i) > SIndex(progressJob) ) then
        progressJob := i
      end-if
    end-do

3: do !EDD
    if(progressJob = 0) then
                      progressJob := i
                    elif( dueDate(i) > dueDate(progressJob) ) then
        progressJob := i
      end-if
    end-do

4: do !EDDP
      if(progressJob = 0) then
                      progressJob := i
                    elif( SOPriority(i) > SOPriority(progressJob) ) then
                      progressJob := i
                    elif( SOPriority(i) = SOPriority(progressJob) ) then
                      if( orderDueDate(parentCO1(i)) < orderDueDate(parentCO1(progressJob)) ) then
                        progressJob := i
                       end-if
                    end-if
    end-do

5: do  !ODDT
      if(progressJob = 0) then
                      progressJob := i
                    elif( SIndex(i) > SIndex(progressJob) ) then
        progressJob := i
      end-if
    end-do

6: do  !WSPT
      if(progressJob = 0) then
                      progressJob := i
                    elif( SOPriority(i)/OP(i) > SOPriority(progressJob)/OP(progressJob) ) then
        progressJob := i
      end-if
    end-do



       7: do  !TWK
      if(progressJob = 0) then
                      progressJob := i
                    elif( SIndex(i) < SIndex(progressJob) ) then
        progressJob := i
      end-if
    end-do

8: do  !SOPN
      if(progressJob = 0) then
                      progressJob := i
                    elif( SIndex(i) < SIndex(progressJob) ) then
        progressJob := i
```

```
            end-if
          end-do

        9: do  !SPT
            if(progressJob = 0) then
                            progressJob := i
                          elif( OP(i) < OP(progressJob) ) then
              progressJob := i
            end-if
          end-do

        end-case

   end-if
 end-do
 !writeln("currentJob():" + progressJob)

 if( progressJob > 0) then
   if(  readySeq(progressJob) = 1 ) then
     simulationClock := simulationClock + OP(progressJob)

     if( previousJob > 0) then
       fixedAC1( progressJob, previousJob):= AC(progressJob, previousJob) = 1
       sethidden(fixedAC1(progressJob, previousJob), false)

     end-if

     previousJob := progressJob
     readySeq(progressJob) := 2
   end-if
 end-if


 AllOpsDone := true

forall(i in SO|requestedMachine(i) = MachineID(m) ) do

  if( readySeq(i) <= 1) then
    AllOpsDone := false
  end-if
end-do

 end-do


   setparam("XPRS_SCALING",0)
   setparam("XPRS_FEASTOL",1.0E-10)
          setparam("XPRS_MIPTOL",1.0E-9)
   setparam("XPRS_MAXMIPSOL",0)
   !setparam("XPRS_MAXTIME",10)

   !minimize(PrimalObj)
   status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
   !writeln('Solution after machine-', m ,' is ', status(getprobstat), ",Objective: ",getobjval )

   setparam("XPRS_SCALING",0)
```

```
!setparam("XPRS_HEURDEPTH",5)
setparam("XPRS_FEASTOL",1.0E-10)
        !setparam("XPRS_BIGMMETHOD",1)
        !setparam("XPRS_OPTIMALITYTOL",0)
        !setparam("XPRS_PRESOLVE",0)
        setparam("XPRS_MIPTOL",1.0E-9)
setparam("XPRS_MAXMIPSOL",0)
!setparam("XPRS_MAXTIME",10)

minimize(PrimalObj)
status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
writeln('Solution (Actual) after machine-', m ,' is ', status(getprobstat), ",Objective: ",getobjval )
!writeln("CPU time: " , gettime-starttime)
!writeln('best bound is ', getparam("XPRS_BESTBOUND"))
!writeln('Number of integer found is ', getparam("XPRS_MIPSOLS"))


!Record CV
MachineCV(m) := getobjval


        !Hidden all capacity constraints
        forall(mm in MC|MachineDone(mm) = true) do
                forall(i in SO|requestedMachine(i) = MachineID(mm) ) do
            forall( ii in SO |requestedMachine(ii) = MachineID(mm) and i <> ii) do
              sethidden(CapacityST1(i,ii), true)
              sethidden(CapacityST3(i,ii), true)
            end-do
          end-do
        end-do

        !Hidden precedence constaint
        forall(i in SO) do
          forall(ii in SO | parentSO(ii) = i) sethidden(PrecST2(i, ii), true)
        end-do

        forall(k in CO) do
          forall(i in SO | parentCO(i) = k) sethidden(PrecST3(k, i),true)
        end-do
!)

forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
    sethidden(CapacityST1(i,ii), true)
    sethidden(CapacityST3(i,ii), true)
    sethidden(fixedAC(i,ii), false)
  end-do
end-do


        forall(i in SO|requestedMachine(i) = MachineID(m) ) do
  forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
    sethidden(fixedAC1(i,ii), true)
    !writeln("AC(",i,",",ii,"):",strfmt(getsol(AC(i,ii)),13),",tempAC:",tempAC(i,ii))
  end-do
end-do
```

```
end-do

NumberOfSM := NumberOfSM + 1
TotalCUPTimeSM := TotalCUPTimeSM + (gettime - startTimeSM)
if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSM := TotalGapSM + 0
else
  TotalGapSM := TotalGapSM + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure
```

Appendix D. Source Code of Xpress-MP Model – ESBP

```
model MA
uses "mmxprs", "mmsystem"   !Xpress-Optimizer

parameters
  inputFileName = 'MA.dat'
  outputFileName = 'out.dat'
  maxCPUTime = 10
  maxCPUTimeSelection = 5
  maxCPUTimeReOpt = 10
  maxCPUTimeFullReOpt = 10
  rollingInterval = 20
  overlapInterval = 10

  initMachineSubDispatch = 1 !*** 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT
  initPartSubDispatch = 1    !*** 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT
  machineSubMethod = 0 !*** 0-Opt, 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT
  partSubMethod = 1    !*** 0-Opt, 1-ATC, 2-ATC2 , 3-EDD, 4-EDDP, 5-ODDT, 6-WSPT, 7-TWK, 8-
SOPN, 9-SPT
  rhp = 1  !*** 0-No RHP, 1-RHP

end-parameters


forward procedure SB

forward procedure partSubproblem(currentPartID:string)

forward procedure machineSubproblem(currentMachineID:string)

forward procedure partSubproblemOpt(currentPartID:string)

forward procedure machineSubproblemOpt(currentMachineID:string)

forward procedure partSubproblemDsp(currentPartID:string)

forward procedure machineSubproblemDsp(currentMachineID:string)

forward procedure simulation

forward procedure checkSolution


forward function getLongestPath(i, j: integer):integer

forward function getSOParnetCO(i: integer):integer

forward procedure push(i, j: integer)

forward procedure pull(i, j: integer)
```

```
forward procedure pushcycle(i, j, k: integer)

forward function getLongestPath1(i, j: integer):integer

forward procedure push1(i, j: integer, nodePush: boolean)

forward procedure printResult

forward function getInfoToCO(i: integer):integer

forward function getNumberOfOP(i, j: integer):integer

forward function getInfoSlack(i: integer):integer


forward procedure setTimeBucket

forward procedure solveSubproblem(bucketNumber:integer)


declarations
  CO:set of integer       !Customer Orders
  SO: set of integer      !Shop Order
  Part: set of integer    !Parts
  PO: set of integer      !Purchase Orders
  MC: set of integer        !Machines
end-declarations

initializations from inputFileName
  CO SO Part PO MC
end-initializations

finalize(CO)
finalize(SO)
finalize(Part)
finalize(PO)
finalize(MC)


declarations
  !Customer Orders
  orderNumber: array(CO) of integer
  orderRequestedItem: array(CO) of string
  orderQuantity: array(CO) of integer
  orderDueDate: array(CO) of integer
  orderPET: array(CO) of mpvar
  orderPriority: array(CO) of integer
  orderTardiness: array(CO) of mpvar

