UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE PRESENTE À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

OFFERTE À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

PAR

YANG YANG

ALGORITHMS FOR TWO-STAGE FLOW-SHOP WITH A SHARED MACHINE IN STAGE ONE AND TWO PARALLEL MACHINES IN STAGE TWO

ABSTRACT

Scheduling problems may be encountered in many situations in everyday life. Organizing daily activities and planning a travel itinerary are both examples of small optimization problems that we try to solve every day without realizing it. However, when these problems take on larger instances, their resolution becomes a difficult task to handle due to prohibitive computations that generated.

This dissertation deals with the Two-Stage Flow-shop problem that consists of three machines and in which we have two sets of jobs. The first set has to be processed, in this order, by machine M_1 and then by machine M_2. Whereas, the second set of jobs has to be processed, in this order, by machine M_1 and then by machine M_3. As we can see, machine M_1 is a shared machine, and the other two machines are dedicated to each of the two subsets of jobs.

This problem is known to be strongly NP-Hard. This means there is a little hope that it can be solved by an exact method in polynomial time. So, special cases, heuristic, and meta-heuristic methods are well justified for its resolution.

We thus started in this thesis to present special cases of the considered problem and showed their resolution in polynomial time.

In the approximation front, we solved the considered problem with heuristic and meta-heuristic algorithms.

In the former approach, we designed two heuristic algorithms. The first one is based on Johnson's rule, whereas the second one is based on Nawez, Enscore, and Ham algorithm. The experimental study we have undertaken shows that the efficiency and the quality of the solutions produced by these two heuristic algorithms are high.

In the latter approach, we designed a Particle Swarm Optimization algorithm. This method is known to be popular because of its easy implementation. However, this algorithm has many natural shortcomings. We thus combined it with the tabu search algorithm to compensate the negative effects. The experimental study shows that the new hybrid algorithm outperforms by far not only the standard Particle Swarm Optimization, but also the tabu search method we also designed for this problem.

ACKNOWLEDGEMENTS

First of all, I must give my most sincere thanks to my supervisor Professor Djamal Rebaïne, for his availability, his involvement and his support throughout his research. I am particularly grateful that he was attentive to my ideas and comments, and also for responding too many of my questions. Without this assistance and dynamism, this research would probably not have been born.

Research work could not be done without financial support. As such, I would like to thank Professor Djamal Rebaïne again, because he gave me a scholarship to reduce much of my burden, allowing me to focus on my research.

Secondly, I would also like to thank my parents deeply. They really sacrificed a lot for me in the past 31 years. It is difficult to find the language to express this gratitude. I can only say that, at present, my greatest wish is to be able to ensure that they live a happy life forever.

Finally, I would also like to thank all who helped me in the past four years.

TABLE OF CONTENTS

ABSTRACT I
ACKNOWLEDGEMENTS III
TABLE OF CONTENTS IV
LIST OF TABLES VI
LIST OF FIGURES VII
Chapter 1 General introduction 8
Chapter 2 Scheduling problems 12
2.1 Introduction 12
2.2 Problem description 15
2.3 Classification of scheduling problems 16
2.4 Flow-shop problem 23
Chapter 3 Concepts of complexity theory 25
3.1 Introduction 25
3.2 Class P and NP 25
3.3 NP-hard and NP-complete problems 26
3.4 Combinatorial optimization problems 28
3.5 Exact algorithms 29
3.5.1 Branch and bound 29
3.5.2 Dynamic programming 30
3.5.3 Reduction methods 30
3.5.4. Constructive methods 31
3.6 Approximation approach 31
3.6.1 Heuristic algorithms 31
3.6.2 Meta-heuristic algorithms 32
3.7. Solving scheduling problems 35
Chapter 4 Two-stage Flow-shop with a shared machine in stage one and two parallel machines in stage two 37
4.1 Introduction 37
4.2 Study of special cases 38
4.2.1 First special case: standard two-stage flow-shop 38
4.2.2 Second special case: constant processing times 40
4.2.3 Third special case: Large processing times in Stage 1 48
4.2.4 Fourth special case: Large processing times in Stage 2 53
Chapter 5 Heuristic approach 57

5.1 Introduction 57
5.2 Heuristic algorithms of FSP 57
5.2.1 CDS heuristic algorithm 58
5.2.2 Palmer heuristic algorithm 59
5.2.3 RA heuristic algorithm 59
5.2.4 NEH heuristic algorithm 59
5.2.5 Gupta heuristic algorithm 60
5.3 Heuristic design for $F3|M1 \rightarrow M2; M1 \rightarrow M3|Cmax$ 60
5.3.1 A heuristic based on Johnson's rule 61
5.3.2 A heuristic based on NEH 65
5.3.3 Experimental study 66
Chapter 6 Meta-heuristic approach 70
6.1 Introduction 70
6.2 Tabu Search (TS) 70
6.2.1 Neighbourhood search 71
6.2.2 Principles of Tabu search 73
6.3 Particle Swarm Optimization (PSO) 75
6.3.1 Principles of PSO 75
6.3.2 Process of PSO algorithm 78
6.4 Encoding 79
6.5 Hybridization 82
6.5.1 Features of TS and PSO 82
6.5.2 Design of the hybrid algorithm 84
6.5.3 Experimental study 88
Conclusion 90
References 92

LIST OF TABLES

Table 2.1: Processing times of jobs 15
Table 2.2: Processing times of jobs for Example 2.2 23
Table 4.1: Processing times of jobs for Example 4.1 39
Table 5.1: Set of two-machine flow shop problems 58
Table 5.2: Processing times of jobs for Example 5.1 63
Table 5.3: Schedule two subsets by Johnson's rule 63
Table 5.4: Processing times of jobs for Example 5.2 66
Table 5.5.1: Simulation of both heuristics where processing times are in [50, 100] . 68
Table 5.5.2: Simulation of both heuristics where processing times are in [20, 100] . 68
Table 5.5.3: Simulation of both heuristics where processing times are in [10, 100] . 69
Table 5.5.4: Simulation on the heuristic based on Johnson's rule 69
Table 6.1: Processing times of jobs for Example 6.1 72
Table 6.2: Neighbourhood of solution $x = (3,4,2,1)$ 72
Table 6.3: Tabu list after the first iteration 73
Table 6.4: Tabu list after the second iteration 73
Table 6.5: Position of the particle and the corresponding ROV value 80
Table 6.6: After SWAP operations, the particle component position is adjusted 82
Table 6.7: Characteristics of TS and PSO approaches 83
Table 6.8: Initial tabu list and swarm 85
Table 6.9: The tabu list and swarm after first iteration 85
Table 6.10: The tabu list and swarm after second iteration 86
Table 6.11: The tabu list and swarm (Unified Pardon Rule) 86
Table 6.12: The tabu list and swarm (Prioritized Pardon Rule) 86
Table 6.13.1: Simulation of hybrid PSO where processing times are in [50, 100] 89
Table 6.13.2: Simulation of hybrid PSO where Processing times are in [20, 100] 89
Table 6.13.3: Simulation of hybrid PSO where Processing times are in [10, 100] 89

LIST OF FIGURES

Figure 1.1: The model studied 10
Figure 2.1 CIM and a production control system 13
Figure 2.2 Two Gantt charts for Example 2.1 16
Figure 2.3: Gantt chart for order A→B→C→D 24
Figure 2.4: Gantt chart for order B→D→C→A 24
Figure 3.1: Relationship between NP, P, and NPC classes 27
Figure 4.1: Two-stage flow-shop with one shared machine in stage one and two parallel machines in stage two 38
Figure 4.2: Divide the problem into two standard two-stage flow-shop problems 39
Figure 4.3: Gantt chart for the case $p_1 = p_2 = p$ 41
Figure 4.4: Size of J_1 larger than that of J_2 44
Figure 4.5: Size of J_2 larger than that of J_1 45
Figure 4.6: $\left\lfloor \frac{n_2}{k} \right\rfloor < n_1$ 47
Figure 4.7: $\left\lfloor \frac{n_2}{k} \right\rfloor \geq n_1$ 47
Figure 4.8: Crossed scheduling for the jobs of J_1 and J_2 50
Figure 4.9: J_1 is processed before J_2. 50
Figure 4.10: No change on the makespan after inserting job j 54
Figure 4.11: One job of another subset between two jobs of either J_1 or J_2 55
Figure 5.1: Minimal value of two functions. 62
Figure 6.1: Model of the swarm of PSO 76
Figure 6.2: Updating the particle 77

Chapter 1

General introduction

The theory of scheduling is one well-established discipline of combinatorial optimization field. A scheduling problem involves organizing, over time, the realization of a set of jobs over a set of resources, usually named as machines.

A job j_i is composed of n different operations $\{O_{i1}, O_{i2}, \dots, O_{in}\}$, and every operation requires a period of time t_{ij}. The machine is a kind of technique or human resource to be used to carry out jobs, and it is available in limited proportions and capacities.

The evaluation of a scheduling solution is compared to one or more goals of performance that satisfy certain conditions of treatment.

The popularity of scheduling theory stems from the fact that a multitude of situations encountered in practice within companies and organizations that can be reduced to the problems of scheduling. For example, machines may represent processors and jobs, nurses and patients, workers and products, etc.

Among the scheduling problems that have been studied in the literature, the problem of serial workshops (Flow-shop Scheduling Problem, FSP) is one of the first that was studied in the early fifties. Johnson, in 1954, was the first to examine and design an efficient solution for the Flow-shop with two machines [Johnson, 1954]. Since then, many studies appeared in the literature. However, experience has shown a gap between

the theory and what actually happens in the centres of production. We may mention the following constraints that the theory of scheduling has not considered until recently.

1. The latency of the work corresponding to the different time needed between the end of an operation and the beginning of the next operation in the same job. For example, the cooling time before their next job manipulation. These times, in some cases to a significant degree, must not be ignored, but in others cases, we could incorporate latency into the processing time of an operation to reduce the complexity of the problem. However, the correctness of the solution will be only slightly influenced or not at all affected.
2. Resources constraints corresponding to the situation where jobs cannot be processed unless sufficient (human or else) resources are available.

Research in scheduling can be divided into two parts: modelling and algorithm design. The modelling part is about designing appropriate mathematical models to capture the complexity of real problems, whereas the algorithm design part is about the design of analysis of algorithms, the convergence algorithms and the optimization quality.

In this dissertation, we study the model of flow-shop with two stages as Figure 1.1. The criterion we seek to minimize the overall completion time of the jobs, known as the makespan. This problem may be defined as follows.

We are given a set J of n independent jobs partitioned in two disjoint subsets that have to be scheduled on a two-stage flow-shop: the first stage contains one machine and the second stage contains two dedicated machines. The first subset of jobs is

processed first by the machine of the first stage, and then by on machine of the second stage. The second subset is processed by the machine of the first stage, the by the other machine of the second stage. We assume that all of the jobs are available at time 0 and have exactly two operations executed on two different machines; the transportation time and latency are included in the processing time of operation; pre-emption is not allowed and the machines in the model are always available and can process only one job at one time [Chikhi, Boudhar, and Soukhal, 2011]. This problem is strongly NP-hard [Tuong Soukhal and Miscopein, 2009].

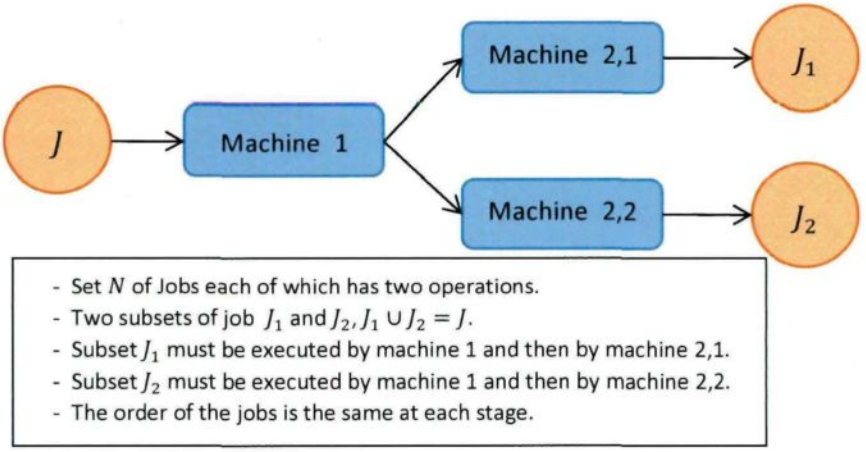


Figure 1.1: The model studied

Our study is three folds. First, we seek polynomial time algorithms to solve this model in some special cases, in our case the processing times are special pattern. The second goal is to solve the general model by two meta-heuristic algorithms. The third goal is to improve the original meta-heuristic and make it more efficient for our model.

This thesis is divided into six chapters. Besides this present chapter, Chapter 2 introduces the different models encountered in the theory of scheduling. Chapter 3 is devoted to the basic concepts used in the complexity theory, along with the different

methods used when confronted with an NP-difficult problem. Chapter 4 first proposes a formulation of the problem we are considering, and then presents several special cases along with their solving algorithms and proofs of their optimality. Chapter 5 is devoted to the study of the heuristic approach. In this chapter, we proposed two new algorithms along with an experimental study to discuss their performance. Chapter 6 is about the meta-heuristic approach. In this chapter, we first proposed two meta-heuristic algorithms: one is based on the tabu search approach, and the second is based on the particle swarm optimization approach. After studying their respective performance through an experimental simulation, we proposed a third approach which is a hybridisation of the two approaches. Again an experimental study is performed to see how efficient this new approach is. We close this dissertation by a conclusion.

Chapter 2

Scheduling problems

2.1 Introduction

With the development of science and technology, the scale of production has become increasingly important, and the process of production has also become more and more complicated, and market competition is getting increasingly fierce. In this environment, organizations have to face increasingly greater numbers of problems, such as the control of production process in response to changing production planning, and also how to maximize their interests or efficiencies.

As a solution to these problems, [Harrington, 1974] introduced the concept of CIM (computer integrated manufacturing), as pictured by Figure 2.1. CIM is a manufacturing approach in which computers are used to control the entire production process [Serope and Steven, 2006]. In a CIM system, functional areas such as design, analysis, planning, purchasing, cost accounting, inventory control and distribution are all linked through the computer with factory floor functions such as material handling and management, thereby allowing monitoring of all of the operations.

The production plan is an important component of CIM as it plays a substantial role in the entire operation of an enterprise. The task of the production plan is to maximize the benefits of the targeted companies. The development of a production plan is

generally considered as a static situation. The production plan is executed by the scheduling system.

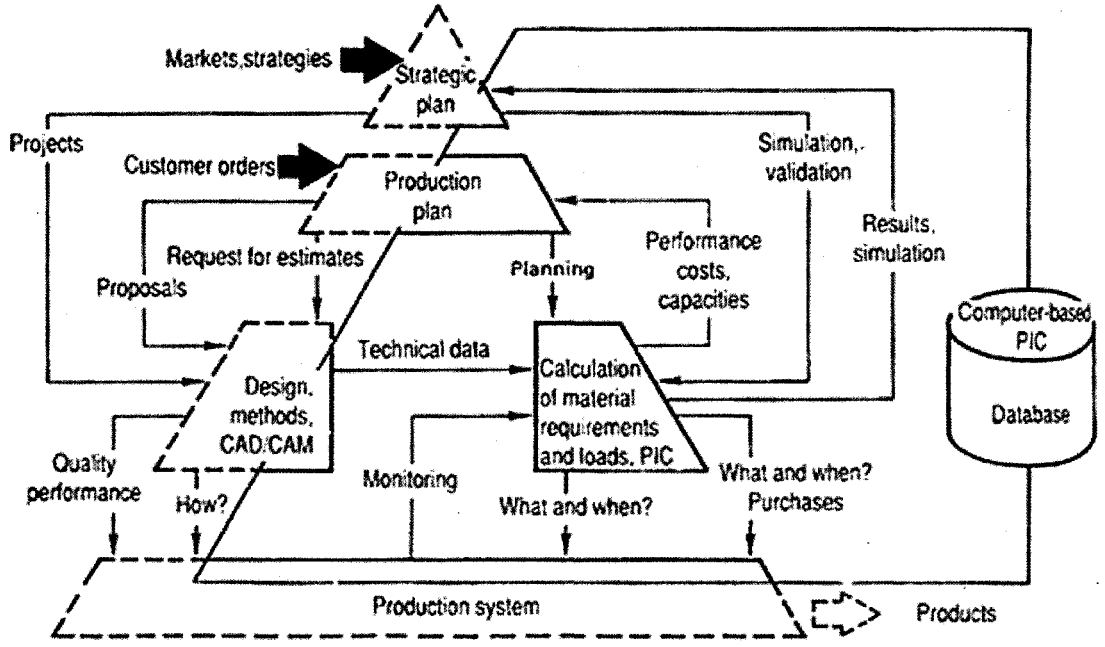


Figure 2.1 CIM and a production control system [Toni and Tonchia, 1998]

Production scheduling plans the production process as a decisive advantage, which is the core of the production plan. An efficient scheduling method is a key to improve the efficiency of production. Improvements in production scheduling now allow us to pay more attention to improving the efficiency of production and resource use. Production scheduling is based on the production plan and depends on market demands and conditions and technical equipment. The task of production scheduling is to plan and organize the production process. Its main factors are as follows:

(1) The number of products.

(2) Production Line.

(3) Production Order.

(4) The production constraints.

In the field of theoretical research, production planning and production scheduling are referred to as scheduling problems. The difference between them is that the production planning principally considers the long-term plan while neglecting or simplifying production constraints. In contrast, production scheduling considers the plan in the short term; its main purpose is to organize the production and distribute resources over time. So, it must take into account a variety of constraints in the real production environment. Therefore, production scheduling is the process of achieving the production plan.

A scheduling problem involves the organization, over time, of the realization of a set of tasks based on the resource availability.

Production management may differ for different optimization objectives, such as strategy optimization or model optimization of a scheduling problem. Each production environment is almost unique because of the dynamics of the production environment and the diversity of the knowledge production, which makes it difficult to find a corresponding method for all situations. Scheduling problems are usually of optimization nature and are part of the combinatorial optimization class (for more details see Section 3.4). It is worth to mention that the vast majority of scheduling problems turn out to be NP-complete (see the definition in Chapter 3). When a problem is shown to be an NP-complete problem, this means that there is a *little hope* to find an exact algorithm to solve it within a reasonable time. The use of approximation approach or the design of well solvable cases is therefore justified.

2.2 Problem description

Scheduling problems are described basically by two sets:

- $J = \{J_1, J_2, \ldots, J_n\}$ of n jobs that will be executed in the system.
- $M = \{M_1, M_2, \ldots, M_m\}$ of m machines present in the system to process set J.

A scheduling problem involves assigning the set M to complete all jobs of J with some constraints. Scheduling has two constraints: occupation constraints and order constraints. Occupation constraints indicate that each job is executed by at most one machine at the same time and that each machine can process at most one job at the same time; order constraints indicate that each job must be executed in a certain order.