  !Shop Order
  SONumber: array(SO) of string
  parentCO: array(SO) of integer
  parentSO: array(SO) of integer
  SOItem: array(SO) of string
  SOQuantity: array(SO) of integer
  SOPriority: array(SO) of integer
```

```
requestedMachine: array(SO) of string
OP: array(SO) of integer
PST: array(SO) of mpvar    !Desicion variables
PET: array(SO) of mpvar       !Desicion variables
LL: array(SO) of integer
parentCO1: array(SO) of integer
numberOfIncomingArc: array(SO) of integer
numberOfToken: array(SO) of integer
dueDate: array(SO) of integer
TimeBucket: array(SO) of integer

!Parts
partNumber: array(Part) of integer
partID: array(Part) of string
RWType: array(Part) of integer
ASType: array(Part) of integer
EDType: array(Part) of integer
PartCV: array(Part) of real
PartDone: array(Part) of boolean

!Purchase Orders
PONumber: array(PO) of integer
POItem: array(PO) of string
POTime: array(PO) of integer
POQuantity: array(PO) of integer

!Machines
MachineNumber: array(MC) of integer
MachineID: array(MC) of string
MachineCV: array(MC) of real
MachineDone: array(MC) of boolean
MachineEST: array(MC) of real


TM: array(SO, PO) of mpvar    !Desicion variables
AM: array(SO, PO) of mpvar    !Desicion variables
AC: array(SO, SO) of mpvar    !Desicion variables

BigM: real
starttime: real
maxCV: real
allDone: boolean


MaterialST1: array(PO) of linctr
MaterialST2: array(SO) of linctr
MaterialST3: array(SO, PO) of linctr
MaterialST4: array(SO, PO) of linctr
MaterialST5: array(SO, PO) of linctr
CapacityST1: array(SO, SO) of linctr
CapacityST2: array(SO, SO) of linctr
CapacityST3: array(SO, SO) of linctr
CapacityST4: array(SO, SO) of linctr
CapacityST5: array(SO) of linctr

PrecST1: array(SO) of linctr
```

```
PrecST2: array(SO, SO) of linctr
PrecST3: array(CO, SO) of linctr
PrecST4: array(SO) of linctr
PrecST5: array(SO, CO) of linctr
Tard1: array(CO) of linctr
Tard2: array(CO) of linctr
fixedAC: array(SO,SO) of linctr
fixedAM: array(SO,PO) of linctr
tempAC: array(SO, SO) of real
tempAM: array(SO, SO) of real
DelayPrecST: array(SO,SO) of linctr

UE: array(SO) of real
UE_st: array(SO) of real
UE_temp: array(SO) of real
UE1: array(SO, CO) of real
UE1_st: array(SO, CO) of real
UE1_temp: array(SO, CO) of real

tk: real
iteration: real
BestObjPrime: real

PathFrom: array(SO) of integer
Flow: array(SO) of integer
UpdatePathFrom: boolean
UpdatePathFrom1: array(SO) of boolean
counter: integer
sumAC: real
sumAC1: real

PathToSO: array(SO, SO) of integer
UpdatePathToSO: array(SO) of boolean
nodePush: array(SO) of boolean

 NumberOfSM: real
TotalCUPTimeSM: real
TotalGapSM: real
NumberOfSP: real
TotalCUPTimeSP: real
TotalGapSP: real

NumberOfSR: real
TotalCUPTimeSR: real
TotalGapSR: real

initialSolDispatch: real
initialSolRHP: real
initialSolSB: real

TotalROWS: real
TotalCOLS: real
TotalELEMS: real
TotalMIPENTS: real

readySeq: array(SO) of integer
```

```
  POQuantityS: array(PO) of integer


  !Simulation
  simulatorClock: integer
  nextEventTime: integer
  allOrderDone: boolean
  noMorePart: boolean

  simOrderPET: array(CO) of integer
  simOrderTardiness: array(CO) of integer

  readyOrder: array(SO) of integer
  numberOfChild: array(SO) of integer
  simPST: array(SO) of integer
  simPET: array(SO) of integer

  nextArrivalTime: array(Part) of integer
  onhand: array(Part) of integer
  currentOrder: array(Part) of integer

  nextAvailableTime: array(MC) of integer
  currentJob: array(MC) of integer

  SIndex: array(SO) of real

  numberOfOPS: array(SO) of integer
  pathToCO: array(SO) of integer

  numberOfOPSSlack: array(SO) of integer
  pathToCOSlack: array(SO) of integer

  currentBucket: integer
  maxBucket: integer
  inputBucket: integer
  fixedPST: array(SO) of linctr

end-declarations

initializations from inputFileName
  ! CO (Customer Orders) data
  orderNumber orderRequestedItem orderQuantity orderDueDate orderPriority

  ! SO (Shop Order)
  SONumber parentCO parentSO SOItem SOQuantity SOPriority requestedMachine OP parentCO1

  !Part (Parts)
  partNumber partID RWType ASType EDType

  !PO (Purchase Orders)
  PONumber POItem POTime POQuantity

  !MC (Machines)
  MachineNumber MachineID
end-initializations
```

```
!-----------------------------------------------------------------

starttime:= gettime
BigM := 10000
!Objective
PrimalObj:= sum(k in CO) orderPriority(k)*orderQuantity(k)*orderTardiness(k)

forall(k in CO) Tard1(k) := orderTardiness(k) >= orderPET(k)-orderDueDate(k)
forall(k in CO) Tard2(k) := orderTardiness(k) >= 0


!--------------Material constraints

!Limits on PO Quantity
forall(j in Part | RWType(j) = 1) do
  forall(s in PO | POItem(s) = partID(j) )
    MaterialST1(s):= sum(i in SO | SOItem(i) =  partID(j) ) TM(i,s) <= POQuantity(s)
end-do

!Satisfy SO Requirements
forall(j in Part | RWType(j) = 1) do
  forall(i in SO | SOItem(i) = partID(j) )
    MaterialST2(i):= sum(s in PO |POItem(s) = partID(j) ) TM(i,s) = SOQuantity(i)
end-do


forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST3(i,s):=  PST(i) >= POTime(s)*AM(i,s)
  end-do
end-do

! if TW(s,i) > 0, AW = 1, else, AW = 0

forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST4(i,s):= TM(i,s) >= AM(i,s)
  end-do
end-do


forall(j in Part | RWType(j) = 1) do
  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) )
      MaterialST5(i,s):= TM(i,s) <= 1000* AM(i,s)
  end-do
end-do



!-----------Capacity constraints
!if PST(i) >= PET(j), then AC(i,j) =1 , else, AC(i,j) = 0
```

```
forall(m in MC) do
            forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
        CapacityST1(i,ii):= PET(i) <= PST(ii) + BigM*AC(i, ii)
   end-do
end-do

(!
forall(m in MC) do
   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
        CapacityST2(i,ii):= PET(ii) <= PST(i) + BigM*(1-AC(i, ii))
   end-do
end-do
!)

forall(m in MC) do
            forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
        CapacityST3(i,ii):= AC(i,ii) + AC(ii,i) = 1
   end-do
end-do


!------------Precedence constraints
forall(i in SO) PrecST1(i):= PET(i) = PST(i) + OP(i)

forall(i in SO) do
  forall(ii in SO | parentSO(ii) = i) PrecST2(i, ii):= PST(i) >= PET(ii)
end-do

forall(k in CO) do
   forall(i in SO | parentCO(i) = k) PrecST3(k, i):= orderPET(k) = PET(i)
end-do

!forall(i in SO) PST(i) <= 8000000
forall(i in SO) PST(i) <= 10000
forall(i in SO) PST(i) is_integer
forall(i in SO) PET(i) is_integer
forall(i in SO, s in PO) AM(i,s) is_binary
forall(i in SO, s in PO) TM(i,s) is_integer
forall(i in SO, ii in SO) AC(i,ii) is_binary


!********* perform main program

simulation

initializations from inputFileName
  POQuantity
end-initializations

SB

checkSolution
```

```
printResult

!********* end of main program


procedure SB

declarations

status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string

end-declarations

starttime := gettime


case rhp of

1: do

        setTimeBucket

        inputBucket := 1

        while( inputBucket <= maxBucket) do
           writeln("---------Current Bucket: ", inputBucket,' -------')
                solveSubproblem(inputBucket)
                inputBucket := inputBucket + 1
        end-do

        forall(m in MC) do
                forall(i in SO|requestedMachine(i) = MachineID(m)  ) do
             forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
               sethidden(CapacityST1(i,ii), false)
           end-do
        end-do


        forall(m in MC) do
                forall(i in SO|requestedMachine(i) = MachineID(m) ) do
             forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii)
               sethidden(CapacityST3(i,ii), false)
           end-do
        end-do


        forall(m in MC) do
          forall(i in SO|requestedMachine(i) = MachineID(m) ) do
            sethidden(CapacityST5(i), true)
          end-do
        end-do


        checkSolution
```

```
            !printResult


            writeln('Check-Solution is ', getprobstat, ",Objective: ",getobjval )

            initialSolRHP := getobjval

            if(initialSolDispatch < initialSolRHP or getprobstat = 6) then
              writeln("---------use dispatch rule solution: ", initialSolDispatch ,' -------')
              !simulation
              forall(i in SO) do
                sethidden(fixedPST(i), false)
                fixedPST(i) :=  PST(i)= simPST(i)
                setparam("XPRS_SCALING",0)
                setparam("XPRS_MAXMIPSOL",0)
                setparam("XPRS_MAXTIME",maxCPUTime)
                minimize(PrimalObj)
              end-do
            end-if

    end-do

0: do
     forall(i in SO) do
       fixedPST(i) := PST(i) = getsol(simPST(i))
     end-do
   end-do

end-case



setparam("XPRS_SCALING",0)
setparam("XPRS_FEASTOL",1.0E-10)
setparam("XPRS_MIPTOL",1.0E-9)
setparam("XPRS_MAXMIPSOL",0)
setparam("XPRS_MAXTIME",maxCPUTimeFullReOpt)

minimize(PrimalObj)

objective := getobjval

forall(i in SO) writeln("SO--PST(", i,"):", getsol(PST(i)) )
status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
writeln('Initial Solution is ', status(getprobstat), ",Objective: ",getobjval )
writeln("CPU time: " , gettime-starttime)


!fixed TM(i,s)
forall(j in Part | RWType(j) = 1) do
   forall(i in SO| SOItem(i) = partID(j)) do
     forall(s in PO | POItem(s) = partID(j) ) do
              tempAM(i,s):= getsol(AM(i,s))
              fixedAM(i,s):= AM(i,s) = tempAM(i,s)
            end-do
   end-do
```

```
end-do

!fixed AC(i,ii)
forall(m in MC) do
   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
         tempAC(i,ii):= getsol(AC(i,ii))
         fixedAC(i,ii):= AC(i,ii) = tempAC(i,ii)
      end-do
   end-do
end-do

forall(i in SO) do
   sethidden(fixedPST(i), true)
end-do




objective := getobjval

count:= 1
maxCount := 3
currentObj := getobjval
bestObj := getobjval + 1

while( count <= maxCount and currentObj < bestObj) do
  bestObj := getobjval

  writeln("CurrentObj(Full-Opt):", getobjval, ";Count:", count )

  forall(j in Part | RWType(j) = 1 ) do
     forall(i in SO| SOItem(i) = partID(j)) do
        forall(s in PO | POItem(s) = partID(j) ) do
           !sethidden(MaterialST3(i,s), true)
           sethidden(fixedAM(i,s), true)
        end-do
     end-do

     partSubproblem( partID(j) )


     !fixed TM(i,s)
     if(getobjval < objective) then
                        forall(i in SO| SOItem(i) = partID(j)) do
                           forall(s in PO | POItem(s) = partID(j) ) do
                              tempAM(i,s):= getsol(AM(i,s))
                              fixedAM(i,s):= AM(i,s) = tempAM(i,s)
                           end-do
                        end-do
                        objective := getobjval
                        writeln("BestObj:", getobjval)
```

```
            end-if

            forall(i in SO| SOItem(i) = partID(j)) do
              forall(s in PO | POItem(s) = partID(j) ) do
                 sethidden(fixedAM(i,s), false)
               end-do
            end-do

  end-do



  !objective := getobjval
  writeln("CurrentObj:", getobjval)