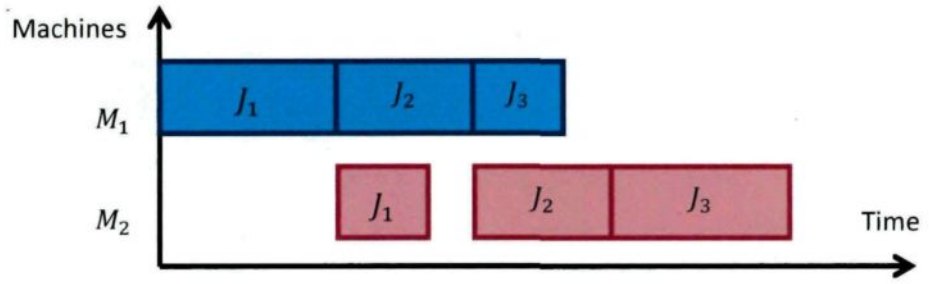
All scheduling solutions can be represented by a diagram called the Gantt chart. This is a type of bar charts, developed by Gantt [Gantt, 1910]. These diagrams help to visualize a solution. Gantt charts illustrate the starting and finishing dates of the terminal elements and summary elements of a project. Terminal elements and summary elements comprise the work breakdown structure of the project. The chart has two perpendicular axes; the horizontal axis represents the time units, while the vertical axis represents the machines that are in the centre.

Example 2.1: Let $J = \{J_1, J_2, \ldots, J_n\}$, with $n = 3$, and $M = \{M_1, M_2, \ldots, M_m\}$, with $m = 2$. Table 2.1 shows the processing time for each job.

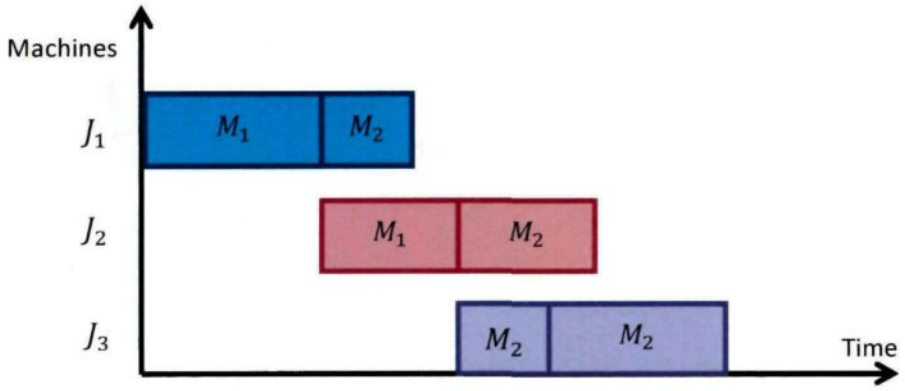
	J_1	J_2	J_3
M_1	4	3	2
M_2	2	3	4

Table 2.1: Processing times of jobs

Figure 2.2 shows the Gantt chart associated with the scheduling solution. From this diagram, we can determine the value of the criterion we considered. This diagram may help to determine the strength or the weakness of this solution.



(a) A machine-oriented Gantt chart for example 2-1



(b) A job-oriented Gantt chart for example 2-1

Figure 2.2 Two Gantt charts for Example 2.1

2.3 Classification of scheduling problems

The variety of scheduling problems that arise in practice leads to a notation that allows us to classify them. This notation was first proposed by Graham *et al.* (1979) and expanded later by several authors to include other new scheduling problems.

This notation comprises three fields and is of the form $\alpha|\beta|\gamma$. The first field α represents the environment of the machines; the second field β describes the

characteristics of the jobs and the resources that are utilized, and the third field γ represents the criterion (or the set of criteria) we are optimizing.

Let us now go into details. Field α consists of two parameters α_1 and α_2 : $\alpha_1 \in \{\varnothing, \mathrm{P}, \mathrm{Q}, \mathrm{R}, \mathrm{PMPM}, \mathrm{QMPM}, \mathrm{G}, \mathrm{X}, \mathrm{O}, \mathrm{J\,F}\}$ and α_2 are equal to positive integer

1. If $\alpha_1 \in \{\varnothing, \mathrm{P}, \mathrm{Q}, \mathrm{R}, \mathrm{PMPM}, \mathrm{QMPM}\}$: any job comprises one single operation.
2. If $\alpha_1 = \varnothing$: we have one single machine to process the set of jobs. The processing times $\mathrm{p_{ij}}$ are reduced to $\mathrm{p_i}$.
3. If $\alpha_1 \in \{\mathrm{P}, \mathrm{Q}, \mathrm{R}\}$, then we have a set of *m* machines (m > 1) operating in parallel, that is to say each job can be processed by one of the machines $\mathrm{M_1}, \ldots, \mathrm{M_m}$. Usually we distinguish between three models as below according to the speed of the machines:
    a. If $\alpha_1 = \mathrm{P}$, then the machines are identical, and thus the speed is the same for the machines. The processing time $\mathrm{p_{ij}}$ of job $\mathrm{J_i}$ on $\mathrm{M_j}$ are reduced to $\mathrm{p_i}$ for all machines $\mathrm{M_j}$.
    b. b. If $\alpha_1 = \mathrm{Q}$, then the machines have related speeds, and we say that the machines are uniform. Indeed, within this model, the processing times become as $\mathrm{p_{ij}} = \mathrm{p_i}/\mathrm{s_j}$ where $\mathrm{s_j}$ is the speed of machine $\mathrm{M_j}$.
    c. Finally, if $\alpha_1 = \mathrm{R}$, then there is no relationship between their speeds, and the machine are said to be unrelated. In this case, the processing times depend on the machine in which the jobs are processed. So the notation $\mathrm{p_{ij}}$ denote the processing time of job $\mathrm{J_i}$ on machine $\mathrm{M_j}$.

4. If $\alpha_1 = \mathrm{PMPM}$ or $\alpha_1 = \mathrm{QMPM}$, then we have multi-purpose machine model with identical and uniform speeds, respectively.
5. If $\alpha_1 \in \{\mathrm{G}, \mathrm{X}, \mathrm{O}, \mathrm{J}\ \mathrm{F}\}$, then we have a multi-operational model known as the general shop. This means that the jobs comprise several operations. We indicate the general shop by setting $\alpha_1 = \mathrm{G}$. Job shops, flow-shops, open shops, and mixed shops are special cases of the general shop. The differences are based on the nature of the job routing.
    a. The job shop is indicated by $\alpha_1 = \mathrm{J}$. In this case, an associated route through the machine is associated with each job.
    b. The Flow-shop is indicated by $\alpha_1 = \mathrm{F}$. In this case, the route though the machine is the same for the whole set of jobs; by convention each job start from machine 1, and then machine 1, and so on, until reaching machine m. If addition, if the same order of processing is kept through all the machines, then we have a restricted model (which exists in its own right) known as the permutation flow shop. There are some situations where this model is dominant over the set of all the solution of a flow shop model.
    c. The open shop is indicated by $\alpha_1 = \mathrm{O}$. The route of processing of the jobs, in this case, is not known in advance, but is part of the solution.
    d. The mixed shop indicated by $\alpha_1 = \mathrm{X}$ is a combination of a flow shop and an open shop (the combination of a job shop and an open shop is known as the general mixed shop).

Parameter α_2 denotes the number of machines utilized if $\alpha_2 = k$. However, this value is fixed in advance. Thus, it is part of the input of the problem. If the number of machines is arbitrary, we set $\alpha_2 = \varnothing$.

The second field β describes the characteristics of the jobs and resources utilized. It comprises eight parameters $\beta_1\beta_2\beta_3\beta_4\beta_5\beta_6\beta_7\beta_8$.

- $\beta_1 \in \{\varnothing, \text{pmtn}\}$ indicates whether pre-emption is permitted or not.
- $\beta_2 \in \{\varnothing, \text{res}\}$ indicates whether resources considerations are taken into account. The presence of *res* means that additional resources other than machines (such as manpower) are needed for the processing of jobs. In case parameter *res* is not empty, then it is further divided into three fields "res $\lambda\sigma\delta$ " denoting respectively the number of resources, the total amount of available resources per time unit, and the maximum resource requirements of operations. Note that a dot "." for each of these parameters indicates that the corresponding variable can take any integer value, whereas a positive integer indicates that the corresponding variable is fixed. Moreover, researchers in this area distinguish between renewable and non-renewable resources.
- $\beta_3 \in \{\varnothing, \text{prec}\}$ indicates whether precedence relations exists between jobs. Sometimes, we only consider special graphs. In this case, we have to specify the nature of this graph. Usually we consider tree (denoted by tree), chains (denoted by chains), series-parallel graphs (denoted by sp-graph).
- $\beta_4 \in \{\varnothing, r_j\}$ indicates whether release dates are associated with jobs. Otherwise, we assume that all jobs are available for processing at time 0.

- $\beta_5 \in \{\varnothing, p_{ij} = 1,\ p_i = p,\ p_1 \leq p_{ij} \leq p_2\}$ indicates special values that can be taken by the processing times of the jobs. Other values are also possible. The case of $\varnothing$ indicates that the processing times are arbitrary.
- $\beta_6 \in \{\varnothing, d_i\}$ indicates whether deadlines for the jobs are considered.
- $\beta_7 \in \{\varnothing, n_i \leq k\}$ indicates the maximal number of operations of jobs in the case of a job shop model.
- $\beta_8 \in \{\varnothing,$ nowait$\}$ indicates in the case of shop models whether the processing of the jobs is done in a no-wait manner. This means that once a job is completed in a machine, it should immediately, without any waiting time, start its processing in the following machine.

The third field γ represents the criterion we are optimizing. The quality of a schedule is evaluated to a given criterion (or to a set of given criteria if we are in a multi-objective environment). In scheduling theory, several criteria are considered in the literature to build a solution; Mellor [Mellor, 1966] enumerated 26 different criteria. Associated with each job J_i is a function f_i that depends on the completion time C_i of that job. Basically two type of objective function are considered in the literature:

1. Bottleneck objective function $f_{\max} = \max_{1 \leq i \leq n} f_i(C_i)$.
2. Sum of objective functions $\sum_{i=1}^{n} f_i(C_i)$.

Even though Mellor [Mellor, 1966] has enumerated more than 27 criteria (some of them are equivalent between each other), the most common criteria utilized in the literature in building a scheduling solution are as follows:

a. The overall completion time (known as the makespan): $C_{\max} = \max_{1 \le i \le n} C_i$.

b. The mean finish time: $\sum_{i=1}^{n} C_i$.

c. The maxim lateness: $L_{\max} = \max_{i=1}^{n} \{C_i - d_i\}$.

d. The number of tardy jobs: $\sum_{i=1}^{n} U_i$ where $U_j = 1$ if $C_i \le d_i$, 0 otherwise.

e. The total tardiness: $\sum_{i=1}^{n} \max\{0, C_i - d_i\}$.

Let us first mention that when deadline are specified, then in some cases there is no need to minimize an objective function. The only problem we need to solve is to find a feasible solution. If it is the case, then the field γ = --.

Let us also mention that we usually differentiate between two types of criteria: regular and non-regular.

Definition 2.1: A criterion is regular if it is non-increasing with respect to the completion times of the job.

Most of the research in scheduling theory has been done under the assumption that criteria considered are regular. However, a few papers appeared in the literature in which non regular criteria are also studied (for more details see for e.g. [Raghavachari, 1988].

Definition 2.2: A schedule is called semi active when machines never idle if they can process jobs.

Theorem [Brucker, 1995] Semi-active schedules are dominant with respect to regular criteria.

To conclude a scheduling problem is fully described by the above notation. As an illustration, the following problems denote:

1. $1|prec, pmtn|C_{\max}$: scheduling problem with a single machine, the jobs are related by a general precedence graph, and pre-emption is allowed. The criterion to minimize is the makespan.
2. $P_m|tree, p_i=1, r_i|L_{\max}$: scheduling problem with m (fixed) identical machines operating in parallel, the precedence constraints form a tree, the processing times are unitary, each job is associated with a release date, and deadline which is stated in the criterion $L_{\max}$ to be minimized.
3. $F_m|no-wait|\sum \omega_i C_i$: flow shop problem with no-wait in process. The criterion to minimize is the mean weighted finish time.

Let us recapitulate. The goal is to build a schedule that generates an optimal (or a near optimal) solution with respect to a given criterion (or several criteria if we are in a multi-objective environment; however this is not the case in this thesis). We will assume throughout this dissertation that all the parameters are known in advance.

Definition 2.3: A solution is feasible if a machine does not process more than one job at a time, and a job is not processed by more than one machine at a time. In addition, depending on the problem, a number of specific characteristics may be requested to be satisfied.

Definition 2.4: We say that a schedule is optimal if it minimizes a given criterion.

2.4 Flow-shop problem

In this dissertation, FSP is the problem we will be focussing on. The Flow-shop scheduling problem (FSP) can be described as follows:

A set of n jobs is to be processed by Stage 1, Stage 2, and so on until reaching Stage m, in that order. Each centre may have more than one machine operating in parallel. The processing time $t_{i,j}$ $(i = 1,2,\dots,n; j = 1,2,\dots,m)$of job i in centre j is given. For the FSP, we usually make the following assumptions:

(1) Each machine can only process one job, at the same time.

(2) A job cannot be processed by two machines at the same time.

The preparation time is included in the processing time, and has nothing to do with the order.

Example 2.2: Let A, B, C, D be 4 jobs, where every job includes two operations O_1, O_2 and the processing times are shown in Table 2.2.

Jobs	t_1	t_2
A	15	4
B	8	10
C	6	5
D	12	7
Total	41	26

Table 2.2: Processing times of jobs for Example 2.2

Here, t_1 is the processing time of operation O_1 and t_2 is the processing time of operation O_2. One machine, M_1, is in operation O_1 and another machine M_2 is in operation O_2. In the above scheduling problem, we use three variables (i, j, k) to express that the operation j of job i is executed by machine k. If the jobs are executed

with order A, B, C, and D, then the Gantt chart that expresses it, as it is pictured by Figure 2.3.

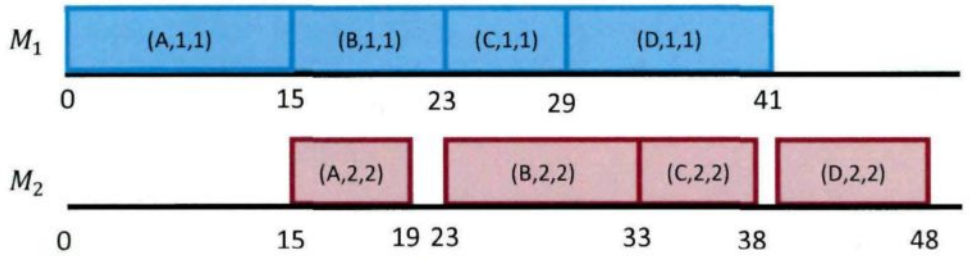


Figure 2.3: Gantt chart for order A→B→C→D

In Figure 2.3, the boxes represent the operations while the length of the box indicates the processing time t_{ijk} of the operation (i, j, k).

In a Gantt chart, a feasible schedule should ensure the order of jobs and that no overlap occurs between boxes. For this example, if we change the job order to B, D, C, and A, and the corresponding Gantt chart is shown in Figure 2.4.

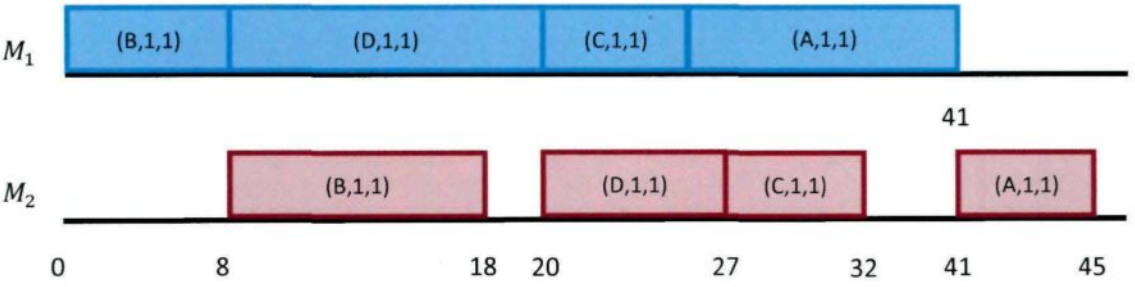


Figure 2.4: Gantt chart for order B→D→C→A

The makespan of the first schedule is 48, while the makespan of the second schedule is 45. So, we may conclude that the second solution is better than the first one.

Chapter 3

Concepts of complexity theory

3.1 Introduction

Complexity theory is used to measure the degree of difficulty to solve given problem. Intuitively, if longer computing time is needed to solve a problem, we could say this problem is harder, otherwise, we say the problem is easier. An important source of information about this theory is the book of [Garey and Johnson, 1979].

3.2 Class P and NP

Polynomial time is the central concept in computational complexity. This is the criterion that determines whether an algorithm finds a solution efficiently. Let us recall that a polynomial time algorithm is an algorithm whose running time is bounded above by a polynomial expression of the size of the input of the considered problem.

Definition 3.1: Class P is the class of problems for which there exists a polynomial time algorithm that solve them.

However, there exist problems for which we do not know whether there exist polynomial time algorithms for their resolution. In order to proceed we need first to introduce another class named the NP class.

Intuitively, NP class represents the set of all decision problems (see the definition below) for which we may not have a known polynomial method to solve them, but if we

were given a candidate answer, we are able to verify that whether it is the right answer of our question in polynomial time. From this definition, it is easy to observe that class P is a subset of class NP.

3.3 NP-hard and NP-complete problems

Besides class P, there is also another subclass of NP class called NP-complete (NPC for short) class. This class contains the hardest problems of NP in the following sense.

Definition 3.2: NP-Complete class is a class that is composed of problems that:

- Belong to class NP.
- If only one problem of that class can be solved in polynomial time, then every problem in that class can also be solved in polynomial time.

The central notion behind NP-completeness is the concept decision problems and the concept of polynomial reduction (transformation).

Definition 3.3: A decision problem is a problem for which the solution is a yes or a no answer.

Definition 3.4: A decision problem L is said to be reducible to another decision problem K if we can transform an instance of problem L into a instance of problem K in such a way there is yes-answer to problem L if, and only if, there is a yes-answer problem K.

In order to prove the NP-completeness of a problem L, from above, we proceed as follows.

1. First, show that the decision version of the considered problem is class NP.

2. Find a known NP-complete problem K and a construct a polynomial reduction from L to K such that there is a yes answer to L if, and only if, there is yes answer to problem K.

In other word, we have to prove that problem L is a special case of problem K. Let us also observe that proving that a problem is NP-hard means only that there exists at least one instance of that problem which is difficult to solve. In other words, there can be instances of that problem which can be solved efficiently.

[Cook, 1971] was the first to prove the existence of such a problem known as the satisfiability problem. Shortly after this important result, 21 other problems were proved to be NP-complete by [Karp, 1972]. Nowadays, thousands of problems have been proved to be NP-complete. In fact, as cited in [Graham *et al.*, 1979] the vast majority of combinatorial optimization problems (the definition follows shortly) are NP-complete.

Figure 3.1 resumes the relationship between NP, P, and NP-complete classes.

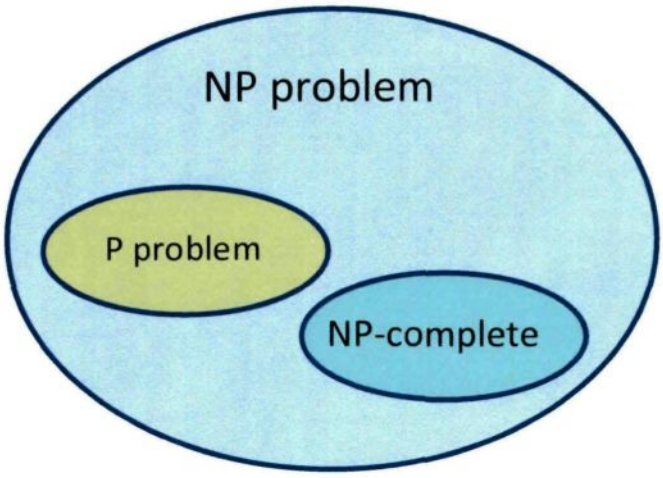


Figure 3.1: Relationship between NP, P, and NPC classes

The most important open question in the complexity theory is whether P = NP, that is to say whether all problems in NP can be solved in polynomial time. This widely believed not to be the case.

Before closing this section, let us make the following observations. First, when we talk about optimization problem problems, we prefer to use the term NP-hard problem when its corresponding decision version is NP-complete. Second, when we talk about the encoding of an input, we mean the binary encoding. However, even if it is not used in practice, one could also use the unary encoding. Within that respect, in the complexity theory, the degree of hardness of a problem is further refined.

Definition 3.5: A problem is NP-complete in the strong sense if it still remains NP-complete even if the input is encoded in unary.

Definition 3.5: A problem is NP-complete in the weak sense if it can be solved in polynomial time within the unary encoding. This type of algorithm is called pseudo-polynomial time.

3.4 Combinatorial optimization problems

Combinatorial optimization is a topic of operations research that consists of finding an optimal solution from a finite set of solution, which constitutes the search (solution) space. These kinds of problems are involved in many areas such as: information technology, timetabling, production scheduling, graph theory, transportation, bioinformatics, and so on. Let us observe that, even though the solution space is finite, in many such problems, exhaustive search is not feasible for the simple reason that the

solution space is huge to enumerate in a reasonable time even with the fastest computer.

Most of combinatorial optimization problems are NP-Complete problem as are scheduling problems. However, for small instances they can still be solved efficiently. In what follows, we describe some resolution techniques that are used to solve scheduling problems.

3.5 Exact algorithms

In what follows we will be discussing a number of approaches used to solve combinatorial optimization problems.

3.5.1 Branch and bound

A branch and bound algorithm consists of breaking up the target problem into successively smaller sub-problems, computing bounds on the objective function associated with each sub-problem, and using them to discard certain of these sub-problems from further consideration. The procedure ends up when each sub-problem has either produced a feasible solution or been shown to contain no better solution than the one already in hand. The best solution found at the end of the procedure is the global optimum. Applying a branch-and-bound algorithm requires specifying the several ingredients (lower bounds, dominance rules, search strategies, etc). It is worth mentioning that the effectiveness of branch and bound algorithms relies not only on the tightness of these ingredients but also on the running time to compute them. For more details on this technique see for e.g. [Balas and Toth, 1985].

3.5.2 Dynamic programming

The method of dynamic programming is an approach, which starts by establishing a recursion to link the optimal solution of the whole problem under consideration to those of its sub-problems. Then by carefully implementing this recursion though the use of a table, we avoid solving sub-problems several times. When the number of different sub-problems is polynomially bounded, the dynamic programming approach generates efficient solutions. For more details on this approach, see for e.g. [Dasgupta *et al.*, 2007].

3.5.3 Reduction methods

This method consists of reducing the problem under study into another problem for which a method of resolution is already known. The most used one are the following. However, one could use other problems to reduce the original problem into them. For more details on this approach, see for e.g. [Levitin, 2003].

1. Graph theory: this approach consists of translating the problem into an equivalent problem of graph theory, such as travelling salesman problem, graph coloration, shortest path, linear assignment, etc.
2. Mathematical programming: this approach consists of formulating the problem into a mathematical program by introducing suitable decision variables and an objective function. We usually try either the linear programming formulations with or without integer variables. The reason is that for those formulations, there exist several resolution methods, and certain of them are quite efficient such as the simplex methods. On the other hand, from the computational point

of view, when we succeed to derive a linear formulation, this means that the corresponding problem can be classified as an easy problem.

3.5.4. Constructive methods

This approach uses some properties related to the problem under study to generate simple rules that may lead to the solution of the considered problem. We may mention Johnson rule that solves to optimality the two-machine problem with respect to the overall completion time criterion.

3.6 Approximation approach

Approximation algorithms are methods that find near optimal solutions. They are often used to solve NP-hard problems since it is unlikely that these problems can be solved exactly by efficient methods.

We usually distinguish between two types of approximation techniques: the heuristic approach and the meta-heuristic approach. In the former, only one solution is generated, whereas in the second approach, several solutions may be generated iteratively.

3.6.1 Heuristic algorithms

Heuristic algorithms usually start with an empty solution. According to some predefined rules for the problem under study, the algorithm expands the partial solution at each iteration until getting into the complete solution. Greedy methods and list scheduling fall into this category of algorithms. The strength of these algorithms is that the solution is produced very quickly, even though there is no guarantee that it is

optimal. Even though, for some problems this way of proceeding may lead to optimal solutions as it is the case for Johnson rule in the two machine flow shop problem for the makespan criterion or the Shortest Processing Time rule in the single machine problem for the mean finish time criterion.

Heuristic algorithms may produce results by themselves, or they may be used in conjunction with optimization algorithms to improve their efficiency (e.g., they may be used to generate good seed values).

Let us mention an important feature of heuristic algorithms. They lend themselves to a mathematical analysis. This analysis measures the distance that separates the optimum solution (that we do not know) to the value of the solution produced by this algorithm. Thus evaluation may be undertaken in the worst case or in the probabilistic case. In the former, the goal if to find an upper bound on this distance; for more details, see for e.g. [Fisher, 1982]. In the latter, we measure this distance by means of average and standard deviation; for more details, see for e.g. [Rinnooy Kan, 1986].

3.6.2 Meta-heuristic algorithms

A meta-heuristic algorithm, also called ameliorative methods, is a method that solves a problem (usually of optimization nature) by iteratively trying to improve a candidate solution over the space of feasible solutions with regard to a given measure of quality, without guaranteeing the optimal solution. Let us point out that the main difference between meta-heuristic algorithms and heuristic algorithms is that the former produce several solutions, whereas the latter generate one single solution.

Popular meta-heuristic algorithms for combinatorial optimization problems include simulated annealing [Kirkpatrick, Gelatt, and Vecchi, 1983], genetic algorithms [Holland, 1975], ant colony optimization [Dorigo, 1992], scatter search [Glover, 1977], tabu search [Glover, 1986], and particle swarm optimization [Kennedy and Eberhart, 1995]. Tabu search algorithm and Particle swarm optimization algorithm are discussed in details in Chapter 6. For the sake of completeness, we present briefly in the following some of the above cited meta-heuristic algorithms. For a general view on these techniques, see for e.g. [Fatos and Ajith, 2008].

1. **Hill climbing (steepest descent) technique:** This method is an iterative search procedure that, starting from an initial feasible solution, progressively improves it by applying a series of local modifications. At each iteration of the algorithm, the algorithm moves to a better feasible solution. The search terminates when no more improvement is possible. The major drawback of this approach is that, since it is somehow greedy, it ends up in a local optimum frequently of low quality. Meta-heuristic algorithms (such as the ones that follow) extend steepest descent methods by allowing the search beyond the first local optimum. An immediate improvement of this technique is to repeat the *hill climbing technique* from several different initial feasible solutions.

2. **Simulated annealing algorithm:** Annealing is the process of slowly cooling a physical system in order to obtain states with globally minimum energy. By simulating such a process, near globally-minimum-cost solutions can be generated in a efficient way. The corresponding algorithm (see for e.g. [Kirpatrick, Gelatt, and Vecchi, 1983]) uses an approach similar to hill-climbing, but from time to time accepts solutions that

are worse than the current solution. The probability of such acceptance decreases over time. In order to apply the Simulated Annealing method to a specific problem, the following parameters must be specified: the search space, the objective function, the neighborhood of solution *S*, the initial temperature, the cooling factor, and the stopping criteria. The choice of these parameters has a significant impact on the efficiency of the method.

***3.* Genetic algorithms:** Genetic algorithm (GA), introduced by [Holland, 1975] is a method that uses techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. Indeed, once the fitness function is defined, which is associated to the objective function, the genetic algorithm consists of improving an initial set of solutions, generated usually at random, through the use of application of the mutation, crossover, and selection operators. Crossover operator consists in generating new solutions by combining candidate solutions, whereas the mutation operator generates a solution by slightly changing a candidate solution. Basically, a genetic algorithm starts from an initial set candidate solutions, say *P*, and repeats the following steps until some stopping criterion is reached:

a. Generate a set of new solutions, say Q, from the set of solutions that can be obtained from *P* through the use of crossover and mutation operators.

b. Choose a subset $T \subset Q \cup P$ according to the selection rule, and set *P* = *T*.

For more details, see for e.g. [Yagiura and Ibaraki, 2001].

3.7. Solving scheduling problems

We present in this section the methodology to follow in order to solve a scheduling problem. Indeed, the first question to ask, when confronted to a scheduling problem, is whether the problem is NP-hard or not. The only way to state that a problem is easy to solve is to exhibit a polynomial solution. If the problem has already been shown to be in the class P, then we usually try to design other solutions with a better time complexity than the existing solution. If, on the other hand, the problem under study is shown to be NP-hard, then several approaches can be tried.

1. We could tight up the NP-hardness by showing the NP-hardness in the strong sense, or by exhibiting a pseudo-polynomial time algorithm.
2. We could also study to what extent the problem can be solved efficiently by designing polynomial solutions to special cases. What we mean by that is we relax some constraints of the problem and see whether the problem is well solvable or it is still NP-hard. In scheduling problem, we usually consider the cases where the processing times are restricted to some particular values, we assume that preemption is permitted; the precedence graphs are of special types if precedence relations exist between jobs, etc.
3. Design exact algorithms (branch and bound, dynamic programming, mathematical formulation, etc.) and try to solve larger input of the problem through the design of ad hoc properties (tight lower bound bounds, strong dominance relations, etc.).
4. The approximation approach by designing either heuristic algorithms or a meta-heuristic algorithms:

a. If we have chosen the heuristic approach, then the goal here is generally to undertake a worst-case analysis and get a better ratio by measuring the distance between the optimal value and the value produced by the heuristic algorithm. A simulation is also performed to measure the effectiveness of the proposed solution.

b. If we have preferred the meta-heuristic approach, then in this case we try to design efficient solutions that solve large instances. In the recent years, many papers appeared in the literature combined two or more met-heuristic algorithms. This approach seems to be promising as the results they produce are by far much better.

Chapter 4

Two-stage Flow-shop with a shared machine in stage one and two parallel machines in stage two

4.1 Introduction

In this dissertation, we study one kind of FSP, named Two-Stage Flow-shop with one shared machine in stage 1 and two parallel machines in stage two. It is denoted by $F3|M_1 \rightarrow M_2, M_1 \rightarrow M_3|C_{max}$. This problem may be described as follows.

We are given a set of n independent jobs to be distributed in two disjoint subsets and scheduled on a two-Stage Flow-shop. The first Stage contains one machine and the second one contains two machines operating in parallel. We assume that all jobs are available at time 0 and have exactly two operations to be executed by the two stages. Furthermore, the jobs in the first subset must be executed on the machine of stage one, and on machine of stage two, whereas, the jobs of the second subset must be processed by the machine of stage one, and the other machine of stage two (as illustrated by Figure 4.1).

The processing times of each job in each Stage are not equal to 0; the transport time between different Stages are included in the processing time; the three machines are always available and can process only one job at a time, and one job can be executed by only one machine at a time. The goal is to minimize the makespan of the the jobs.

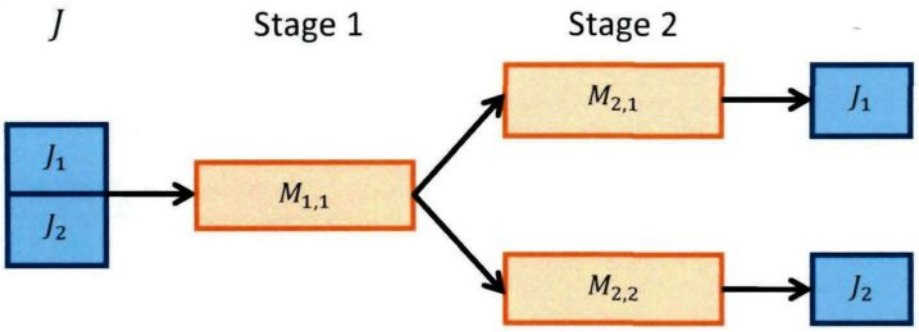


Figure 4.1: Two-stage flow-shop with one shared machine in stage one and two parallel machines in Stage two

Even though a problem is NP-hard, there might still are special instances for which this problem may be solvable in polynomial time. In this section, Then, several special cases for which the corresponding problem is well solved are discussed.

4.2 Study of special cases

This section is devoted to the presentation of special cases that are solvable in polynomial time.

4.2.1 First special case: standard two-stage flow-shop

We consider the following two cases: $J_1 = \emptyset$ or $J_2 = \emptyset$. This means that Stage 2 is reduced to one machine. In this case, this problem corresponds to the standard flow-shop, as illustrated by Figure 4.2.

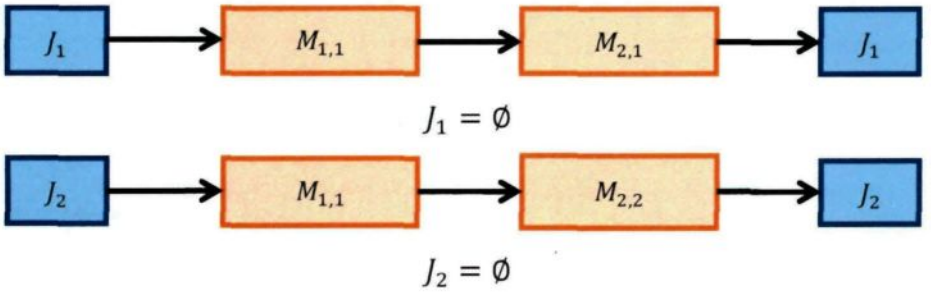


Figure 4.2: Divide the problem into two standard Two-stage flow-shop problems

This problem can then be solved by the Johnson's rule. Johnson's rule may be resumed as follows: If $\min\{p_{i,1}, p_{j,2}\} \leq \min\{p_{i,2}, p_{j,1}\}$, then job i precedes job j.

We can use this rule to construct the optimal schedule of a two-Stage FSP. The following is the corresponding algorithm.

Step 1: Select the job with the shortest processing time. If that processing time is for the first Stage, then schedule the job first. If that processing time is for the second Stage then schedule that job last.

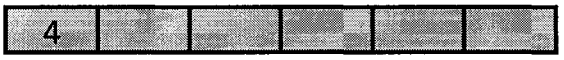
Step 2: Repeat steps 1 until all the jobs have been scheduled.

Example 4.1: Assume we have a two-machine Flow-shop problem, and there are 6 jobs to be executed; the processing times are shown in Table 4.1.

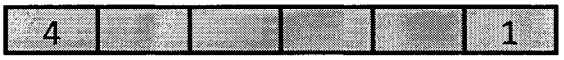
Job	1	2	3	4	5	6
The processing time in Stage 1	10	5	11	3	7	9
The processing time in Stage 2	4	7	9	8	10	15

Table 4.1: Processing times of jobs for Example 4.1

1. The smallest time is located with Job 4 (3, 8). Since the time is in Stage 1, schedule this job first. Eliminate Job 4 from further consideration.

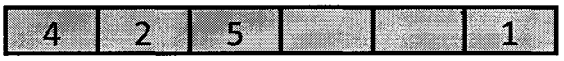


2. The next smallest time is located with Job 1 (10, 4). Since the time is in Stage 2, schedule this job last. Eliminate Job 1 from further consideration.



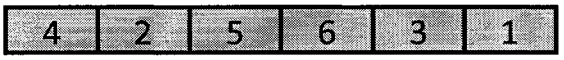
3. The next smallest time after that is located with Job 2 (5, 7). Since the time is in Stage 1, schedule this job first. Eliminate Job 2 from further consideration.

4. The next smallest time after that is located with Job 5 (7, 10). Since the time is in Stage 1, schedule this job first. Eliminate Job 5 from further consideration.



5. The next smallest time after that is located with Job 6 (9, 15). Since the time is in Stage 1, schedule this job first. Eliminate Job 6 from further consideration.

6. The only job left to consider is job 3.



For this schedule, the minimal makespan is 56.

4.2.2 Second special case: constant processing times

In this case, the processing times of the jobs both stage the same. Furthermore, the processing times of the jobs within the same subset are also identical. Formally we have

$$p_{i,k} = \begin{cases} p_1, k = 1,2, if\ i \in J_1, \\ p_2, k = 1,2, if\ i \in J_2. \end{cases}$$

Since there is just one machine in Stage 1, we have that

$$C_{M_{1,1}} = n_1 \times p_1 + n_2 \times p_2.$$

On the other hand, the following is an obvious lower bound on the value of the makespan.

$$min\ C_{max} = C_{M_{1,1}} + \min(p_1, p_2).$$

For the different subsets, the processing times of the jobs may be the same or different. So, we need to distinguish between these two cases.

A. Processing times of the two subsets are identical

This corresponds to the case where no difference between J_1 and J_2. Intuitively, an optimal solution is achieved by any permutation of the jobs.

Theorem 4.1 An optimal solution can be achieved by any permutation, if the processing times of the two subsets are identical and constant.

Proof: Recall that we have $p_1 = p_2 = p$. So, in this case, for any processing sequence, no job is delayed on stage two. So, the completion time of the last job of J in Stage 2 of any sequence is $(n_1 + n_2 + 1)p$ as pictured by Figure 4.3. Thus, the statement of the theorem follows immediately.

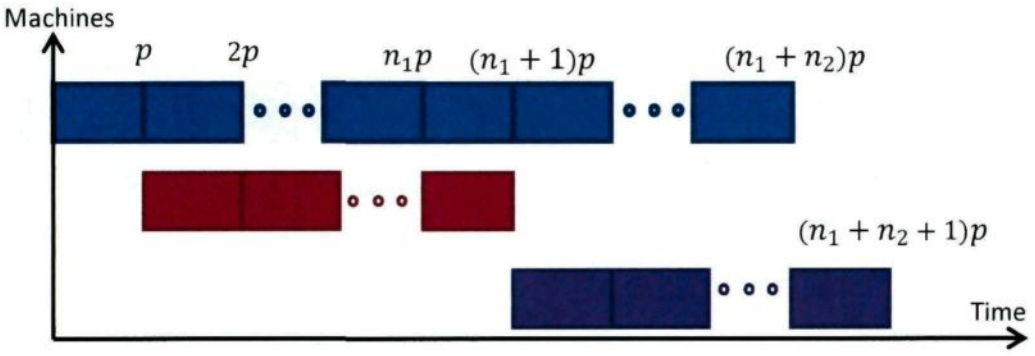


Figure 4.3: Gantt chart for the case where $\boldsymbol{p_1 = p_2 = p}$

B. Processing times of two subsets are different

Without loss of generality, we suppose that $p_1 > p_2$. In what follows, we discuss two cases: $\frac{p_1}{2} \leq p_2 < p_1$ and $\frac{p_1}{k+1} < p_2 < \frac{p_1}{k} (k > 1, k \in Z)$.