  !reoptimization
  forall(m in MC) do
    forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
        !sethidden(CapacityST1(i,ii), true)
        sethidden(fixedAC(i,ii), true)
      end-do
    end-do

    machineSubproblem( MachineID(m) )

    if(getobjval < objective) then
                      !fixed AC(i,ii)
              forall(i in SO|requestedMachine(i) = MachineID(m) ) do
                forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
                  tempAC(i,ii):= getsol(AC(i,ii))
                  fixedAC(i,ii):= AC(i,ii) = tempAC(i,ii)
                end-do
              end-do
      objective := getobjval
              writeln("BestObj:", getobjval)
           end-if

           forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
        sethidden(fixedAC(i,ii), false)
      end-do
    end-do


  end-do

  count:= count +1
  currentObj := getobjval
end-do


writeln("CPU time: " , gettime-starttime)


forall(k in CO) writeln("CO--orderTardiness(", k,"):", getsol(orderTardiness(k)))
```

```
forall(k in CO) writeln("CO--PET(", k,"):", getsol(orderPET(k)))
forall(k in CO) writeln("CO--DueDate(", k,"):", getsol(orderDueDate(k)))
forall(k in CO) writeln("CO--PET(", k,"):", getsol(orderPET(k)))
!forall(i in SO) writeln("SO--PST(", i,"):", getsol(PST(i)), ";PET(",i,"):", getsol(PET(i)), ";OP(",i,"):",
getsol(OP(i)) )

end-procedure


procedure partSubproblem(currentPartID:string)

(!
  machineSubMethod = 1 !*** 1-Opt, 2-Dispatch
  partSubMethod = 1    !*** 1-Opt, 2-Dispatch
 !)

case partSubMethod of

  0: partSubproblemOpt(currentPartID)

  1..9 : partSubproblemDsp(currentPartID)

end-case


end-procedure



procedure machineSubproblem(currentMachineID:string)

case machineSubMethod of

  0: machineSubproblemOpt(currentMachineID)

  1..9: machineSubproblemDsp(currentMachineID)

end-case


end-procedure



procedure partSubproblemOpt(currentPartID:string)

declarations
status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string

end-declarations

startTimeSP := gettime

!Start from raw materials
forall(j in Part | RWType(j) = 1 and partID(j) = currentPartID) do
```

```
  setparam("XPRS_SCALING",0)
  setparam("XPRS_MAXMIPSOL",0)
  !setparam("XPRS_MAXTIME",maxCPUTimeFullReOpt)

  minimize(PrimalObj)
  status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
  writeln('Solution after part-', j ,' is ', status(getprobstat), ",Objective: ",getobjval )
  !writeln("CPU time: " , gettime-starttime)

  !Record CV
  PartCV(j) := getobjval

end-do

NumberOfSP := NumberOfSP + 1
TotalCUPTimeSP := TotalCUPTimeSP + (gettime - startTimeSP)
if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSP := TotalGapSP + 0
else
  TotalGapSP := TotalGapSP + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")


end-procedure



procedure machineSubproblemOpt(currentMachineID:string)

declarations
status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string

end-declarations

startTimeSM := gettime

!Start from machines
forall(m in MC | MachineID(m) = currentMachineID) do

   setparam("XPRS_SCALING",0)
   !setparam("XPRS_HEURDEPTH",5)
   setparam("XPRS_FEASTOL",1.0E-10)
         !setparam("XPRS_BIGMMETHOD",1)
         !setparam("XPRS_OPTIMALITYTOL",0)
         !setparam("XPRS_PRESOLVE",0)
         setparam("XPRS_MIPTOL",1.0E-9)
   setparam("XPRS_MAXMIPSOL",0)
   !setparam("XPRS_MAXTIME",maxCPUTimeFullReOpt)
   minimize(PrimalObj)
   status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
```

```
    writeln('Solution after machine-', m ,' is ', status(getprobstat), ",Objective: ",getobjval )
    !writeln("CPU time: " , gettime-starttime)
    !writeln('best bound is ', getparam("XPRS_BESTBOUND"))
    !writeln('Number of integer found is ', getparam("XPRS_MIPSOLS"))

    !Record CV
    MachineCV(m) := getobjval


end-do

NumberOfSM := NumberOfSM + 1
TotalCUPTimeSM := TotalCUPTimeSM + (gettime - startTimeSM)
if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSM := TotalGapSM + 0
else
  TotalGapSM := TotalGapSM + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure


procedure partSubproblemDsp(currentPartID:string)

declarations

          status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
          fixedAM1: array(SO,PO) of linctr
          tempAM1: array(SO,PO) of real

end-declarations

startTimeSP := gettime

!Start from raw materials
forall(j in Part | RWType(j) = 1 and partID(j) = currentPartID) do

  cOrder := 0
  simulatorClock := 100000
  onhand1 := 0
  currentPO := 0


  forall(s in PO | POItem(s) = partID(j) ) do
    POQuantityS(s) := POQuantity(s)
  end-do

  forall(i in SO | SOItem(i) = partID(j) ) do
    readySeq(i) := 1
  end-do
```

```
AllPODone := false

      while( AllPODone = false) do
        simulatorClock := 100000

        forall(s in PO | POItem(s) = partID(j) ) do
          if( POQuantityS(s) > 0 ) then
            simulatorClock := minlist( simulatorClock, POTime(s) )
          end-if
        end-do

        forall(s in PO | POItem(s) = partID(j) ) do

     if( POQuantityS(s) > 0 and POTime(s) <= simulatorClock ) then
            onhand1 := onhand1 + POQuantityS(s)
            POQuantityS(s) := 0
            currentPO := s
          end-if

        end-do

        !writeln("nextArrivaltime:" + simulatorClock + ",s:" + currentPO)
    forall(i in SO | SOItem(i) = partID(j) ) do
              forall(jj in SO| parentCO(jj) = parentCO1(i) ) do
                SIndex(i) := 0!getLongestPath1(i,jj)
              end-do
             end-do



             forall(i in SO | SOItem(i) = partID(j) and readySeq(i) = 1) do
               if(readySeq(i) = 1 ) then

                 case partSubMethod of
                   1: do !ATC
                        NOO := getInfoToCO(i)
                 temp_exp := -1*maxlist( 0,orderDueDate(parentCO1(i)) - pathToCO(i) -
simulatorClock) /5*17.5
                 SIndex(i) :=  SOPriority(i) * exp( temp_exp )
               end-do

             2: do !ATC1
                 NOO := getInfoToCO(i)
                 temp_SIndex :=  simulatorClock + pathToCO(i) - orderDueDate(parentCO1(i))
                                 if( temp_SIndex > 0) then
                                   SIndex(i) := temp_SIndex * SOPriority(i)
                                 else
                                   SIndex(i) := temp_SIndex
                                 end-if
               end-do

             5: do  !ODDT
                 NOO := getInfoToCO(i)
                 temp_SIndex :=  simulatorClock + pathToCO(i) - orderDueDate(parentCO1(i))
                                 if( temp_SIndex > 0) then
```

```
                                        SIndex(i) := temp_SIndex * SOPriority(i)
                                      else
                                        SIndex(i) := temp_SIndex
                                      end-if
                end-do

               7: do  !TWK
                    NOOS := getInfoSlack(i)
                      SIndex(i) := pathToCOSlack(i)
                end-do