Case 2.1 $\frac{p_1}{2} \leq p_2 < p_1$

Let us again distinguish two cases: $n_1 \geq n_2$ and $n_1 < n_2$. Let us start with the first case, and derive the following result:

Theorem 4.2 Let $J_1 = \{x_1, x_2, \dots, x_{n_1}\}, J_2 = \{y_1, y_2, \dots, y_{n_2}\}$, and assume that $n_1 \geq n_2$. If $\frac{p_1}{2} \leq p_2 < p_1$, then $S = (x_1, x_2, \dots, x_{n_1-n_2}, x_{n_1-n_2+1}, y_1, x_{n_1-n_2+2}, y_2, \dots, x_{n_1}, y_{2,n_2})$ is an optimal solution.

Proof: Since the last processed job in S is y_{n_2}, then we first need to prove that the equations

$$c_{y_{n_2},2} = C_{M_{1,1}} + p_2 \qquad (4.1)$$

and

$$c_{y_{n_2},2} > c_{x_{n_1},2}, \qquad (4.2)$$

Hold. Second, $c_{y_{n_2},2}$ is the makespan of S and is also the lower bound of the model. From here, we may conclude that schedule S is optimal. To do so, we proceed by mathematical induction. The start time y_1 is

$$b_{y_1,1} = c_{x_{n_1-n_2+1},1}.$$

So, its start time in Stage 2 is

$$b_{y_1,2} = c_{y_1,1} = b_{y_1,1} + p_2,$$

and its completion time is

$$c_{y_1,2} = b_{y_1,2} + p_2 = b_{y_1,1} + 2p_2.$$

The start time of the second job of J_2 in Stage 1 is

$$b_{y_2,1} = c_{y_1,1} + p_1 = b_{y_1,1} + p_2 + p_1,$$

and its start time in Stage 2 is

$$b_{y_2,2} = \max(b_{y_2,1} + p_2, c_{y_1,2}).$$

Since

$$b_{y_2,1} + p_2 = b_{y_1,1} + 2p_2 + p_1 > c_{y_1,2},$$

then

$$b_{y_2,2} = b_{y_2,1} + p_2 = c_{y_2,1}.$$

It means that job y_2 need not wait to be processed in Stage 2. Hence,

$$c_{y_2,2} = c_{y_2,1} + p_1 = b_{y_2,1} + 2p_2 = c_{x_{n_1-n_2+1},1} + 2p_2.$$

Now, assume that the start time of y_k, $1 \leq k \leq n_2$, is

$$b_{y_k,1} = c_{x_{n_1-n_2+k},1}.$$

Its start time in Stage 2 is

$$b_{y_k,2} = c_{y_k,1} = b_{y_k,1} + p_2,$$

and its completion time is

$$c_{y_k,2} = b_{y_k,2} + p_2 = b_{y_k,1} + 2p_2.$$

So, the start time of y_{k+1} in Stage 1 is

$$b_{y_{k+1},1} = c_{y_k,1} + p_1 = b_{y_k,1} + p_2 + p_1$$

Its start time in Stage 2 is

$$b_{y_{k+1},2} = \max(b_{y_{k+1},1} + p_2, c_{2,j,2}).$$

Since

$$b_{y_{k+1},1} + p_2 = b_{2,j,1} + 2p_2 + p_1 > c_{2,j,2},$$

then

$$b_{y_{k+1},2} = b_{y_{k+1},1} + p_2.$$

It means that job y_{k+1} need not wait to be executed in Stage 2. Therefore,

$$c_{y_{k+1},2} = c_{y_{k+1},1} + p_2 = b_{y_{k+1},1} + 2p_2 = c_{x_{n_1-n_2+k},1} + 2p_2.$$

Therefore, we have

$$b_{y_{n_2},1} = c_{x_{n_1},1},$$

and

$$c_{y_{n_2},2} = c_{y_{n_2},1} + p_2 = b_{y_{n_2},1} + 2p_2 = c_{x_{n_1},1} + 2p_2.$$

As y_{n_2} is the last job processed in S, then $c_{y_{n_2},1} = C_{M_{1,1}}$. Therefore

$$c_{y_{n_2},2} = C_{M_{1,1}} + p_2.$$

Moreover, since $\frac{p_1}{2} \leq p_2 < p_1$, then $c_{y_{n_2},2} = c_{x_{n_1},1} + 2p_2 \geq c_{x_{n_1},1} + p_1$. It follows that (4.1) and (4.2) are established. Therefore, schedule S is optimal (Figure 4.4). □

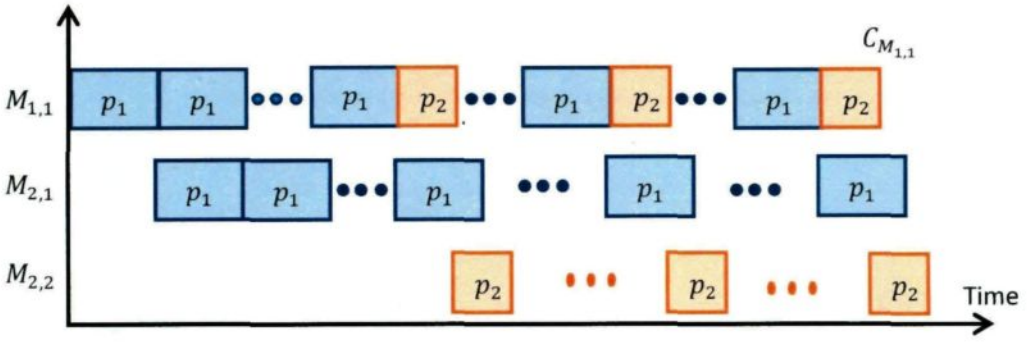


Figure 4.4: Size of J_1 larger than that of J_2.

Now, let us focus our attention to the case where $n_2 > n_1$. We get the following result.

Theorem 4.3: Let $J_1 = \{x_1, x_2, \dots, x_{n_1}\}$, $J_2 = \{y_1, y_2, \dots, y_{n_2}\}$, and assume that $n_1 < n_2$. If $\frac{p_1}{2} \leq p_2 < p_1$, then $S = (x_1, y_1, x_2, y_2, \dots x_{n_1}, y_{n_1}, y_{n_1+1}, y_{n_1+2}, \dots, y_{n_2})$ is an optimal schedule.

Proof: Again, since the last processed job in S is y_{n_2}, then we first need to prove (4.1) and (4.2), which in turn establish Theorem 4.3. Similarly as in Theorem 4.2, we have that

$$c_{y_{n_1},2} = c_{y_{n_1},1} + p_2,$$

$$b_{y_{n_1+1},1} = c_{y_{n_1},1},$$

$$c_{y_{n_1+1},1} = b_{y_{n_1+1},1} + p_2 = c_{y_{n_1},1} + p_2 = c_{y_{n_1},2}.$$

It means that y_{n_1+1} need not wait to be processed in Stage 2 as after y_{n_1}, jobs in J_1 will have already finished. So, next, the rest jobs of J_2 will be continuously processed. Furthermore, in Stage 1 and 2, the processing times of J are the same. Thus, the rest of the jobs need not wait either to be processed in Stage 2. Now, since y_{n_2} is the last job of S, then $c_{y_{n_2},1} = C_{M_{1,1}}$. Therefore, we have that $c_{y_{n_2},2} = C_{M_{1,1}} + p_2$, and since $\frac{p_1}{2} \leq p_2 < p_1$, then $c_{y_{n_2},2} = c_{x_{n_1},1} + 2p_2 \geq c_{x_{n_1},1} + p_1$. It follows that (4.8) and (4.9) hold. Thus schedule S is optimal (Figure 4.5). □

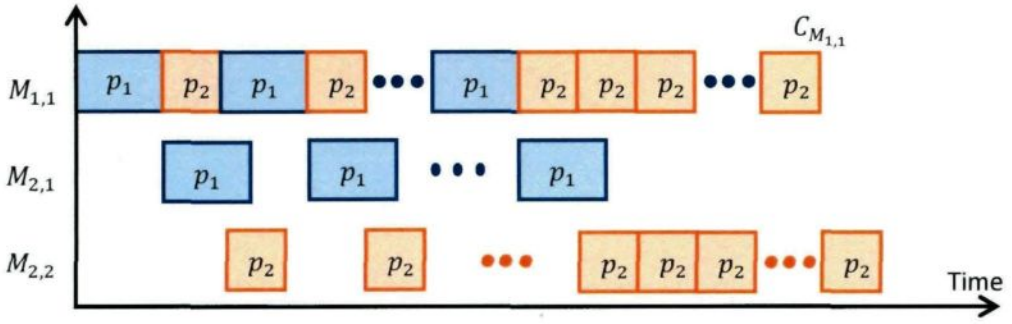


Figure 4.5: Size of J_2 larger than that of J_1

Based on Theorem 4.2 and Theorem 4.3, we may conclude that Case 2.1 is solvable in polynomial time with an optimal makespan as follows:

$$C_{max} = C_{M_{1,1}} + p_2.$$

The following discusses the case in which the processing times of two subsets are different.

Case 2.2 $\frac{p_1}{k+1} < p_2 < \frac{p_1}{k}$

First, let us point out that If *k=1,* then Case 2.2 is the same as Case 2.1. So, in what follows we assume that $k > 1$. In this case, we must first reconstruct the subset that has the shorter processing time. The method is as follows:

Compose k original jobs as a group and regard it as a new subset $j'_{2,i}$, where k is the smallest integer, which makes $k \times p_2 > p_1$, and the jobs' processing time of $j'_{2,i}$ is $k \times p_2 = p'_2$. Thus, subset j'_2 includes $\left\lfloor \frac{n_2}{k} \right\rfloor$ new jobs and $n_2 \bmod k$ original jobs, that is to say

$$J_2 = \left\{ j'_{2,1}, j'_{2,2}, \cdots j'_{2,\left\lfloor \frac{n_2}{k} \right\rfloor}, j_{2,1}, j_{2,2}, \cdots j_{2,(n_2 \bmod k)} \right\}.$$

Now, if $\left\lfloor \frac{n_2}{k} \right\rfloor < n_1$, we first get $n_1 - \left\lfloor \frac{n_2}{k} \right\rfloor$ jobs out of N_1 and put them at the top of the schedule. Then, we use $\left\lfloor \frac{n_2}{k} \right\rfloor$ jobs of J_1 and $\left\lfloor \frac{n_2}{k} \right\rfloor$ new jobs of J_2 to make $\left\lfloor \frac{n_2}{k} \right\rfloor$ pairs of jobs (put the jobs of J_1 in front of the new jobs of J_2) and then put them into the schedule. At last, we put $j \bmod n$ jobs originally from J_2 at the end of the schedule.

On the other hand, if $\left\lfloor \frac{n_2}{k} \right\rfloor \geq n_1$, we use n_1 jobs from J_1 and n_1 new jobs from J_2 to make n_1 pairs of jobs (put the jobs of J_1 in front of the new jobs of J_2), and put them at the front of the schedule. Then, we put the rest of the original jobs of J_2 at the end of the schedule. Therefore, Case (2.2) is reduced to Case (2.1) already studied above.

Both of these two schedules, pictured by Figure 4.6 and Figure 4.7, generate the minimum makespan $C_{max} = C_1 + p_2$.

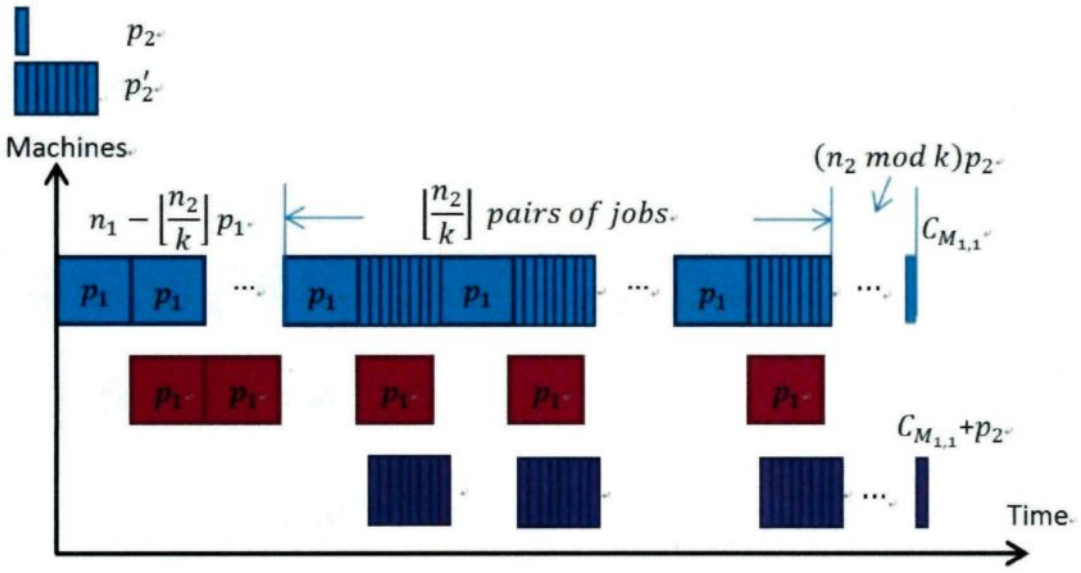


Figure 4.6: $\left\lfloor \frac{n_2}{k} \right\rfloor < n_1$

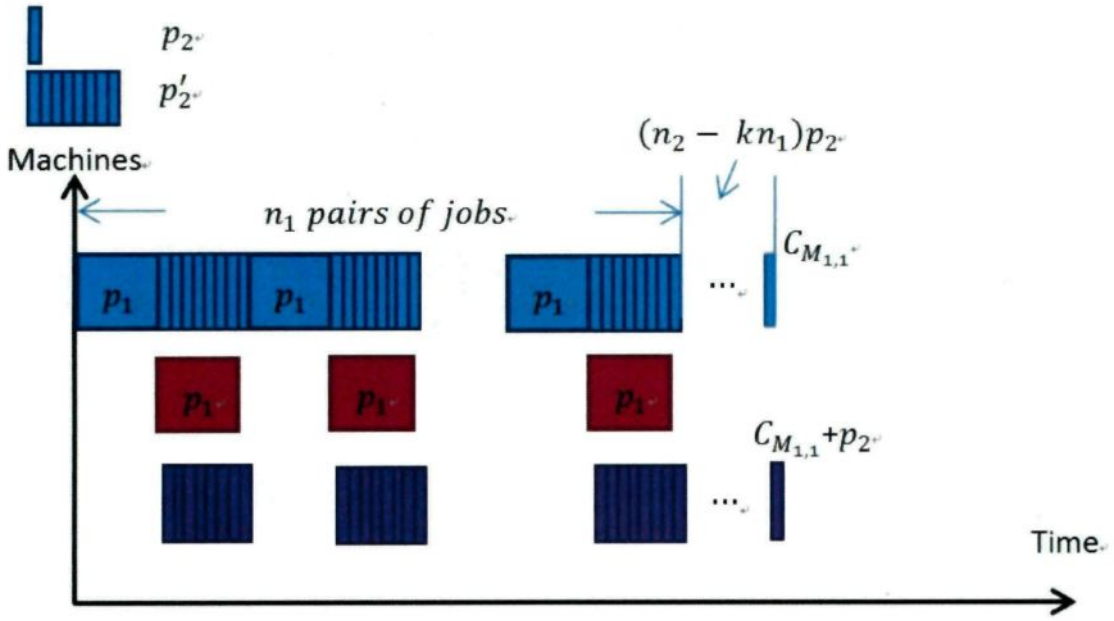


Figure 4.7: $\left\lfloor \frac{n_2}{k} \right\rfloor \geq n_1$

Combining Case 2.1 with Case 2.2, we derive a polynomial algorithm in the case of constant processing time as follows. Without loss of generality, we may suppose that $p_1 \geq p_2$.

Step 1: If $\frac{p_1}{2} \leq p_2 < p_1$, then process J_1 and J_2 as in Case 2.1.

Step 2: If $\frac{p_1}{k+1} < p_2 < \frac{p_1}{k}$, $k \geq 2$, reconstruct J_2 as follows:

$$J_2 = \left\{j'_{2,1}, j'_{2,2}, \dots j'_{2,\left\lfloor\frac{n_2}{k}\right\rfloor}, j_{2,1}, j_{2,2}, \dots j_{2,(n_2 \bmod k)}\right\}.$$

Step 3: Schedule $\left\{j'_{2,1}, j'_{2,2}, \dots, j'_{2,\left\lfloor\frac{n_2}{k}\right\rfloor}\right\}$ and J_1 as in Case 2.1, and then, put the rest of the jobs of J_2 at the end of the schedule generated in Step 2.

In the previous section, we considered the case with constant processing time. Now, we turn our attention into some more general processing times.

4.2.3 Third special case: Large processing times in Stage 1

In this case, for each subset, the minimal job processing time in Stage 1 is larger than the maximal processing time in Stage 2. It then follows that no job will wait to be executed in Stage 2. Next, based on Lemma 4.1, we provide the algorithm for this case though Theorem 4.1. Let us first define following notations.

- If $J_2 = \emptyset$, P is the optimal schedule of problem, and its makespan is C_P.
- If $J_1 = \emptyset$, Q is the optimal schedule of problem, and its makespan is C_Q.
- C_{J_1,M_1} is the total processing times of J_1 in Stage 1.
- C_{J_2,M_1} is the total processing times of J_2 in Stage 1.

Without loss of generality, we suppose that $C_P \geq C_Q$. Let us first prove then following result.

Lemma 4.1: Let S be a schedule of $J = \{J_1, J_2\}$ in which the jobs in J_1 and J_2 are processed alternately in Stage 1. Then, there exists another schedule S', where the jobs of J_1 and J_2 are continuously processed in Stage 1. Moreover, the makespan of S' is not worse than the makespan of S.

Proof: For any subset, the maximum processing time in Stage 2 is always smaller than the minimum processing time in Stage 1. So, we know that any job need not wait when it is ready to enter Stage 2. This means that the makespan of a subset equals its total processing time in Stage 1 plus the processing time of its last job in S in Stage 2. Without loss of generality, we assume that the last job in S belongs to J_2 and one job of J_1 is processed first, and in S, the jobs from the two subsets are not continuously processed. Let us now consider another schedule, say S', in which J_1 is processed before $J_{2,}$. Before proceeding further, let us introduce the following notations in schedule S:

- The start time to process J_1 is $T_{1,1} = 0$.
- The makespan of J_1 is C_1.
- The start time to process J_2 is $T_{2,1}$.
- The end of processing time of J_2 in Stage 1 is $C_{2,1}$.
- The makespan of J_2 is C_2.
- The makespan of S is $C_{max} = max\{C_1, C_2\}$, see Figure 4.8.

Similarly, we introduce the following notations in schedule S':

- The start time to process J_1 is $T'_{1,1} = 0$.

- The makespan of J_1 is C_1'.
- The start time to process J_2 is $T_{2,1}'$.
- The end of the processing time of J_2 in Stage 1 is $C_{2,1}'$.
- The makespan of J_2 is C_2'.
- The makespan of S', $C_{max}' = max\{C_1', C_2'\}$, see Figure 4.9.

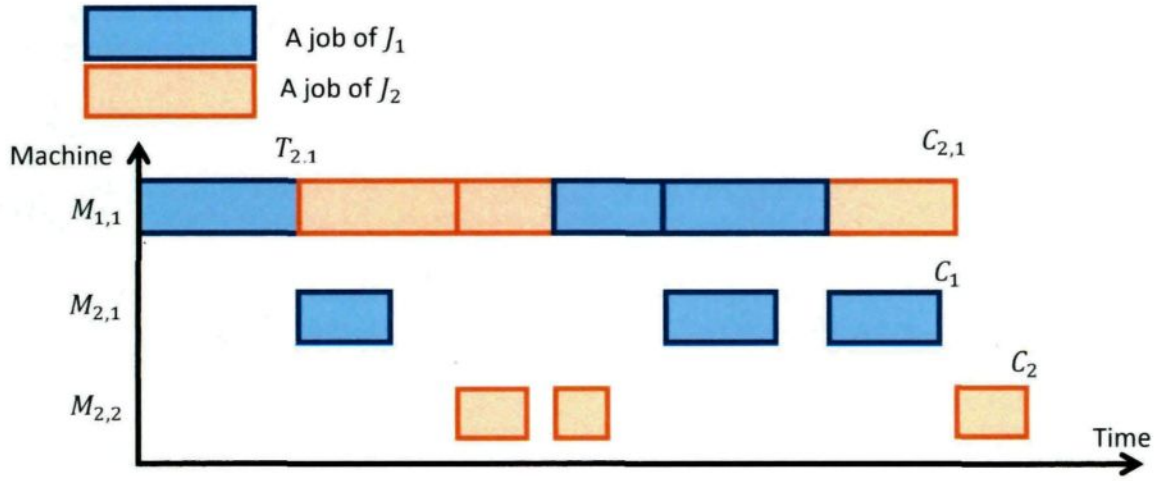


Figure 4.8: Crossed scheduling for the jobs of J_1 and J_2

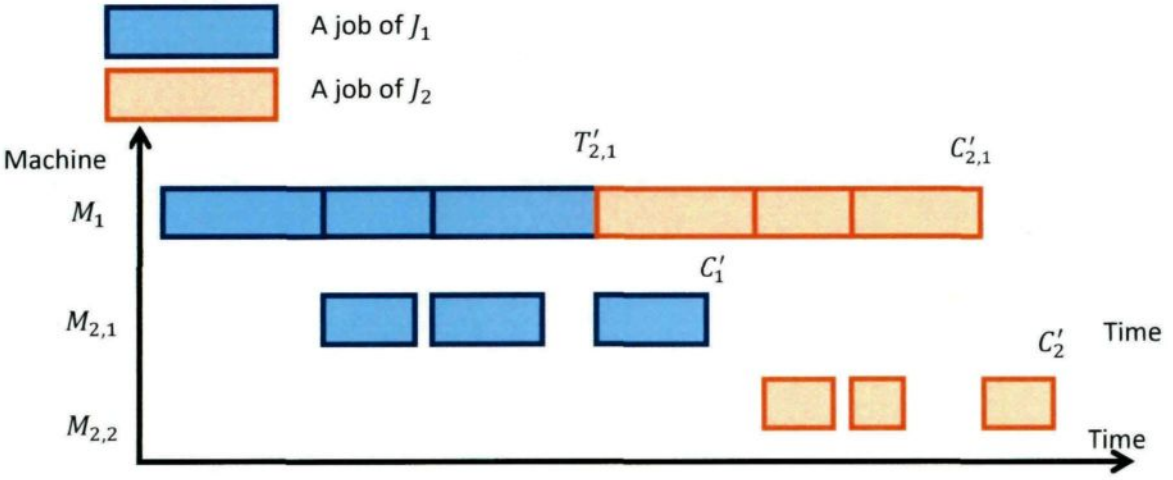


Figure 4.9: J_1 is processed before J_2.

Since there is just one machine in Stage 1, and that all jobs must first pass through it, then we get that $C_{2,2} = C'_{2,2}$. The sequencings of the jobs are the same for both schedules, and there is no waiting time when the last job of J_2 enters Stage 2. Therefore, we may deduce that $C_2 = C'_2$. However, in S, J_1 is not continuously processed. This leads to $C_1 > C'_1$. Let us summarize what we have got so far:

$$\left.\begin{array}{l} C_{max} = max\{C_1, C_2\} \\ C'_{max} = max\{C'_1, C'_2\} \\ C_1 > C'_1 \\ C_2 = C'_2 \end{array}\right\} \Rightarrow C'_{max} \leq C_{max}$$

It then follows that $S'$is not worse than S. Therefore, the result is established and this ends the proof. □

Now we are ready to state the following result.

Theorem 4.4: if the minimal processing time in Stage 1 is larger than the maximal processing time in Stage 2, then the value of the optimal makespan $of\ F3|M_1 \rightarrow M_2; M_1 \rightarrow M_3|C_{max}$ problem is $\min\left[max\left(C_P, C_{J_1,M_1} + C_Q\right),\ C_{J_2,M_1} + C_P\right]$. Furthermore, we have that

1) $C_{max} = max\left(C_P, C_{J_1,M_1} + C_Q\right)$ if, and only if, the optimum schedule is $S = P + Q$, where $P + Q$ means that, first, schedule J_1 and J_2 as P and Q, and then, put J_2 before J_1.
2) $C_{max} = C_{J_2,M_1} + C_P$ if, and only if, the optimum schedule is $S = Q + P$, where $Q + P$ means that that, first, schedule J_1 and J_2 as P and Q, and then, put J_1 before J_2.

Proof: From Lemma 4.1, there are only two cases that we need to distinguish: either J_1 precedes J_2 or J_2 precedes J_1. So, in any case, $F3|M_1 \rightarrow M_2; M_1 \rightarrow M_3|C_{max}$ is reduced

to $F2|M_1 \rightarrow M_2|C_{max}$, and can be scheduled by Johnson's rule. Let us recall the following:

- $C_P \geq C_Q$.
- P is the optimal schedule of J_1, and its makespan is C_P.
- Q is the optimal schedule of J_2, and its makespan is C_Q.

We start with the sufficient condition. If J_1 precedes J_2, then the optimal schedule of J is $P + Q$ and $C_{J_1} = C_P$ and $C_{J_2} = C_{J_1,M_1} + C_Q$. Since $C_{max} = \{C_{J_1}, C_{J_2}\}$, then $C_{max} = max(C_P, C_{J_1,M_1} + C_Q)$. Now, if J_2 precedes J_1, then the optimal schedule of J is $Q + P$, and $C_{J_1} = C_{J_2,M_1} + C_P$ with $C_{J_2} = C_Q$. Since $C_{max} = \{C_{J_1}, C_{J_2}\}$ and $C_P \geq C_Q$, then we have $C_{max} = C_{J_2,M_1} + C_P$. Thus, the result follows.

Let us now prove the necessary condition, and proceed as follows:

(1) If $C_{max} = max(C_P, C_{J_1,M_1} + C_Q)$, whether C_P and $C_{J_1,M_1} + C_Q$, which one is bigger, in order to make $C_{J_1} = C_P$ and $C_{J_2} = C_Q$, we always must schedule J_1, J_2 as in P and Q, and obviously J_1 must precede J_2. So the optimal schedule of J is $P + Q$

(2) If $C_{max} = C_{J_2,M_1} + C_P$, then similarly as above, we schedule J_1 and J_2 as in P and Q, respectively. Obviously J_2 must precede J_1. It follows that the optimal solution is $Q + P$. The necessary condition is thus established.

This ends the proof as both sufficient and necessary conditions are now established. □

Therefore, for the case of large processing times in Stage 1, the following is the algorithm we might derive:

Step 1: Schedule J_1 and J_2 as in the standard flow-shop by Johnson's rule.

Step 2: If $C_{max} = max(C_P, C_{J_1,M_1} + C_Q)$, the optimal solution is $S = P + Q$.

Step 3: Else (we have $C_{max} = C_{J_2,M_1} + C_P$), the optimal solution is $S = Q + P$.

4.2.4 Fourth special case: Large processing times in Stage 2

Without loss of generality, we suppose that $C_P \geq C_Q$. Let us first proceed with the following result.

Lemma 4.2: Les P be an optimal solution of J_1, and C_P the corresponding makespan. Let also a, b and c are three different jobs of J. If $p_{a,2} \geq p_{b,1} + p_{c,1}$, then inserting a job of J_2 into P will not change C_P.

Proof: Let x, y be two jobs of J_1 and assume they are adjacent in P such that x precedes y. The start time of y in stage 2 is

$$b_{y,2} = \max[(b_{x,1} + p_{x,1} + p_{x,2}), (b_{x,1} + p_{x,1} + p_{y,1})].$$

Since $p_{a,2} \geq p_{b,1} + p_{c,1}$, then $p_{x,2} > p_{y,1}$. So, we have that

$$b_{y,2} = b_{x,1} + p_{x,1} + p_{x,2},$$

$$c_{y,2} = b_{y,2} + p_{y,2} = b_{x,1} + p_{x,1} + p_{x,2} + p_{y,2}.$$

Now, let us insert job j of J_2, say j, between x and y. The start time of y in stage 2 will be delayed and the new one will be

$$b'_{y,2} = \max[(b_{x,1} + p_{x,1} + p_{x,2}), (b_{x,1} + p_{x,1} + p_{j,1} + p_{y,1})].$$

Since $p_{a,2} \geq p_{b,1} + p_{c,1}$, then we have that $p_{x,2} > p_{j,1} + p_{y,1}$. So, $b'_{y,2} = b_{x,1} + p_{x,1} + p_{x,2}$, and $c'_{y,2} = b'_{y,2} + p_{y,2} = b_{x,1} + p_{x,1} + p_{x,2} + p_{y,2} = c_{y,2}$. Therefore, inserting job j into P does not change C_P (see Figure 4.10 for an illustration). Thus, the result is established. □

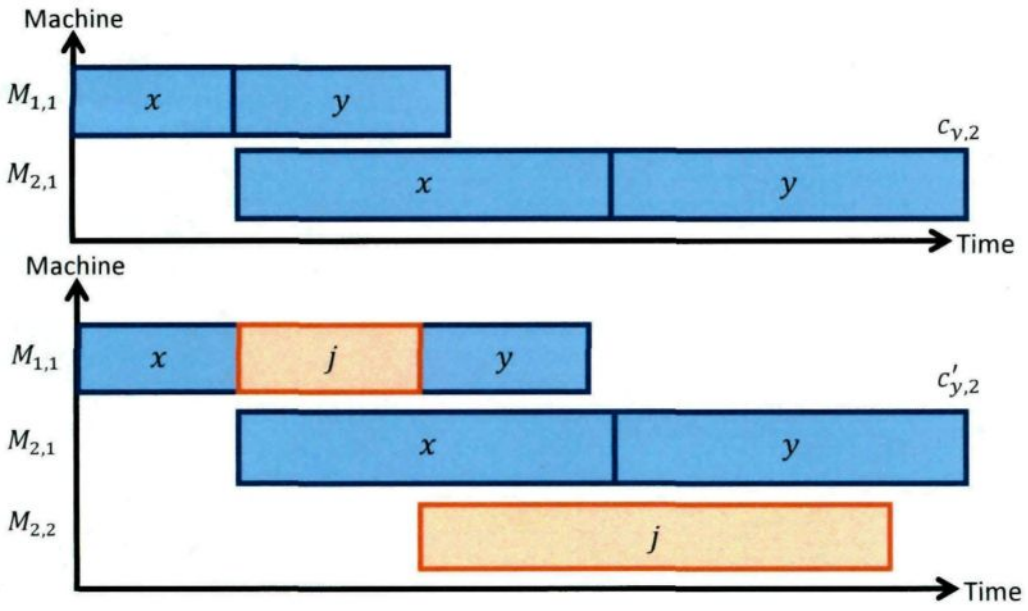


Figure 4.10: No change on the makespan after inserting job j

Now we are ready to state the following result. Let us first introduce the following notation.

- $P = \{\lambda(1), \lambda(2), \dots \lambda(n_1)\}$ is the optimal solution of J_1 (obtained by Johnson's rule), with a makepan C_P, and $\lambda(i)$ denotes the job processed at position i.
- $Q = \{\mu(1), \mu(2), \dots \mu(n_2)\}$ is the optimal solution of J_2 (obtained by Johnson's rule), with a makespan C_Q, and $\mu(i)$ denotes the job processed at position i.

Theorem 4.5: Let $n_1 \geq n_2$, and a, b and c be three different jobs of J such that $p_{a,2} \geq p_{b,1} + p_{c,1}$, a, b, c in J. An optimal solution is achieved by sequence $S1$ or $S2$, as defined below, depending on the smallest value of their corresponding makespan.

$$S1 = (\lambda(1), \mu(1), \lambda(2), \mu(2), \dots, \lambda(n_2), \mu(n_2), \lambda(n_1 - n_2 + 1), \dots, \lambda(n_1)),$$

or

$$S2 = (\mu(1), \lambda(1), \mu(2), \lambda(2), \dots, \mu(n_2), \lambda(n_2), \lambda(n_1 - n_2 + 1), \dots, \lambda(n_1)).$$

Proof: Let us consider two cases: either $\lambda(1)$ or $\mu(1)$ is processed at the first position of S. Let us start with the first case. We insert the jobs of Q between every two jobs of$P$, one by one, and create a schedule $S = (\lambda(1), \mu(1), \lambda(2), \mu(2), \dots, \lambda(n_2), \mu(n_2), \lambda(n_1 - n_2 + 1), \dots, \lambda(n_1))$, see Figure 4.11.

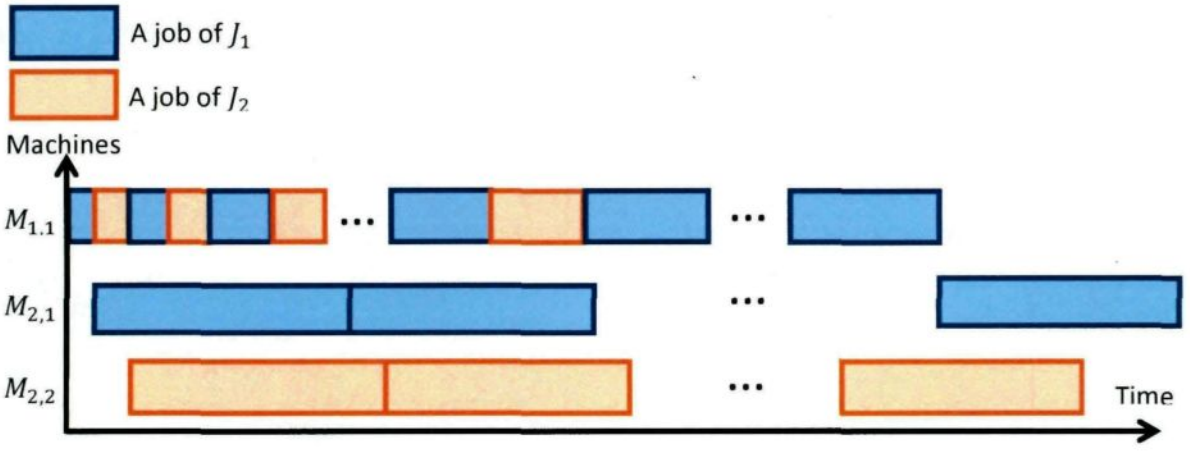


Figure 4.11: One job of another subset between two jobs of J_1 or two jobs of J_2

Note that since between two jobs of J_1 or two jobs of J_2, there is just one job of another subset. From Lemma 4.2, it follows that

$$C_{J_1} = C_P, C_{J_2} = p_{\lambda(1),1} + C_Q.$$

As $C_{max} = max(C_{J_1}, C_{J_2})$, then we have that $C_{max} = max\left(C_P, p_{\lambda(1),1} + C_Q\right)$. Therefore, any change of permutation of jobs will increase C_P and C_Q, and then C_{J_1} and C_{J_2} will be increased too. Therefore, S is the optimal schedule.

On the other hand, if $\mu(1)$ is at the first position of the job sequence of J, and we insert the jobs of P between every two jobs of Q, one by one, then

$$S = (\mu(1), \lambda(1), \mu(2), \lambda(2), \dots, \mu(n_2), \lambda(n_2), \lambda(n_1 - n_2 + 1), \dots, \lambda(n_1)),$$

and

$$C_{J_1} = p_{\mu(1),1} + C_P,\ C_{J_2} = C_Q.$$

As $C_{max} = max(C_{J_1}, C_{J_2})$, then we $C_{max} = max\left(p_{\mu(1),1} + C_P, C_Q\right)$. So, we just need to compare the two values of the makespan in order to derive the optimal solution. □

The following is the algorithm we might derive from the above discussion:

Step 1: Scheduling J_1 and J_2 according to Johnson's rule to generate, respectively, the sequences $P = \{\lambda(1), \lambda(2), \dots, \lambda(n_1)\}$ and $Q = \{\mu(1), \mu(2), \dots, \mu(n_2)\}$.

Step 2: The final solution is one of the following with a smaller makespan:

a. $S1 = (\lambda(1), \mu(1), \lambda(2), \mu(2), \dots, \lambda(n_2), \mu(n_2), \lambda(n_1 - n_2 + 1), \dots, \lambda(n_1))$.

b. $S2 = \left(\mu(1), \lambda(1), \mu(2), \lambda(2), \dots, \mu(n_2), \lambda(n_2), \lambda(n_1 - n_2 + 1), \dots, \lambda(n_1)\right)$.

Chapter 5

Heuristic approach

5.1 Introduction

In this chapter, we present two heuristic algorithms to solve the general version of the problem we are considering in this dissertation. They start from scheduling the jobs by an existed heuristic of standard two-stage flow-shop. Then, according to some rules, it expands from the current iteration to the next iteration the partial solution, until reaching the terminal condition. Even the time complexity of such approach is usually attractive, experimental studies have shown that the quality of the solution may be poor.

5.2 Heuristic algorithms of FSP

Many heuristics of FSP are based on Johnson's rule. Next, we will introduce some of the more popular heuristics used to solve FSP. In order to obtain the optimal solution of FSP, we need to use the optimal algorithm, as Johnson's rule. Unfortunately, except standard two-stage FSP, the vast majority of FSP are not in P-class, their exact optimal algorithm's time complexity is often too large and grows exponentially with the scale of the problem. It was proved that Flow-shop problems with three machines are already NP-Hard [Garey and Johnson, 1979]. Therefore, the use of heuristic approach is well justified.