               8: do  !SOPN
                    NOOS := getInfoSlack(i)
                      SIndex(i) :=  (orderDueDate(parentCO1(i)) - pathToCOSlack(i) - simulatorClock)/
numberOfOPSSlack(i)
                end-do


             end-case

                  end-if
               end-do




          !assign raw materals
          cOrder := 0
          noMorePart := false
          while( onhand1 > 0 and noMorePart = false) do

            forall(i in SO|SOItem(i) = partID(j) and readySeq(i) = 1) do
                            !writeln("Job(" + i + "): OP:" + OP(i) )
                           if(readySeq(i) = 1 ) then

                             case partSubMethod of
                                1: do !ATC
                             if(cOrder = 0) then
                                         cOrder := i
                                       elif( SIndex(i) > SIndex(cOrder) ) then
                                         cOrder := i
                                       end-if
                           end-do

                         2: do  !ATC1
                             if(cOrder = 0) then
                                         cOrder := i
                                       elif( SIndex(i) > SIndex(cOrder) ) then
                                         cOrder := i
                                       end-if
                           end-do

                         3: do !EDD
                             if(cOrder = 0) then
                                         cOrder := i
                                       elif( dueDate(i) < dueDate(cOrder) ) then
                                         cOrder := i
```

```
                end-if
        end-do

      4: do !EDDP
            if(cOrder = 0) then
              cOrder := i
            elif( SOPriority(i) > SOPriority(cOrder) ) then
              cOrder := i
            elif( SOPriority(i) = SOPriority(cOrder) ) then
              if( orderDueDate(parentCO1(i)) <
orderDueDate(parentCO1(cOrder)) ) then
                        cOrder := i
              end-if
            end-if
        end-do

      5: do  !ODDT
          if(cOrder = 0) then
                  cOrder := i
                elif( SIndex(i) > SIndex(cOrder) ) then
                  cOrder := i
                end-if
        end-do

      6: do  !WSPT
            if(cOrder = 0) then
                      cOrder := i
                    elif( SOPriority(i) > SOPriority(cOrder) ) then
                      cOrder := i
                    end-if
        end-do



              7: do  !TWK
          if(cOrder = 0) then
                  cOrder := i
                elif( SIndex(i) < SIndex(cOrder) ) then
                  cOrder := i
                end-if
        end-do

      8: do  !SOPN
          if(cOrder = 0) then
                  cOrder := i
                elif( SIndex(i) < SIndex(cOrder) ) then
                  cOrder := i
                end-if
        end-do

      9: do  !SPT
          if(cOrder = 0) then
                  cOrder := i
                elif( OP(i) < OP(cOrder) ) then
                  cOrder := i
                end-if
```

```
                    end-do

              end-case

                    end-if
                   end-do

            if( cOrder > 0) then
              !PST(currentOrder(j)) := simulatorClock
              !PET(currentOrder(j)) := simulatorClock
              onhand1 := onhand1-1
              readySeq(cOrder) := 2
              !writeln("fixedAM(" + cOrder + "," + currentPO + "," + onhand)
              fixedAM1( cOrder, currentPO):= AM(cOrder, currentPO) = 1
        sethidden(fixedAM1(cOrder, currentPO), false)


              cOrder := 0
            else
              noMorePart := true
            end-if
          end-do



    AllPODone := true
    forall(i in SO | SOItem(i) = partID(j) ) do
      if(readySeq(i) = 1 ) then
        AllPODone := false
      end-if
    end-do

  end-do


  setparam("XPRS_SCALING",0)
  setparam("XPRS_MAXMIPSOL",0)
  !setparam("XPRS_MAXTIME",10)

  minimize(PrimalObj)
  status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
  writeln('Solution after part-', j ,' is ', status(getprobstat), ",Objective: ",getobjval )
  !writeln("CPU time: " , gettime-starttime)

  !Record CV
  PartCV(j) := getobjval


  forall(i in SO| SOItem(i) = partID(j)) do
    forall(s in PO | POItem(s) = partID(j) ) do
     !sethidden(MaterialST3(i,s), true)
     sethidden(fixedAM1(i,s), true)
    end-do
  end-do

end-do
```

```
NumberOfSP := NumberOfSP + 1
TotalCUPTimeSP := TotalCUPTimeSP + (gettime - startTimeSP)

if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSP := TotalGapSP + 0
else
  TotalGapSP := TotalGapSP + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure




procedure machineSubproblemDsp(currentMachineID:string)

declarations

status: array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})of string
fixedAC1: array(SO,SO) of linctr
tempAC1: array(SO, SO) of real


end-declarations

startTimeSM := gettime

!Start from machines
forall(m in MC | MachineID(m) = currentMachineID) do


   progressJob := 0
   previousJob := 0
   simulationClock := 10000000

   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     simulationClock := minlist(getLongestPath(0,i), simulationClock)
   end-do

   forall(i in SO|requestedMachine(i) = MachineID(m) ) do
     readySeq(i) := 0
   end-do


   AllOpsDone := false

   while( AllOpsDone = false) do

   NoReadyOrder := true
```

```
  forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 1 ) do
    NoReadyOrder := false
  end-do

  if(NoReadyOrder = true) then
    minRelease := 100000
    forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 0) do
      minRelease := minlist(getLongestPath(0,i), minRelease)
    end-do
    simulationClock := maxlist(minRelease, simulationClock)
  end-if



  forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 0 ) do
    !writeln("readySeq(" + i + "): "+readySeq(i) + "," + getLongestPath(0,i) )

    if( getLongestPath(0,i) <= simulationClock ) then
      readySeq(i) := 1
    end-if

    forall(j in SO|requestedMachine(j) = MachineID(m) and i<>j) do
      if(getLongestPath1(j,i) > 0 and readySeq(j) < 2) then
        readySeq(i) := 0
      end-if
    end-do
  end-do



 forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 1) do
   if(readySeq(i) = 1 ) then

     case machineSubMethod of
       1: do !ATC
           NOO := getInfoToCO(i)
           temp_exp := -1*maxlist( 0,orderDueDate(parentCO1(i)) - pathToCO(i) - simulationClock)
/5*17.5
           SIndex(i) :=  SOPriority(i) * exp( temp_exp )
         end-do

       2: do !ATC1
           NOO := getInfoToCO(i)
           temp_SIndex :=  simulationClock + pathToCO(i) - orderDueDate(parentCO1(i))
                         if( temp_SIndex > 0) then
                           SIndex(i) := SOPriority(i) /OP(i)
                         else
                           SIndex(i) := temp_SIndex
                         end-if
         end-do

       5: do  !ODDT
           NOO := getInfoToCO(i)
           temp_SIndex :=  simulationClock + pathToCO(i) - orderDueDate(parentCO1(i))
                         if( temp_SIndex > 0) then
                           SIndex(i) := temp_SIndex * SOPriority(i)
```