5.2.1 CDS heuristic algorithm

By extending Johnson's rule, we can construct our first heuristic, CDS (Campbell-Dudek-Smith) heuristic algorithm [Campbell, Dudek, and Smith, 1970]. It is used to solve an n-job, m-machine Flow-shop problem. This algorithm is recognized as the most efficient and robust among the existing heuristic algorithms, and it has been the standard for comparison in many studies.

First, get CDS heuristic group m machines so as to get a set of two-machine Flow-shop problems, which has m-1 factors (see Table 5.1). Then, use Johnson's rule to get an m-1 optimal schedule for those m-1 problems. Finally, choose the best one as an approximate optimal solution of the original problem.

Step i	Simulation of two-machine problems		Combined processing time	
	Group 1	Group 2	t'_{i1}	t'_{i2}
1	1	M	t_{i1}	t_{im}
2	1,2	m,m-1	$t_{i1} + t_{i2}$	$t_{im} + t_{i,m-1}$
3	1,2,3	m,m-1,m-2	$t_{i1} + t_{i2} + t_{i3}$	$t_{im} + t_{i,m-1} + t_{i,m-2}$
...	...	...	...	....
m-1	1,2,...,m-1	m,m-1,...,2	$t_{i1} + t_{i2} + \cdots + t_{i,m-1}$	$t_{im} + t_{i,m-1} + \cdots + t_{i2}$

Table 5.1: Set of two-machine flow shop problems

In Step k, the combined processing time is defined as:

$$t'_{i1} = \sum_{j=1}^{k} t_{ij},$$

$$t'_{i2} = \sum_{j=1}^{k} t_{i,m-j+1}.$$

5.2.2 Palmer heuristic algorithm

The Palmer heuristic algorithm [Palmer, 1965] sorts the jobs by slop order index. According to the order of the machine, the job whose processing time tends to increase is given greater weight number; otherwise, the job whose processing time tends to reduce is given smaller weight number. The slop order index of job i is defined as

$$s_i = \sum_{j=1}^{m}(2j - m - 1)t_{ij}, i = 1,2, \dots, m.$$

According to the non-increasing order of s_i, we can construct a schedule of jobs as

$$S_{i_1} \geq S_{i_2} \geq \cdots \geq S_{i_n}.$$

5.2.3 RA heuristic algorithm

Dannenbring combined the Palmer heuristic and CDS heuristic, and proposed a rapid access (RA) heuristic algorithm [Dannenbring, 1977]. The RA heuristic does not need to solve m-1 two-machine problems, but just needs to solve one simulated problem by Johnson's rule. Combined processing time is defined as follows:

$$t'_{i1} = \sum_{j=1}^{k} w_{j1}t_{ij}.$$

$$t'_{i2} = \sum_{j=1}^{k} w_{j2}t_{i,j}.$$

The weights are defined as follows:

$$W_1 = \{w_{j1} | j = 1,2, \dots m\} = \{m, m-1, \dots, 2, 1\}.$$

$$W_2 = \{w_{j2} | j = 1,2, \dots m\} = \{1,2, \dots, m-1, m\}.$$

5.2.4 NEH heuristic algorithm

The NEH heuristic algorithm [Nawaz, Enscore, and Ham, 1983] supposes that the priority of the job that has the longer total processing time should be greater than that

of the job that has a shorter total processing time. That means that we need to calculate each job's total processing time, and then sort all jobs in descending order according to the total processing time. Doing this produces the optimal schedule for the first two jobs, and we then put the remaining jobs into the job queue that has been scheduled until all the jobs are inserted into the queue. Ultimately, we get a schedule for the problem. NEH is a relatively good performance heuristic; it is often used to optimize the initial solution of meta-heuristic. The steps of this algorithm are as follows:

Step 1: Sort n jobs in descending order by total processing times.

Step 2: Scheduling the first two jobs, and let the makespan be minimal.

Step 3: Insert the remaining jobs, always keeping the makespan minimal.

5.2.5 Gupta heuristic algorithm

Gupta proposed a heuristic similar to the Palmer heuristic [Gupta, 1971]. At the beginning, the algorithm calculates the parameter of each job:

$$S(i) = \frac{C}{\min_{1 \le j \le m-1}(t_{ij}+t_{i(j+1)})}.$$

Here, if $t_{im} \le t_{i1}$, then $C = -1$, else $C = 1$. After that, we sort the jobs by the order of increasing of parameters and a schedule is thus derived for the problem.

5.3 Heuristic design for $F3|M_1 \to M_2; M_1 \to M_3|C_{max}$

In the literature, there is not much discussion regarding the models discussed in this dissertation. Based on Johnson's rule and the NEH heuristic, in this section, we will construct two heuristic algorithms for the model we are studying and test their performance.

5.3.1 A heuristic based on Johnson's rule

Before proceeding, let us clarify the following points:

- When $J_2 = \emptyset$, the corresponding problem P is solvable by the Johnson rule. The generated makespan is denoted by C_P.
- When $J_1 = \emptyset$, the corresponding problem Q is solvable by the Johnson rule. The generated makespan is denoted by C_Q.
- C_{J_1,M_1} is the total processing times of J_1 in Stage 1.
- C_{J_2,M_1} is the total processing times of J_2 in Stage 1.

The reason we say that this heuristic method is based on Johnson's rule is that we need to use the rule to get C_P and C_Q. Here C_P has two meanings. First, it is the minimal makespan of J_1, if J_1 is executed in a two-stage Flow-shop system; second, in the model of Two-Stage Flow-shop with two parallel machines in Stage two, it is the minimum total completion time of J_1. Similarly, C_Q is the minimal makespan of J_2 and it is also the minimum total completion time of J_2. Without loss of generality, we suppose that $C_Q \leq C_P < C_{J_1,M_1} + C_Q$.

According to Section 4.3.3, we can consider that C_P is a lower bound of the makespan, while $C_{J_1,M_1} + C_Q$ is the upper bound. So

$$C_P \leq C_{max} \leq C_{J_1,M_1} + C_Q.$$

We assume that $f(S)$ and $g(S)$ are the functions of the total completion time of J_1 and J_2. The focus of the curves of these two functions is the minimal makespan of the model (Figure 5.1), the image of $f(S)$ through point (S, C_P) and the image of $g(S)$ through point $(S, C_{J_1,M_1} + C_Q)$, here $S = P + Q$.

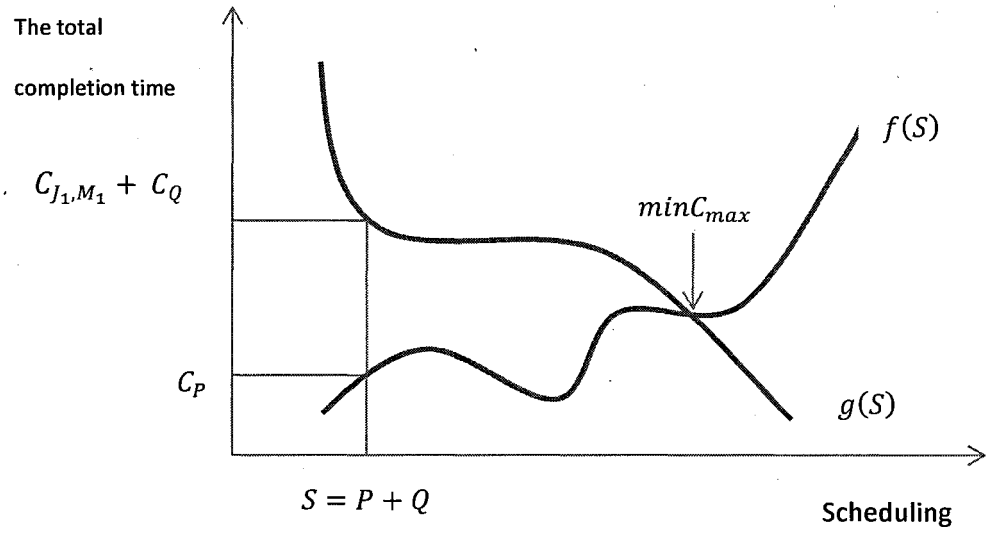


Figure 5.1: Minimal value of two functions

Based on this discussion, the basic principle of building the heuristic algorithm is that we try to get a new schedule by keeping changing the order of jobs and letting $|f(S) - g(S)|$ get as small as possible. Let us now give the specific steps of the heuristic.

Assume that there are m jobs in J_1 and n jobs in J_2; the initial scheduling is $S = P + Q$. In this scheduling, there are $m - 1$ positions between every two jobs of J_1 and $n - 1$ positions between every two jobs of J_2.

Step 1: Using the scheduling $S = P + Q$ as the initial schedule. Upper Bound= $g(S) = C_{J_1,M_1} + C_Q$, Lower Bound= $f(S) = C_P$ and $i = 1$

Step 2: Insert job j of J_2 into the $m - i + (j - 1)$ position, $j = (1,2 \ldots \min(i, n))$, and if $m - i + (j - 1) < 1$, then we put this job before all the jobs of N_1. So the new solution is S'. Obviously, $f(S') \geq f(S)$ and $g(S') \leq g(S)$.

Step 3: If $S' = S$, then return $max(f(S'), g(S'))$.

Step 4: If $f(S') \geq g(S')$ return $C_{max} = \min\left(\max(f(S'), g(S')), \max(f(S), g(S))\right)$;

If $C_{max} = max(f(S'), g(S'))$, then the final schedule is S', else the final schedule is S.

Step 5: If $f(S') < g(S')$, let $S = S'$, $i = i + 1$ and go back to step 2.

Let us now concentrate on the convergence of the heuristic. When $i \geq m + (n - 1)$, all jobs of J_2 precede those of J_1. As in the process of the algorithm, the order of jobs of each subset is never changed. So with the increase of i, it will no longer generate new scheduling. Therefore, the algorithm will end at the third step.

The following is a running example to illustrate the above heuristic algorithm.

Example 5.1: Table 5.2 presents the jobs of J_1 and J_2.

	$J_{1,1}$	$J_{1,2}$	$J_{1,3}$	$J_{1,4}$
M_1	8	9	3	4
$M_{2,1}$	10	6	6	9

	$J_{2,1}$	$J_{2,2}$	$J_{2,3}$
M_1	7	3	8
$M_{2,2}$	9	5	1

Table 5.2: Processing times of jobs for Example 5.1

We use Johnson's rule to schedule the two subsets as shown in Table 5.3.

1	2	3	4	Makespan	C_{J_1,M_1}
$J_{1,3}$	$J_{1,4}$	$J_{1,1}$	$J_{1,2}$	34	24

1	2	3	Makespan
$J_{2,2}$	$J_{2,1}$	$J_{2,3}$	20

Table 5.3: Schedule two subsets by Johnson's rule

Set the initial schedule:

1	2	3	4	5	6	7
$J_{1,3}$	$J_{1,4}$	$J_{1,1}$	$J_{1,2}$	$J_{2,2}$	$J_{2,1}$	$J_{2,3}$

	J_1	J_2	Makespan
Total Time	34	44	44

Insert $J_{2,2}$ between $J_{1,1}$ and $J_{1,2}$

1	2	3	4	5	6	7
$J_{1,3}$	$J_{1,4}$	$J_{1,1}$	$J_{2,2}$	$J_{1,2}$	$J_{2,1}$	$J_{2,3}$

	J_1	J_2	Makespan
Total Time	34	44	44

Continue to adjust the schedule

1	2	3	4	5	6	7
$J_{1,3}$	$J_{1,4}$	$J_{2,2}$	$J_{1,1}$	$J_{2,1}$	$J_{1,2}$	$J_{2,3}$

	N_1	N_2	Makespan
Total Time	40	43	43

1	2	3	4	5	6	7
$J_{1,3}$	$J_{2,2}$	$J_{1,4}$	$J_{2,1}$	$J_{1,1}$	$J_{2,3}$	$J_{1,2}$

	N_1	N_2	Makespan
Total Time	48	34	48

The algorithm terminates, and returns the solution $S = \{j_{1,3}, j_{1,4}, j_{2,2}, j_{1,1}, j_{2,1}, j_{1,2}, j_{2,3}\}$, and the corresponding makespan is 43.

5.3.2 A heuristic based on NEH

We also gave a detailed description of another heuristic for standard FSP; this is NEH, which is a relatively efficient heuristic. It is often used to optimize the initial solution of a meta-heuristic algorithm.

As we have observed so far, it is not difficult to find that if we want to minimize makespan, we must let the idle time of the machines be as short as possible. This makes it easy to think that we should let the jobs of two subsets, as much as possible, stagger through stage 1.

Here, we just make few changes to the NEH; the new version of algorithm can be described as follows:

Step 1: Sort jobs of two subsets by descending order of total processing time.

Step 2: Find out the optimal scheduling of the first job of J_1 and first job of J_2.

Step 3: Suppose that the total processing time of the first job of J_1 is larger than that of J_2. Insert the second job of J_1 into the job sequence, and keep the makespan minimal, and then insert the second job of J_2 into the job sequence, and keep also the makespan minimal.

Step 4: The remaining jobs of the two subsets are alternately inserted into the job sequence.

Example 5.2: Let us consider the following instance as in Table 5.4 in which the jobs are already sorted.

	$J_{1,1}$	$J_{1,2}$	$J_{1,3}$
M_1	8	9	3
$M_{2,1}$	10	6	6

	$J_{2,1}$	$J_{2,2}$
M_1	7	3
$M_{2,2}$	9	5

Table 5.4: Processing times of jobs for Example 5.2

Now, we schedule the two jobs $j_{1,1}$ and $j_{2,1}$, which have the biggest total processing time of the two subsets.

$$S = (j_{1,1}, j_{2,1}).$$

We then insert the second biggest job of J_1 into the job sequence,

$$S = (j_{1,1}, j_{2,1}, j_{1,2});$$

and the next is the second biggest of J_2,

$$S = (j_{1,1}, j_{2,1}, j_{1,2}, j_{2,2}).$$

Insert the remaining work into the sequence,

$$S = (j_{1,1}, j_{2,1}, j_{1,2}, j_{2,2}, j_{1,3}).$$

The corresponding makespan is 36.

5.3.3 Experimental study

The simulation we have undertaken runs in Apple's iMac with Core i3 and 4GB memory and be programmed by Java. All the processing time of jobs are randomly generated.

Tables 5.5.1 to 5.5.3 show the simulation results of the two algorithms, and their comparison. In Table 5.5.1, 5.5.2, and 5.5.3, the processing times are random positive

integers drawn from [50,100], [20,100], and [10,100], respectively. The sizes of the job sets are 50, 100 and 200. In the program we have coded, we added appropriate to ensure that the special cases studied in Chapter 4 do not appear.

From Table 5.5.1 to Table 5.5.3, we observe that the heuristic based on Johnson's rule is slightly better than the heuristic based on NEH. With the growth of the problem size, the rate of increase of the time complexity of the heuristic based on NEH is much higher than that of the heuristic based on Johnson's rule. Let us also mention that most of the time, both algorithms find an optimal solution as the makespan generated is $\mathrm{LB} = C_{M_{1,1}} + \min_{\substack{1\le i\le n \\ j=1,2}}\{p_{j,i,2}\}$, which is a lower bound. Let us point out that the quality of the solution gets much better as the size of the input gets over 200 jobs.

For further verification of this observation, we conducted a separate experiment to determine which algorithm has a better performance. In this experiment, we carried out 100,000 calculations in each case, and the value of processing time was still a random positive integer. The experimental results show that the performance of the heuristic based on Johnson's rule in the smaller range of processing time is superior to a larger range. As the size of the job reaches 500, according to the computations of 400000 groups of data, the heuristic based on Johnson's rule always finds the optimal solution.

Processing time 50-100		Heuristic based on Johnson's rule				Heuristic based on NEH			
The sizes of subsets $(J_1 \times J_2)$	Average of lower bound(ALB)	Number of optimal solutions	Average of the results(AR)	AR/ALB	Average of the running time	The number of optimal solutions	Average of the results(AR)	AR/ALB	Average of the running time
35×15	3773.662	986	3773.699	1.000009804799688	0.025	976	3773.712	1.0000132497293082	1.375
25×25	3779.885	994	3779.894	1.0000023810248195	0.025	988	3779.906	1.0000055557245788	1.38
70×30	7502.456	998	7502.458	1.0000002665793708	0.095	994	7502.462	1.0000007997381124	9.786
60×40	7504.04	999	7504.041	1.0000001332615498	0.105	997	7504.043	1.0000003997846494	9.771
50×50	7497.249	999	7497.25	1.000000133382258	0.1	995	7497.256	1.0000009336758058	9.883
150×50	14941.497	1000	14941.497	1.0	0.441	999	14941.498	1.000000066927698	80.692

Table 5.5.1: Simulation of both heuristics where processing times are in [50, 100]

Processing time 20-100		Heuristic based on Johnson's rule				Heuristic based on NEH			
The sizes of subsets $(J_1 \times J_2)$	Average of lower bound(ALB)	Number of optimal solutions	Average of the results(AR)	AR/ALB	Average of the running time	The number of optimal solutions	Average of the results(AR)	AR/ALB	Average of the running time
35×15	3000.9	991	3000.918	1.0000059982005398	0.04	984	3000.937	1.000012329634443	1.335
25×25	2992.507	991	2992.528	1.0000070175274443	0.02	987	2992.533	1.000008688367312	1.381
70×30	5982.304	998	5982.306	1.0000003343193524	0.095	994	5982.313	1.0000015044370865	9.957
60×40	5969.368	999	5969.369	1.000000167521922	0.105	993	5969.38	1.000002010263063	9.908
50×50	5973.376	997	5973.383	1.0000011718666295	0.13	995	5973.383	1.0000011718666295	9.906
150×50	11920.188	1000	11920.188	1.0	0.355	998	11920.191	1.0000000251673883	76.942

Table 5.5.2: Simulation of both heuristics where processing times are in [20, 100]

Processing time 10-100		Heuristic based on Johnson's rule				Heuristic based on NEH			
The sizes of subsets $(N_1 \times N_2)$	Average of lower bound(ALB)	Number of optimal solutions	Average of the results(AR)	AR/ALB	Average of the running time	The number of optimal solutions	Average of the results(AR)	AR/ALB	Average of the running time
35×15	2736.731	990	2736.761	1.0000109619834758	0.035	2736.783	981	1.000019000771358	1.335
25×25	2736.412	988	2736.447	1.0000127904716103	0.025	2736.438	989	1.0000095014931962	1.355
70×30	5451.699	996	5451.707	1.0000014674324464	0.16	5451.713	992	1.000002568006781	9.98
60×40	5463.257	997	5463.261	1.000000732163982	0.1	5463.264	995	1.0000012812869687	9.952
50×50	5472.099	997	5472.103	1.0000007309809271	0.07	5472.104	996	1.000000913726159	10.052
150×50	10895.569	1000	10895.569	1.0	0.421	10895.575	995	1.0000005506825758	77.613

Table 5.5.3: Simulation of both heuristics where processing times are in [10, 100]

Processing time is between 50-100	Heuristic based on Johnson's rule	Processing time is between 10-100	Heuristic based on Johnson's rule
The sizes of subsets $(N_1 \times N_2)$	Number of optimal solutions	The sizes of subsets $(N_1 \times N_2)$	Number of optimal solutions
75×25	99886	75×25	99707
50×50	99894	50×50	99712
180×20	100000	180×20	99830
120×80	100000	120×80	99949
400×100	100000	400×100	100000
300×200	100000	300×200	100000

Table 5.5.4: Simulation on the heuristic based on Johnson's rule

Chapter 6

Meta-heuristic approach

6.1 Introduction

As introduced in Section 3.6.2, meta-heuristic is a kind of iterative optimization algorithm process. It starts from one or several initial solutions, and evaluates the quality of a solution, based on the value of the objective function. Then, according to certain optimization strategies, it updates the solution as the initial solution for the next iteration, until fulfilling the termination conditions.

In this chapter, we study and use two meta-heuristic algorithms *viz.* Tabu Search (TS) and Particle Swarm Optimization (PSO). The goal is to improve PSO, through hybridization with TS, to make it more efficient for solving $F3|M_1 \rightarrow M_2; M_1 \rightarrow M_3|C_{max}$.

6.2 Tabu Search (TS)

Glover created Tabu search (TS) in 1986 and formalized in 1989 [Glover, 1989] [Glover, 1990]. TS is a local search method which simulates the model of human memory, and uses a tabu table to shield the area that was just searched, to avoid circuitous searching, at the same time, through pardon some good solutions in tabu list, ensure the diversity of the search, so as to achieve global optimization. In this section, we will present the principles, characteristics and processes of TS.

6.2.1 Neighbourhood search

Neighbourhood search, also known as a hill-climbing algorithm that is based on greedy principle, is the basis of TS. Neighbourhood search continues to search until there is no better solution in the current neighbourhood. In topology and related areas of mathematics, a neighbourhood is one of the basic concepts in a topological space. Intuitively speaking, a neighbourhood of a point is a set containing the point where you can move that point some amount without leaving the set.

Consider the following optimization problem: $\min c(x): x \in X$. Here, the objective function $c(x)$ could be linear or non-linear; the solution space X is constructed by a finite number of discrete points in the n-dimensional real-value space. The process of neighbourhood search is to move from one solution to another solution. Parameter s expresses a move; $s(x)$ is the solution obtained after the move and $S(x)$ is the neighbourhood of current solution. The neighbourhood search algorithm can be simply described as follows.

Step 1: Choose an initial solution $x \in X$

Step 2: Choose the best solution $s(x)$ in the neighbourhood $S(x)$ of the current solution.

Step 3: If $c(s(x)) > c(x)$, move x to $s(x)$ and return to Step 2. Otherwise, the algorithm terminates.

Following is a simple example to illustrate the neighbourhood search.

Example 6.1: We are given a two-stage flow-shop problem, 4 jobs to execute, and we seek to minimize the makespan. The processing times of each job in the two centres are shown in Table 6.1.

Job	Centre 1	Centre 2
1	4	5
2	8	3
3	12	7
4	6	9

Table 6.1: Processing times of jobs for Example 6.1

We randomly assign an initial solution: $x = (3,4,2,1)$. This solution represents the processing sequence as 3→4→2→1. The makespan of this solution is $C_x = 36$.

We require that the neighbourhood of the current solution is constructed by the solutions that are obtained by exchanging the position of the two jobs, so the neighbourhood of $x = (3,4,2,1)$ is as shown in Table 5.2.

Schedule	Makespan
(4,3,2,1)	35
(2,4,3,1)	35
(1,4,2,3)	37
(3,2,4,1)	39
(3,1,2,4)	39
(3,4,1,2)	36

Table 6.2: Neighbourhood of solution $x = (3,4,2,1)$

We move the solution to $s(x) = (4,3,2,1)$. The neighbourhood of $x = (4,3,2,1)$ is as shown in Table 6.3.

Schedule	Makespan
(3,4,2,1)	36
(2,3,4,1)	44
(1,3,2,4)	37
(4,2,3,1)	35
(4,1,2,3)	37
(4,3,1,2)	33

Table 6.3: Tabu list after the first iteration

We move the solution to $s(x) = (4,3,1,2)$. The neighbourhood of $x = (4,3,1,2)$ is as shown in Table 6.4.

Schedule	Makespan
(3,4,1,2)	36
(1,3,4,2)	34
(2,3,1,4)	40
(4,1,3,2)	33
(4,2,1,3)	37
(4,3,2,1)	35

Table 6.4: Tabu list after the second iteration

In the neighbourhood, there is no solution better than $x = (4,3,1,2)$. So the algorithm terminates, and returns the optimal schedule $x = (4,3,1,2)$.

6.2.2 Principles of Tabu search

TS algorithm uses neighbourhood search to iteratively move from the current solution to another solution in the neighbourhood of current solution, until reaching the terminal conditions. However, the major disadvantage of the neighbourhood search is that it is easy to fall into local optimal solutions. To circumvent this problem, TS accepts an inferior solution, which means that the obtained solution is not necessarily the best in the neighbourhood.

On the other hand, once the inferior solution is accepted, the algorithm may fall into an infinite loop. To avoid this, the algorithm puts the moves recently accepted into the tabu list and this move cannot be accepted as the initial solution of the next iteration. In its simplest form, a tabu list contains the solutions that have been visited in the recent past. In certain iterations, first, the algorithm will list all the solutions in the neighbourhood and check each solution in turn from best to worst. Then, the algorithm will accept the first best solution, which is not in the tabu list, as the initial solution of the next iteration.

The last important rule of TS is the "pardoned" feature. At the start of the algorithm, we define the size of the tabu list. During the running of the algorithm, more and more visited solutions are put into the tabu list, until it fills up. At this point, when there is a new solution into the tabu list, the solution, which was first put into the tabu list, will be remove, and this solution can be accepted again as the initial solution in the next iterations.

The algorithm that can be derived from above can be summarized as follows:

Step 1: Get an initial solution, and set the tabu list to empty and set the maximal number of iterations.

Step 2: Construct the neighbourhood of the current solution.

Step 3: Choose the best solution in the neighbourhood of the current solution that is not in the tabu list as the current solution.

Step 4: Update the tabu list (add the solution most recently visited and pardon if needed).

Step 5: If algorithm reaches the maximal number of iteration, then stop and output the solution. Otherwise go to Step 2.

6.3 Particle Swarm Optimization (PSO)

In a group of organisms, cooperation and competition between individuals develop the swarm intelligence. Inspired by these, [Kennedy and Eberhart, 1995] proposed a new method called Particle Swarm Optimization. The algorithm is based on simulating the simplified social model and swarm intelligence theory. Through cooperation and competition between individuals, every particle updates its own position. Finally the algorithm returns the swarm-best-position as the optimal solution. In this section, we present the principles and the process of building PSO.

6.3.1 Principles of PSO

A swarm, consisting of m particles, fly in the space of solutions at certain speeds and direction. First, PSO randomly selects several initial positions (called particles) to form a swarm in the space of solution, with an associated initial velocity for each of them. Then, PSO records these initial positions as the self-best-positions of each particle. By comparing the value of the objective function of each particle's self-best position, the best one is retained as the swarm-best-position (as competition). The model of a swarm is described by Figure 6.1. At each iteration, according to the "self-best-position" and the "swarm-best-position", PSO first calculates the new velocity of each particle, and

then uses the new velocity to update each particle's position (as cooperation); and then evaluates every new position. If the new one is better, it updates the self-best-position. Furthermore, if the best of the new positions is better than the current swarm-best-position, it updates the swarm-best-position. When the algorithm terminates, swarm-best-position is returned as the optimal solution.

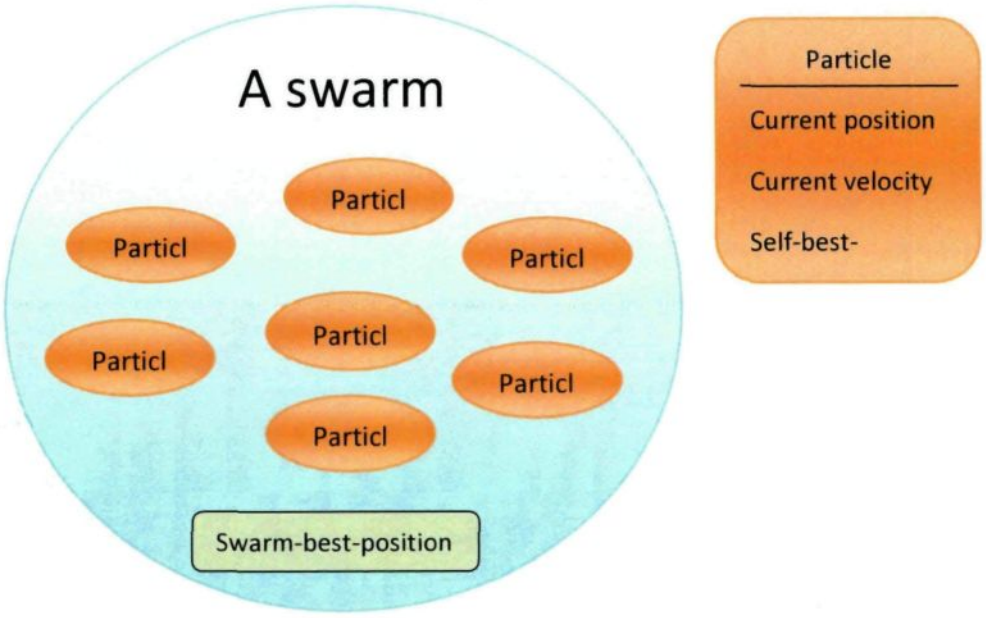


Figure 6.1: Model of the swarm of PSO

In d-dimensional search space, the position and velocity of particle i are

$$X_i = [x_{i,1}, x_{i,2}, \dots, x_{i,d}],$$

$$V_i = [v_{i,1}, v_{i,2}, \dots, v_{i,d}].$$

As the objective function evaluates each particle's position $P = [p_{i,1}, p_{i,2}, \dots, p_{i,d}]$, it finds the self-best-position of each particle $p_b = [p_{b,1}, p_{b,2}, \dots, p_{b,d}]$ and the swarm-best-position $p_{g=}[p_{g,1}, p_{g,2}, \dots, p_{g,d}]$. Then, the following are the formulas we used to update the position and velocity for each particle:

$$v_{i,j}(t+1) = \omega v_{i,j}(t) + c_1 r_1 [p_{b,j} - x_{i,j}(t)] + c_2 r_2 [p_{g,j} - x_{i,j}(t)],$$

$$v_{min} \leq v_{i,j}(t+1) \leq v_{max},$$

$$x_{i,j}(t+1) = x_{i,j}(t) + v_{i,j}(t+1), here\ j = 1, \dots, d.$$

Here, ω is the inertia factor, c_1 and c_2 denote the constant acceleration, r_1 and r_2 are random numbers, in the uniform distribution between 0 and 1. To avoid particles flying out of the search space, we need to define a velocity range $[v_{min}, v_{max}]$. The method of updating is described by Figure 6.2.

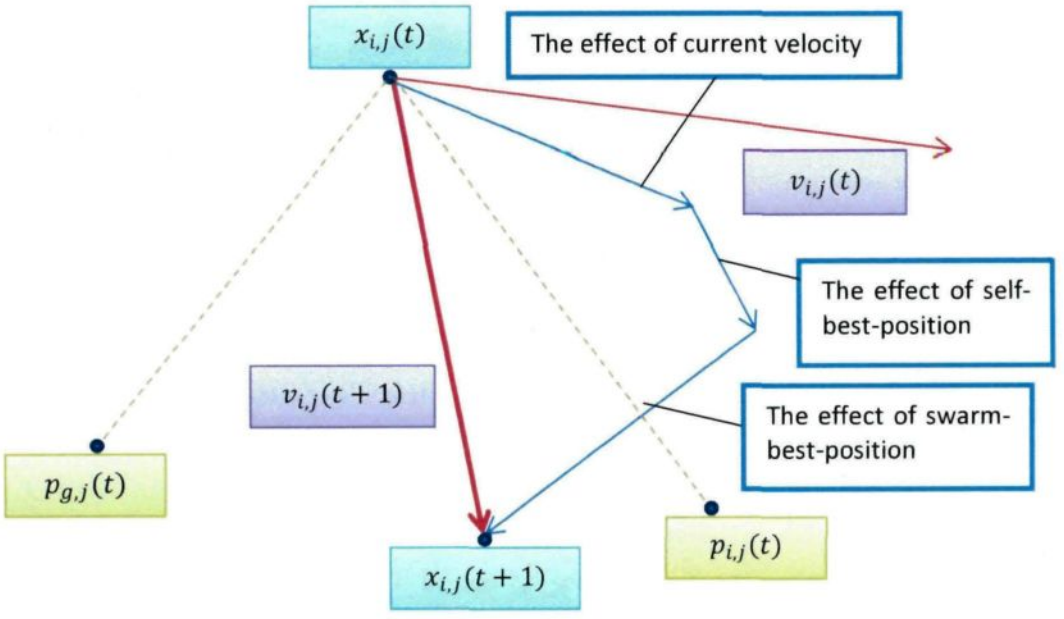


Figure 6.2: Updating the particle

However, in practice, we often have difficulty to determine the maximum velocity and minimum velocity. And, even if we can, after each update, it is not easy to find a reasonable way to handle the situation of the velocity cross-border.

To solve this problem, [Clerc and Kennedy, 2002] proposed the following new formula to update the velocity:

$$v_{i,j}(t+1) = \chi\{v_{i,j}(t) + c_1 r_1[p_{b,j} - x_{i,j}(t)] + c_2 r_2[p_{g,j} - x_{i,j}(t)]\},$$

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \varphi = c_1 + c_2, \varphi > 4.$$

Where χ is the compression factor; it ensures the convergence of the algorithm, and cancels the velocity limit. In the following, we will use this new formula.

6.3.2 Process of PSO algorithm

The algorithm that can be derived for PSO can be described as follows:

Step 1: Initialize the initial position and initial velocity of each particle.

Step 2: In the evaluation of each particle:

2.1 store the position and value of the objective function of this position as "self-best-solution";

2.2 Store the best one of "self-best-position" as "swarm-best-position".

Step 3: Use the "self-best-position" and "swarm-best-position" to update the velocity and position of each particle.

Step 4: Use new velocity to update the position of each particle.

Step 5: Compare the value of the objective function of the new position with value of the "self-best-position" for each particle. If the first is better, then update "self-best-position".

Step 6: Compare the best of all new "self-best-positions" with "swarm-best-position". If the first is better, then update "swarm-best-position".

Step 7: If the termination conditions are fulfilled, then return the "swarm-best-solution". Otherwise, go to Step 3.

6.4 Encoding

Depending on the type of the solution space, the combinatorial optimization problem can be divided into two kinds: continuous problem and discrete problem. In the literature, for the FSP, we usually use the order of the jobs as the encoding of the problem. Obviously, this approach makes discrete the solution space. However, for PSO, the location of the particles is a continuous-valued vector. The major difficulty in applying any version of PSO to combinatorial optimization problems is its continuous nature. The standard PSO algorithm cannot update the order of the jobs, although [Kennedy and Eberhart, 1995] designed the other two versions of the PSO using binary encoding and sequence encoding. However, according to my tests, the practical effect of these two versions is not ideal. To remedy this disadvantage, ranked-order-value (ROV) is usually utilized in PSO to convert the continuous position values into a discrete job permutation. The continuous position vector of the particle is transformed to the order of processing of the jobs, and then we calculate the makespan of the schedule, which is expressed by the position of the particles. In this way, there is no need to modify the operation of the update of the position of particles of the PSO algorithm, and we can ensure the feasibility of the schedule.

The ROV rules are as follows:

For a particle vector position, we first assign ROV value 1 to the smallest component of position vector. Then, the second smallest component is assigned as 2, and so on, until every component is given one unique ROV value. Thus, based on the ROV value, we construct an order of jobs. Let us note that we take into account the fact that there may

be multiple components that might have the same value in the position vector of a particle. If it is the case, then we add a sufficiently small positive number to the value of these locations, make them to be different from each other and do not affect the original job order. It is worth mentioning that in the simulation experiment as presented in Section 6.5.3, if the value of the component is a double-precision real number, when tracking for multiple tests, there have been no cases where multiple components have the same value.

In Table 6.5, we use a sample example to explain the principle and process of the ROV rule. Thinking about a standard FSP with 6 jobs, the position of the particles is a 6-dimensional vector. Suppose that the position vector is

$$X_i = [2.78, 0.04, 1.56, 1.87, 4.92, 0.45].$$

First, giving a ROV value 1 to the minimal component $x_{i,2}$, we next give a ROV value 2 to $x_{i,6}$. Then, respectively, we assign the ROV values to $x_{i,1}$, $x_{i,3}$, $x_{i,4}$, $x_{i,5}$, Finally, we get the order of processing of jobs as $S = (1, 5, 3, 6, 4, 2)$.

Position of component	1	2	3	4	5	6
Component value	2.78	0.04	1.56	1.87	4.92	0.45
ROV value	5	1	3	4	6	2

Table 6.5: Position of the particle and the corresponding ROV value

In this section, our goal is to design a hybrid meta-heuristic with PSO and TS. So, not only we need to see how to apply ROV for PSO, but also to see whether it is appropriate for TS. As presented above, the main use of neighbourhood search techniques in TS differs from the PSO, as these operations do not directly affect the particle's position

vector but act on the order of jobs. Therefore, each time we perform a neighbourhood search operation, the position component of the particles should be adjusted accordingly to ensure the uniqueness of each position's ROV value. Here, we define the neighbourhood of one solution as follows:

The neighbourhood of a solution is a in which all the solutions can be obtained by exchanging once the positions of any two jobs. For a problem with three jobs, one solution is (1, 2, 3) and its neighbourhoods are (2, 1, 3), (3, 2, 1) and (1, 3, 2). If we have four jobs, the size of the neighbourhood of one solution is 6, and if we have *n* jobs, the size of the neighbourhood of one solution is $\frac{n^2-n}{2}$.

However, the tabu list will reduce the size of neighbourhood, so in fact the size of the neighbourhood will be smaller than $\frac{n^2-n}{2}$. Under the ROV rule, after a neighbourhood search, the adjustment of the particle position vector is very simple.

Table 6.6 shows an example. Suppose, before the neighbourhood search, the position vector of the particle is $X_i = [2.78, 0.04, 1.56, 1.87, 4.92, 0.45]$. The order of jobs based on the ROV rules is $S = (1, 5, 3, 6, 4, 2)$. The neighbourhood search operation exchanges the position of component $x_{i,3}$, $x_{i,4}$. Then the new order of jobs is $S' = (1, 5, 6, 3, 4, 2)$.

Obviously, in order to make the position vector of the particle correspond to a new order of jobs, we need to swap the position components 1.56 and 1.87.