```
                    else
                      SIndex(i) := temp_SIndex
                    end-if
          end-do

        7: do  !TWK
            NOOS := getInfoSlack(i)
                 SIndex(i) := pathToCOSlack(i)
          end-do

        8: do  !SOPN
            NOOS := getInfoSlack(i)
                 SIndex(i) :=  (orderDueDate(parentCO1(i)) - pathToCOSlack(i) - simulationClock)/
numberOfOPSSlack(i)
          end-do


      end-case

    end-if
  end-do


  progressJob := 0

  forall(i in SO|requestedMachine(i) = MachineID(m) and readySeq(i) = 1) do
    !writeln("Job(" + i + "): OP:" + OP(i) )
   if(readySeq(i) = 1 ) then
     if(progressJob = 0) then
       progressJob := i
     elif( SIndex(i) > SIndex(progressJob) ) then
       progressJob := i

     end-if

     case machineSubMethod of
        1: do !ATC
            if(progressJob = 0) then
                      progressJob := i
                    elif( SIndex(i) > SIndex(progressJob) ) then
              progressJob := i
            end-if
          end-do

        2: do  !ATC1
            if(progressJob = 0) then
                      progressJob := i
                    elif( SIndex(i) > SIndex(progressJob) ) then
              progressJob := i
            end-if
          end-do

        3: do !EDD
           if(progressJob = 0) then
                      progressJob := i
                    elif( dueDate(i) > dueDate(progressJob) ) then
```

```
      progressJob := i
    end-if
  end-do

4: do !EDDP
    if(progressJob = 0) then
                    progressJob := i
                  elif( SOPriority(i) > SOPriority(progressJob) ) then
                    progressJob := i
                  elif( SOPriority(i) = SOPriority(progressJob) ) then
                    if( orderDueDate(parentCO1(i)) < orderDueDate(parentCO1(progressJob)) ) then
                      progressJob := i
                     end-if
                  end-if
  end-do

5: do  !ODDT
    if(progressJob = 0) then
                    progressJob := i
                  elif( SIndex(i) > SIndex(progressJob) ) then
      progressJob := i
    end-if
  end-do

6: do  !WSPT
    if(progressJob = 0) then
                    progressJob := i
                  elif( SOPriority(i)/OP(i) > SOPriority(progressJob)/OP(progressJob) ) then
      progressJob := i
    end-if
  end-do



       7: do  !TWK
    if(progressJob = 0) then
                    progressJob := i
                  elif( SIndex(i) < SIndex(progressJob) ) then
      progressJob := i
    end-if
  end-do

8: do  !SOPN
    if(progressJob = 0) then
                    progressJob := i
                  elif( SIndex(i) < SIndex(progressJob) ) then
      progressJob := i
    end-if
  end-do

9: do  !SPT
    if(progressJob = 0) then
                    progressJob := i
                  elif( OP(i) < OP(progressJob) ) then
      progressJob := i
    end-if
```

```
            end-do

      end-case

    end-if
  end-do
  !writeln("currentJob():" + progressJob)

  if( progressJob > 0) then
    if(  readySeq(progressJob) = 1 ) then
      !simPST(currentJob(m)) := simulatorClock
      !simPET(currentJob(m)) := simPST(currentJob(m)) + OP(currentJob(m))
      !tempPET := simPET(currentJob(m))
      simulationClock := simulationClock + OP(progressJob)

      if( previousJob > 0) then
        fixedAC1( progressJob, previousJob):= AC(progressJob, previousJob) = 1
        sethidden(fixedAC1(progressJob, previousJob), false)
        !writeln("fixedAC:", progressJob, ",", previousJob)
      end-if

      previousJob := progressJob
      readySeq(progressJob) := 2
    end-if
  end-if


  AllOpsDone := true

 forall(i in SO|requestedMachine(i) = MachineID(m) ) do

   if( readySeq(i) <= 1) then
     AllOpsDone := false
   end-if
 end-do

  end-do

    setparam("XPRS_SCALING",0)
    !setparam("XPRS_HEURDEPTH",5)
    setparam("XPRS_FEASTOL",1.0E-10)
           !setparam("XPRS_BIGMMETHOD",1)
           !setparam("XPRS_OPTIMALITYTOL",0)
           !setparam("XPRS_PRESOLVE",0)
           setparam("XPRS_MIPTOL",1.0E-9)
    setparam("XPRS_MAXMIPSOL",0)
    !setparam("XPRS_MAXTIME",10)

    minimize(PrimalObj)
    status:= ['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
    writeln('Solution after machine-', m ,' is ', status(getprobstat), ",Objective: ",getobjval )
    !writeln("CPU time: " , gettime-starttime)
    !writeln('best bound is ', getparam("XPRS_BESTBOUND"))
    !writeln('Number of integer found is ', getparam("XPRS_MIPSOLS"))
```

```
    writeln("cycle-CPU time: " , gettime-starttime)

            forall(i in SO|requestedMachine(i) = MachineID(m) ) do
      forall( ii in SO |requestedMachine(ii) = MachineID(m) and i <> ii) do
        sethidden(fixedAC1(i,ii), true)
        !writeln("AC(",i,",",ii,"):",strfmt(getsol(AC(i,ii)),13),",tempAC:",tempAC(i,ii))
      end-do
    end-do


end-do

NumberOfSM := NumberOfSM + 1
TotalCUPTimeSM := TotalCUPTimeSM + (gettime - startTimeSM)
if(getparam("XPRS_BESTBOUND")= 0) then
  TotalGapSM := TotalGapSM + 0
else
  TotalGapSM := TotalGapSM + (getparam("XPRS_MIPOBJVAL")-
getparam("XPRS_BESTBOUND"))/getparam("XPRS_BESTBOUND")
end-if

TotalROWS := TotalROWS + getparam("XPRS_ROWS")
TotalCOLS := TotalCOLS + getparam("XPRS_COLS")
TotalELEMS := TotalELEMS + getparam("XPRS_ELEMS")
TotalMIPENTS := TotalMIPENTS + getparam("XPRS_MIPENTS")

end-procedure




procedure simulation

            starttime:= gettime

            simulatorClock := 0
            allOrderDone := false

            forall(i in SO) do
              numberOfchild(i) := 0
            end-do

            forall(i in SO) do
              if(OP(i) > 0) then
                 readyOrder(i) := 0
              else
                 readyOrder(i) := 1
              end-if

              forall(ii in SO) do
                if(parentSO(ii) = i) then
                  numberOfchild(i) := numberOfchild(i) + 1
                end-if
              end-do

               dueDate(i) := orderDueDate(parentCO1(i))
```

```
end-do

forall(j in Part | RWType(j) = 1) do
  currentOrder(j) := 0
  nextArrivalTime(j) := 0
end-do

forall(m in MC) do
  currentJob(m) := 0
  nextAvailableTime(m) := 0
end-do


repeat
   writeln("simulator clock: " + simulatorClock)