Position of component	1	2	3	4	5	6
Component value	2.78	0.04	**1.56**	**1.87**	4.92	0.45
ROV value	5	1	**3**	**4**	6	2
Position of component	1	2	3	4	5	6
Component value	2.78	0.04	**1.87**	**1.56**	4.92	0.45
ROV value	5	1	**4**	**3**	6	2

Table 6.6: After SWAP operations, the particle component position is adjusted

6.5 Hybridization

The hybrid PSO is the main contribution of this chapter. As it is well known, every meta-heuristic has its own advantages and disadvantages. The reason for combination is to introduce the operation of another algorithm into the modified algorithm as this is expected to improve performance. In this chapter, we will use the PSO algorithm as the basic one, and we will introduce some characteristics of TS to improve the performance of the PSO algorithm. This chapter is divided into two parts: in the first part, we will be comparing the advantages and disadvantages of PSO and TS, and then try to find a correspondence between them, which we can use as the basis of our combined operation. In the second part, we will illustrate the details of the combined steps and finally put forward a kind of framework and ideas on improving PSO.

6.5.1 Features of TS and PSO

Before we mix the two meta-heuristics, first of all, we need to look at their characteristics and analyse their strengths and weaknesses. In doing so, we will discover that TS and PSO have complementarities. This is why we combined these two. In Table

6.7, we can clearly see the characteristics of the two algorithms. By comparing these characteristics, we may easily see the theoretical basis for mixing the two algorithms.

Points	TS	PSO
Complexity	In each iteration, the algorithm must update the tabu list and every time the search of the current solution neighbourhood is needed. So, the algorithm is more complex.	The algorithm operation is relatively simple; There is only a need to change the position of the particle according to the velocity and record the best solution.
Local search	The neighbourhood of the current solution is searched in each iteration, so the TS algorithm has a good local search ability.	Does not have the local search capabilities.
Global search	Every time, only one solution is operated, so the global search ability of TS is weak.	There are more particles cooperating in the search of the solution space. So, PSO has a strong global search capability.
Parameter-dependent	Performance of the algorithm is dependent on the length of the tabu list.	Algorithm performance is dependent on the number of particles: The greater number is, the better is the performance of the algorithm.
Dependence on the initial solution solution	The performance of the algorithm does not heavily depend on the quality of the initial solution**.	High dependence on the initial solution
Local optimal solution	The algorithm has the ability to escape from the local optimum.	Since each particle running direction tends to the current optimal solution, the algorithm will easily fall into a local optimal solution
Redundant operations	To a large extent due to the existence of the tabu table, TS avoids repeatedly solving the same case	The algorithm has no mechanism for avoiding redundant operations; it is possible to repeatedly solve the same cases.
Parallelism	No	good parallelism

**For more details, see e.g. [Dingwei *et al.*, 2007]

Table 6.7: Characteristics of TS and PSO approaches

6.5.2 Design of the hybrid algorithm

Let us recall that the two most important capabilities of meta-heuristic are their ability to do a global search and escape local optimal solutions. Through its tabu list, TS has a good capability to fall into the local optimal solution. However, its global search is not that effective compared to other meta-heuristic algorithms (for more details, see for e.g. [Wenxun and Jinxing, 2005]). PSO, on the other hand, randomly distributes the initial positions of the particles in the space of solution. This enables it to do an effective global search. However, its strength is not on escaping local optimal solution. The main ideas of mixing TS and PSO are as follows:

1. Use a heuristic algorithm to produce an initial solution.
2. After updating a position, each particle should search the neighbourhood of the new position. This makes that every particle has broader search range.
3. After the neighbourhood searching, every particle accepts an inferior solution. This means that the particles will not prematurely converge to the swarm-best-position.
4. To avoid falling into an infinite loop, the algorithm puts recent passing positions into the tabu list.

In this chapter, we let all the particles share one tabu list named "Public Tabu List". Here, we define the size of the tabu list as a multiple of the number of particles, and the tabu list will follow the "Unified Pardon Rule". This means that the positions, which were first put into the tabu list, are first pardoned. Let us mention that there is another rule of pardon, called "Prioritized Pardon Rule". In this case, if a position, in the tabu list,

is used to be a "swarm-best-position", it should have the priority pardon, regardless of which particle was first added into the tabu list. Example 6.2 illustrates these two rules of pardon. However, in the simulation we undertook in Section 6.5.3, we used the "Unified Pardon Rule". Our choice for this rule is its simplicity in the implementation.

Example 6.2: In the swarm, we have three particles *a, b* and *c*, and the initial positions of these particles are $p_{a,1}, p_{b,1}$, and $p_{c,1}$. We assume that $p_{a,1}$ is the best one as the swarm-best-position. Table 6.8 shows the tabu list and swarm.

null	null	null	null	null	null

The current positions of $p_{a,1}, p_{b,1}$, and $p_{c,1}$.

Swarm-best-position: $p_{a,1}$

Table 6.8: Initial tabu list and swarm

After the first iteration, three particles move to their next positions $p_{a,2}, p_{b,2}$ and $p_{c,2}$, here, we support that $p_{c,2}$ is better than $p_{a,1}$, so the swarm-best-position is replaced by $p_{c,2}$, and then the positions $p_{a,1}, p_{b,1}$ and $p_{c,1}$ are added into the tabu list as Table 6.9.

$p_{a,1}$	$p_{b,1}$	$p_{c,1}$	null	null	null

The current positions of $p_{a,2}, p_{b,2}$, and $p_{c,2}$.

Swarm-best-position: $p_{c,2}$

Table 6.9: The tabu list and swarm after first iteration

After the second iteration, three particles move to their next positions $p_{a,3}, p_{b,3}$ and $p_{c,3}$, and let us assume that $p_{b,3}$ is better than $p_{c,2}$. So the swarm-best-position is replaced by $p_{b,3}$, and then the positions $p_{a,2}, p_{b,2}$ and $p_{c,2}$ are added into the tabu list as shown in Table 6.10.

$p_{a,2}$	$p_{b,2}$	$p_{c,2}$	$p_{a,1}$	$p_{b,1}$	$p_{c,1}$

The current positions of $p_{a,3}, p_{b,3}$, and $p_{c,3}$.

Swarm-best-position: $p_{b,3}$

Table 6.10: The tabu list and swarm after second iteration

After the third iteration, three particles move to their next positions $p_{a,4}, p_{b,4}$ and $p_{c,4}$. Any new position is better than $p_{b,3}$, so the swarm-best-position is still $p_{b,3}$. Next, we need to add the positions $p_{a,3}, p_{b,3}$ and $p_{c,3}$ into the tabu list, but it is full now. So, three positions should be pardoned.

If we use the "Unified Pardon Rule", the positions $p_{a,1}, p_{b,1}$ and $p_{c,1}$ will be pardoned since they were first added into the tabu list as shown in Table 6.11.

$p_{a,3}$	$p_{b,3}$	$p_{c,3}$	$p_{a,2}$	$p_{b,2}$	$p_{c,2}$

The current positions of $p_{a,4}, p_{b,4}$ and $p_{c,4}$.

Swarm-best-position: $p_{b,3}$

Table 6.11: The tabu list and swarm (Unified Pardon Rule)

But, if we used the "Prioritized Pardon Rule", the positions $p_{a,1}, p_{c,2}$ and $p_{c,1}$ are those which are pardoned, since $p_{a,1}, p_{c,2}$ were used to be the swarm-best-positions. Thus, they should preferentially be pardoned, and $p_{c,1}$ is first added into the tabu list as shown in Table 6.12.

$p_{a,3}$	$p_{b,3}$	$p_{c,3}$	$p_{a,2}$	$p_{b,2}$	$p_{b,1}$

The current positions of $p_{a,4}, p_{b,4}$ and $p_{c,4}$.

The Swarm-best-position: $p_{b,3}$

Table 6.12: The tabu list and swarm (Prioritized Pardon Rule)

Following are the different steps of the Hybrid Algorithm.

Step 1: Initialization

Step 1.1: Initialize the two special particles by the heuristics we designed before.

Step 1.2: Randomly initialize the *m-2* particles and form a swarm with *m* particles.

Step 1.3: Calculate the objective value of each particle

Step 1.4: Each particle's "self-best-position" equals its current objective value, so the "swarm-best-position" equals to the best of all objective values of particles.

Step 1.5: Deposit the current position of all particles into the tabu list.

Step 2: Updating

Step 2.1: According to the velocity of the particle, update the particle position.

Step 2.2: Search the neighbourhood of the particle; update the position of the particle again by the best position in the neighbourhood, and not in the tabu list.

Step 2.3: Update the "self-best-position" and "swarm-best-position", and deposit the current position of all particles into the tabu list.

Step 2.4: If the tabu list is full, then pardon *m* positions by the "Unified Pardon Rule".

Step 3: Termination

If the algorithm termination conditions are met, then return the "group-best position" and its objective value. Otherwise, go to Step 2.

6.5.3 Experimental study

The simulation runs in Apple's iMac with Core i3 and 4GB memory and be programmed by Java. We used three different sizes of the input: 50, 100 and 200, for the simulation. For each size, we also designed different sizes of the subsets. For different sizes of the problem and different of subsets, we randomly generated 10000 different sets of jobs. We calculated their makespan values with the standard PSO and the hybrid PSO, and take the averages of makespan values, respectively.

In the simulation presented below, for both standard PSO and hybrid PSO, the sizes of swarm are 20. Tables 6.13.1 to 6.13.3 show the results generated by the experimental study we performed. In Table 6.13.1, the processing times of each job are random positive integers drawn from [50,100}, [20,100], [10,100], respectively, in Table 6.13.1, 6.13.2, and 6.13.3. The first column of each table shows the scale of the problem and the size of the subsets; the second column is the average of the lower bound on the values of the makespan; the third column is the average of the makespan values obtained by the standard PSO, and the last column is the average of the makespan values obtained by the hybrid PSO.

As mentioned Section 6.5.3, after using the neighbourhood search, hybrid PSO can search a wider range of the solution space. As we can see in the different tables, the different values of the makespan obtained by the hybrid PSO are closer to the lower bound than the standard PSO.

Processing time between 50-100		Standard PSO	Hybrid PSO
The sizes of subsets	Average of lower bound	Average of the results(AR)	Average of the results(AR)
35×15	3775.784	3783.591	3776.012
25×25	3780.015	3788.594	3780.995
70×30	7500.667	7505.215	7500.752
60×40	7501.033	7509.254	7501.945
50×50	7503.009	7510.216	7503.654
150×50	14942.952	14955.651	14943.055
120×80	14939.785	14945.959	14939.951
100×100	14944.354	14960.267	14944.637

Table 6.13.1: Simulation of hybrid PSO where processing times are in [50, 100]

Processing time between 20-100		Standard PSO	Hybrid PSO
The sizes of subsets	Average of lower bound	Average of the results(AR)	Average of the results(AR)
35×15	2999.322	3020.75	3000.1
25×25	2990.15	3025.521	2990.788
70×30	5970.403	5980.201	5970.546
60×40	5975.886	5979.743	5976.156
50×50	5973.25	5981.528	5973.376
150×50	11920.188	11925.002	11920.914
120×80	11923.995	11930.59	11924.415
100×100	11919.484	11922.198	11919.517

Table 6.13.2: Simulation of hybrid PSO where processing times are in [20, 100]

Processing time between 10-100		Standard PSO	Hybrid PSO
The sizes of subsets	Average of lower bound	Average of the results(AR)	Average of the results(AR)
35×15	2730.345	2737.524	2730.732
25×25	2729.528	2736.778	2730.257
70×30	5455.336	5460.023	5455.889
60×40	5460.452	5463.355	5460.998
50×50	5469.057	5476.074	5469.909
150×50	10910.897	10925.256	10911.592
120×80	10896.523	10929.457	10896.833
100×100	10904.006	10909.854	10904.318

Table 6.13.3: Simulation of hybrid PSO where processing times are in [10, 100]

Conclusion

This dissertation addresses the Two-Stage flow-shop scheduling problem with a shared machine denoted by $F3|M_{1,1} \rightarrow M_{2,1}; M_{1,1} \rightarrow M_{2,2}|C_{max}$. In this model, the set J of n independent jobs is partitioned into two disjoint subsets J_1 and J_2. The jobs in the first subset J_1 must be processed first on machine $M_{1,1}$, then on machine $M_{2,1}$; whereas the second subset J_2 must be processed on machine $M_{1,1}$ and then on machine $M_{2,2}$. Machines $M_{2,1}$ and $M_{2,1}$ operate somehow in parallel; they constitute the second stage of the model, whereas machine $M_{1,1}$ constitutes the first stage and shared by J_1 and J_2.

Although, the standard model is NP-Hard, the problem becomes may be polynomially solvable when we relax some constraints. Within this context, we have looked at several special cases: constant processing times, large processing times in Stage one, large processing times in Stage two.

In the heuristic front, we have designed two new heuristic: one is based on John's rule, and the other one is based on Nawaz-Enscore-Ham algorithm. According to the simulation experiment we performed, we observed that the performance of the two heuristic algorithms is good, as most of the time both algorithms produce an optimal solution. It is noteworthy to observe that, as the input size of the problem gets larger, the performance of the two algorithms steadily gets much better when the size of the problem is over 200.

Regarding the meta-heuristic approach, we studied two kinds of meta-heuristic algorithms namely the tabu search method (TS) and the particle swarm optimization

(PSO) method. By comparing the characteristics of the two algorithms, we designed a hybrid meta-heuristic to improve the performance of PSO, According to the simulation experiment we performed we found out that the performance of the hybrid PSO is by far better than the standard PSO and TS approaches.

For future research, it would be interesting to search for other pertinent conditions that make the standard model polynomially solvable. For the hybrid meta-heuristic algorithm, we still have room for improvement. For instance, we could try to test whether the "Prioritized Pardon Rule" and "Private Tabu List" are more efficient than the one we used in this dissertation. We could also consider the question of using the "Unified Pardon Rule" within the "Private Tabu List".

References

[Balas and Toth, 1985] Balas, E., and Toth, P. Branch and bound methods, chapter 10, in Lawler, E.L. *et al.* (eds): *The traveling salesman problems: a guided tour of combinatorial optimization*, John Wiley, 1985.

[Cook, 1971] Cook, S. The complexity of theorem proving procedures, *Proceedings of the3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158. 1971.

[Dasgupta *et al.*, 2007] Dasguna, S., Papadimitriou, CH., Vazirani, UV. *Algorithms*, Chapter 6, John Wiley.

[Campbell, Dudek and Smith, 1970] Campbell, H., Dudekp, R., Smith, M. A heuristic algorithm for the n-job m-machine sequencing problem, *Management Science*, vol. 16 (10), pp. 630-637, June 1970.

[Dannenbring, 1977] Dannenbring, D. *An* evaluation of flow-shop sequencing heuristics, *Management Science*, vol. 23(11), pp. 1174-1182, 1977.

[Dingwei et al., 2007] Dingwei, W., Junwei, W., Hongfeng, W., Ruiyou, Z., *Intelligent Optimization Methods*, Higher Education Press, pp.81-84, 2007

[Dorigo, 1992] Dorigo, M., *Optimization, Learning, and Natural Algorithms*, Phd Thesis, Politecnico di Milano, Italy, 1992.

[Fatos and Ajith, 2008] Fatos, X., Ajith, A. *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, vol. 128, Studies in computational intelligence, 2008.

[Fisher, 1982] Fisher, M.L. Worst case analysis of heuristic algorithms for scheduling and packing, in Dempster *et al.* (eds): *Deterministic and stochastic scheduling*, D. Reidel

Publishing Co, 1982.

[Gantt, 1910] Gantt, HL. Work, Wages and Profit, *The Engineering Magazine*, 1910.

[Garey and Johnson, 1979] Garey, MR., Johnson, DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Computer Press, 1979.

[Glover, 1990] Glover, F. Tabu Search – Part 2. *ORSA Journal on Computing*, pp. 4-32, 1990.

[Glover, 1989] Glover, F. Tabu Search – Part 1. *ORSA Journal on Computing*, pp. 190-206, 1989.

[Glover, 1986] Glover, F. Future Paths for Integer Programming and Links to Artificial Intelligence, *Computers and Operations Research*, vol. 13 (5), pp. 533–549, 1986.

[Glover, 1977] Glover, F. Heuristics for Integer programming: Using Surrogate Constraints, *Decision Sciences*, vol. 8 (1), pp. 156–166, 1977.

[Graham *et al.*, 1979] Graham RL., Lawler EL., Lenstra JK., Rinnooy Kan, AHG. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, *Annals of Discrete Mathematics*, vol. 5, pp. 287-326, 1979.

[Gupta, 1971] Gupta, J. A functional heuristic algorithm for the Flow-shop scheduling problem, *Operations Research Quarterly*, pp. 39-47, 1971.

[Holland, 1975] Holland, JH. *Adaptation in Natural and Artificial Systems*, 1975.

[Johnson, 1954] Johnson, SM. Optimal Two-and-Three-Stage Production with Setup Times Included, *Naval Research Quarterly*, pp. 1-5, 1954.

[Karp, 1972] Karp, R. Reducibility among combinatorial problems, in R.E. Miller and J.W. Thatcher (eds): *Complexity of Computer Computations*, Plenum Press, New York.

[Kennedy and Eberhart, 1995] Kennedy, J., Eberhart, R. Particle Swarm Optimization, *Proceedings of IEEE International Conference on Neural Networks,* pp. 1942-1948, 1995.

[Kirkpatrick Gelatt and Vecchi, 1983] Kirkpatrick, S., Gelatt JR., Vecchi, MP. Optimization by Simulated Annealing, *Science*, vol. 220 (4598), pp. 671–680, 1983.

[Levitin, 2003] Levitin, A. *The design and analysis of algorithms*, Addison Wesley, 2003.

[Mellor, 1966] Mellor, P. A review of job shop scheduling, *Operational Research Quaterly,* vol. 17, pp. 161-171, 1966.

[Nawaz Enscore and Ham, 1983] Nawaz, M., Enscore Jr., E.E., Ham, I., A heuristic algorithm for the m-machine n-job flow-shop sequencing problem, *Omega-International Journal of Management Science,* vol. 11(1), pp. 91–95, 1983.

[Palmer, 1965] Palmer, D. Sequencing jobs through a multi-stage process in the minimum total time: a quick method of obtaining a near optimum, *Operations Research Quarterly,* pp. 101-105, 1965.

[Raghavachari, 1988] Raghavachari, M. *Scheduling with non-regular penalty functions: a review*, Operations Research, vol. 25, pp. 144-164, 1988.

[Rinnooy Kan, 1986] Rinnooy Kan, AHG. Probabilistic analysis of approximation algorithms, *Annals of Discrete Mathematics*, vol. 31, pp. 153-162, 1986.

[Serope and Steven, 2006] Serope, K., Steven, S. *Manufacturing Engineering and Technology* (5th ed.), Prentice Hall, 2006.

[Tuong Soukhal and Miscopein, 2009] Tuong, NH. Soukhal, A. and Miscopein, L. Interfering job set scheduling on three-stage flow-shop with a common stage machine, *MISTA*, pp. 10-12, 2009.

[Toni and Tonchia, 1998] Toni, A. and Tonchia, S., Manufacturing Flexibility: a literature review, *International Journal of Production Research*, 1998, vol. 36(6), pp. 587-617, 1998.

[Wenxun and Jinxing, 2005] Wenxun, X., Jinxing, X., *Modern optimization methods*, Tsinghua University Press, 2005.

[Yagiura and Ibaraki, 2001] Yagiura, M., Ibaraki, T. On meta-heuristic algorithms for combinatorial optimization problems, *Systems and Computers in Japan*, Vol. 32(3), pp. 33-55, 2001.