  !update part status
  forall(j in Part | RWType(j) = 1) do
    if(simulatorClock >= nextArrivalTime(j) ) then

          nextArrivalTime(j) := 1000000000
          forall(s in PO | POItem(s) = partID(j) ) do
            if( POQuantity(s) > 0 and POTime(s) <= simulatorClock ) then
              onhand(j) := onhand(j) + POQuantity(s)
              POQuantity(s) := 0
            end-if

            if( POQuantity(s) > 0 and POTime(s) >= simulatorClock ) then
              nextArrivalTime(j) := minlist( nextArrivalTime(j), POTime(s) )
            end-if
          end-do
          if( nextArrivalTime(j) = 1000000000) then
            nextArrivalTime(j) := simulatorClock
          end-if

          !assign raw materals
          currentOrder(j) := 0
          noMorePart := false
          while( onhand(j) > 0 and noMorePart = false) do

            forall(i in SO | SOItem(i) = partID(j) and readyOrder(i) = 1) do
                   case initPartSubDispatch of
                     1: do !ATC
                         NOO := getInfoToCO(i)
                temp_exp := -1*maxlist( 0,orderDueDate(parentCO1(i)) - pathToCO(i) -
simulatorClock) /5*17.5
                SIndex(i) := SOPriority(i) * exp( temp_exp )
              end-do

            2: do !ATC1
                NOO := getInfoToCO(i)
                temp_SIndex := simulatorClock + pathToCO(i) -
orderDueDate(parentCO1(i))
                          if( temp_SIndex > 0) then
                            SIndex(i) := temp_SIndex * SOPriority(i)
                          else
```

```
                                        SIndex(i) := temp_SIndex
                                      end-if
                         end-do

                       5: do  !ODDT
                           NOO := getInfoToCO(i)
                           temp_SIndex :=  simulatorClock + pathToCO(i) -
orderDueDate(parentCO1(i))
                                      if( temp_SIndex > 0) then
                                        SIndex(i) := temp_SIndex * SOPriority(i)
                                      else
                                        SIndex(i) := temp_SIndex
                                      end-if
                         end-do

                       7: do  !TWK
                           NOOS := getInfoSlack(i)
                             SIndex(i) := pathToCOSlack(i)
                         end-do

                       8: do  !SOPN
                           NOOS := getInfoSlack(i)
                             SIndex(i) :=  (orderDueDate(parentCO1(i)) - pathToCOSlack(i) -
simulatorClock)/ numberOfOPSSlack(i)
                         end-do


                     end-case
                          end-do


                   forall(i in SO | SOItem(i) = partID(j) ) do
                     if( readyOrder(i) = 1 ) then

                       case initPartSubDispatch of
                                1: do !ATC
                             if(currentOrder(j) = 0) then
                                         currentOrder(j) := i
                                       elif( SIndex(i) > SIndex(currentOrder(j)) ) then
                                         currentOrder(j) := i
                                       end-if
                           end-do

                         2: do  !ATC1
                             if(currentOrder(j) = 0) then
                                         currentOrder(j) := i
                                       elif( SIndex(i) > SIndex(currentOrder(j)) ) then
                                         currentOrder(j) := i
                                       end-if
                           end-do

                         3: do !EDD
                             if(currentOrder(j) = 0) then
                                         currentOrder(j) := i
                                       elif( dueDate(i) < dueDate(currentOrder(j)) ) then
                                         currentOrder(j) := i
```

```
                end-if
    end-do

  4: do !EDDP
            if(currentOrder(j) = 0) then
              currentOrder(j) := i
            elif( SOPriority(i) > SOPriority(currentOrder(j)) ) then
              currentOrder(j) := i
            elif( SOPriority(i) = SOPriority(currentOrder(j)) ) then
              if( orderDueDate(parentCO1(i)) <
orderDueDate(parentCO1(currentOrder(j))) ) then
                            currentOrder(j) := i
              end-if
            end-if
    end-do

  5: do  !ODDT
      if(currentOrder(j) = 0) then
                  currentOrder(j) := i
                elif( SIndex(i) > SIndex(currentOrder(j)) ) then
                  currentOrder(j) := i
                end-if
    end-do

  6: do  !WSPT
            if(currentOrder(j) = 0) then
                        currentOrder(j) := i
                      elif( SOPriority(i) >
SOPriority(currentOrder(j)) ) then
                        currentOrder(j) := i
                      end-if
    end-do



          7: do  !TWK
      if(currentOrder(j) = 0) then
                  currentOrder(j) := i
                elif( SIndex(i) < SIndex(currentOrder(j)) ) then
                  currentOrder(j) := i
                end-if
    end-do

  8: do  !SOPN
      if(currentOrder(j) = 0) then
                  currentOrder(j) := i
                elif( SIndex(i) < SIndex(currentOrder(j)) ) then
                  currentOrder(j) := i
                end-if
    end-do

  9: do  !SPT
      if(currentOrder(j) = 0) then
                  currentOrder(j) := i
                elif( OP(i) < OP(currentOrder(j)) ) then
                  currentOrder(j) := i
```

```
                                        end-if
                                end-do

                          end-case

                         end-if
                        end-do

                        if( currentOrder(j) > 0) then
                          simPST(currentOrder(j)) := simulatorClock
                          simPET(currentOrder(j)) := simulatorClock
                          onhand(j) := onhand(j)-1
                          readyOrder(currentOrder(j)) := 2

                          if(parentSO(currentOrder(j)) > 0) then
                            numberOfchild(parentSO(currentOrder(j))) :=
numberOfchild(parentSO(currentOrder(j))) -1
                            if( numberOfchild(parentSO(currentOrder(j))) = 0) then
                              readyOrder(parentSO(currentOrder(j))) := 1
                            end-if
                          end-if
                          currentOrder(j) := 0
                        else
                          noMorePart := true
                        end-if
                      end-do

             end-if
             writeln("nextArrivaltime(" + j + ")" + nextArrivalTime(j))

            end-do

            !Update machine status
            forall(m in MC) do
              if(simulatorClock >= nextAvailableTime(m) ) then

                if( currentJob(m) > 0) then
                  if(parentSO(currentJob(m)) > 0 ) then
                        numberOfchild(parentSO(currentJob(m))) :=
numberOfchild(parentSO(currentJob(m))) -1
                        if( numberOfchild(parentSO(currentJob(m))) = 0) then
                          readyOrder(parentSO(currentJob(m))) := 1
                        end-if
                      end-if
                    end-if
                  end-if
  end-do


            forall(m in MC) do
              if(simulatorClock >= nextAvailableTime(m) ) then

                currentJob(m) := 0

                forall(i in SO|requestedMachine(i) = MachineID(m) and readyOrder(i) = 1) do
                        case initMachineSubDispatch of
```

```
                        1: do !ATC
                            NOO := getInfoToCO(i)
                            temp_exp := -1*maxlist( 0,orderDueDate(parentCO1(i)) - pathToCO(i) -
simulatorClock) /5*17.5
                            SIndex(i) :=  SOPriority(i) * exp( temp_exp )
                          end-do

                        2: do !ATC1
                            NOO := getInfoToCO(i)
                           temp_SIndex :=  simulatorClock + pathToCO(i) - orderDueDate(parentCO1(i))
                                          if( temp_SIndex > 0) then
                                            SIndex(i) := SOPriority(i) /OP(i)
                                          else
                                            SIndex(i) := temp_SIndex
                                          end-if
                          end-do

                        5: do  !ODDT
                            NOO := getInfoToCO(i)
                           temp_SIndex :=  simulatorClock + pathToCO(i) - orderDueDate(parentCO1(i))
                                          if( temp_SIndex > 0) then
                                            SIndex(i) := temp_SIndex * SOPriority(i)
                                          else
                                            SIndex(i) := temp_SIndex
                                          end-if
                          end-do

                        7: do  !TWK
                            NOOS := getInfoSlack(i)
                                 SIndex(i) := pathToCOSlack(i)
                          end-do

                        8: do  !SOPN
                            NOOS := getInfoSlack(i)
                                 SIndex(i) :=  (orderDueDate(parentCO1(i)) - pathToCOSlack(i) -
simulatorClock)/ numberOfOPSSlack(i)
                          end-do


                end-case
                          end-do


              forall(i in SO|requestedMachine(i) = MachineID(m) ) do
                  !writeln("Job(" + i + "): OP:" + OP(i) )
                         if(readyOrder(i) = 1 ) then

                           case initMachineSubDispatch of
                                    1: do !ATC
                                        if(currentJob(m) = 0) then
                                                        currentJob(m) := i
                                                      elif( SIndex(i) > SIndex(currentJob(m)) ) then
                                          currentJob(m) := i
                                        end-if
                                      end-do
```

```
2: do  !ATC1
    if(currentJob(m) = 0) then
                currentJob(m) := i
              elif( SIndex(i) > SIndex(currentJob(m)) ) then
      currentJob(m) := i
    end-if
  end-do

3: do !EDD
    if(currentJob(m) = 0) then
                currentJob(m) := i
              elif( dueDate(i) > dueDate(currentJob(m)) ) then
      currentJob(m) := i
    end-if
  end-do

4: do !EDDP
    if(currentJob(m) = 0) then
                currentJob(m) := i
              elif( SOPriority(i) > SOPriority(currentJob(m)) ) then
                currentJob(m) := i
              elif( SOPriority(i) = SOPriority(currentJob(m)) ) then
                if( orderDueDate(parentCO1(i)) <
orderDueDate(parentCO1(currentJob(m))) ) then
                  currentJob(m) := i
                 end-if
              end-if
  end-do

5: do  !ODDT
    if(currentJob(m) = 0) then
                currentJob(m) := i
              elif( SIndex(i) > SIndex(currentJob(m)) ) then
      currentJob(m) := i
    end-if
  end-do

6: do  !WSPT
    if(currentJob(m) = 0) then
                currentJob(m) := i
              elif( SOPriority(i)/OP(i) >
SOPriority(currentJob(m))/OP(currentJob(m)) ) then
      currentJob(m) := i
    end-if
  end-do



      7: do  !TWK
    if(currentJob(m) = 0) then
                currentJob(m) := i
              elif( SIndex(i) < SIndex(currentJob(m)) ) then
      currentJob(m) := i
    end-if
  end-do
```

```
8: do  !SOPN
     if(currentJob(m) = 0) then
                    currentJob(m) := i
                 elif( SIndex(i) < SIndex(currentJob(m)) ) then
       currentJob(m) := i
     end-if
   end-do

9: do  !SPT
     if(currentJob(m) = 0) then
                    currentJob(m) := i
                 elif( OP(i) < OP(currentJob(m)) ) then
       currentJob(m) := i
     end-if
   end-do

end-case

              end-if
            end-do
            writeln("currentJob(" + m + ")" + currentJob(m))



     if( currentJob(m) > 0) then
              if(readyOrder(currentJob(m)) = 1 ) then
                simPST(currentJob(m)) := simulatorClock
                simPET(currentJob(m)) := simPST(currentJob(m)) + OP(currentJob(m))
                tempPET := simPET(currentJob(m))
                nextAvailableTime(m) := simPET(currentJob(m))
                readyOrder(currentJob(m)) := 2

              end-if
            end-if

          end-if
          writeln("nextAvailabletime(" + m + ")" + nextAvailableTime(m))

 end-do



!determine next event time
tempClock := simulatorClock
firstTime := true
forall(j in Part | RWType(j) = 1) do
  if( tempClock < nextArrivalTime(j) ) then
    if( firstTime = true ) then
      simulatorClock := nextArrivalTime(j)
      firstTime := false
    else
      simulatorClock := minlist(simulatorClock, nextArrivalTime(j) )
    end-if
  end-if
end-do
```

```
    forall(m in MC) do
      if( tempClock < nextAvailableTime(m) ) then
        if( firstTime = true ) then
          simulatorClock := nextAvailableTime(m)
          firstTime := false
        else
          simulatorClock := minlist(simulatorClock, nextAvailableTime(m) )
        end-if
      end-if
    end-do


    allOrderDone := true
    forall(i in SO) do
      if( readyOrder(i) < 2 ) then
        allOrderDone := false
      end-if
    end-do

  until ( allOrderDone = true)


  forall(k in CO) do
    forall(i in SO | parentCO(i) = k) simOrderPET(k) := simPET(i)

    trad := simOrderPET(k)-orderDueDate(k)
    if( trad > 0) then
      simOrderTardiness(k) := simOrderPET(k)-orderDueDate(k)
    else
      simOrderTardiness(k) := 0
    end-if
  end-do

  Objective := 0
  forall(k in CO) do
    writeln("tradiness(" + k + "):" + orderPriority(k)*orderQuantity(k)*simOrderTardiness(k) )
  end-do

  Objective := sum(k in CO) orderPriority(k)*orderQuantity(k)*simOrderTardiness(k)
  Objective1 := sum(k in CO) orderQuantity(k)*simOrderTardiness(k)

  initialSolDispatch := Objective

  writeln("Objective:" + Objective  + " ;CPUTime:" + (gettime-starttime) )
  writeln("Objective1:" + Objective1 )


  forall(k in CO) writeln("CO--orderTardiness(", k,"):", getsol(simOrderTardiness(k)))
  forall(k in CO) writeln("CO--PET(", k,"):", getsol(simOrderPET(k)))
  forall(k in CO) writeln("CO--DueDate(", k,"):", getsol(orderDueDate(k)))
  forall(k in CO) writeln("CO--PET(", k,"):", getsol(simOrderPET(k)))
  forall(i in SO) writeln("SO--PST(", i,"):", getsol(simPST(i)), ";PET(",i,"):", getsol(simPET(i)),
";OP(",i,"):", getsol(OP(i)) )


end-procedure
```